Copyright © 2004, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

LATENCY-INSENSITIVE DESIGN

by

Luca Carloni

Memorandum No. UCB/ERL M04/29

3 August 2004

LATENCY-INSENSITIVE DESIGN

by

Luca Carloni

Memorandum No. UCB/ERL M04/29

3 August 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Latency-Insensitive Design

by

Luca Carloni

Laurea (Università di Bologna, Italy) 1995
M. S. (University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alberto L. Sangiovanni-Vincentelli, Chair
Professor A. Richard Newton
Professor John H. Freeman

Fall 2004

The dissertation of Luca Carloni is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

Fall 2004

Latency-Insensitive Design

Copyright 2004

by

Luca Carloni

Abstract

Latency-Insensitive Design

by

Luca Carloni
Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

Nanometer process technologies make hundreds of millions of transistors available for the design of an entire system on a single chip (system-on-chip). However, designs of this kind expose problems that were barely visible at previous levels of integration.

First, despite the increase in number of metal layers and in aspect ratio, the resistance-capacitance (RC) delay of an average metal line with constant length is getting worse with each process generation. This fact, combined with the increases in operating frequency and average interconnect length, cause on-chip interconnect delay to become the largest fraction of the clock cycle time. To make things worse, it is increasingly difficult to estimate early in the design process the actual interconnect latency because it is affected by several phenomena, like process variations and crosstalk, whose combined effect may vary across chip regions and periods of operation. As a result, nanometer technologies are forcing the semiconductor industry to experience a paradigm shift from computation- to communication-bound design: the number of transistors that a signal can reach in a clock cycle—not the number that designers can integrate on a chip—drives the design process.

Second, to manage the design complexity of a system-on-chip (SOC), the effective reuse of existing intellectual-property (IP) components is essential. Originally, IP cores were mostly functional blocks built for previous design generations within the same company. Frequently, today's IP cores are optimized modules marketed as off-the-shelf components by specialized vendors. The prerequisite for an easy trade, reuse and assembly of IP cores is the ability to combine pre-designed components with little or no effort. The consequent challenge is addressing the communication and synchronization issues that naturally arise while assembling pre-designed components.

Currently available computer-aided design (CAD) tools struggle on handling the increasingly dominant impact of interconnect delay and fall short on providing support for IP reuse. With each process generation, the number of available transistors grows faster than the ability to meaningfully design them (design productivity gap) and designers are forced to iterate many times between circuit specification and layout implementation (timing-closure problem). Ironically, it is the introduction of nanometer technologies that threatens the outstanding pace of technological progress that has shaped the semiconductor industry.

The key to addressing these challenges is the development of methodologies based on formal methods to enable modularity, flexibility, and reusability in system design. The subject of this dissertation—Latency-Insensitive Design—is a step in this direction. My thesis is that "*correct-by-construction methods combining the benefits of synchronous specification with the efficiency of asynchronous implementation are the key to design moderately distributed complex systems composed of tightly interacting concurrent processes.*"

Major contributions are the theory of latency-insensitive protocol and the companion latency-insensitive design methodology. Latency-insensitive systems are synchronous distributed systems composed by functional modules that exchange data on communication channels according to an appropriate protocol. The protocol works on the assumption that the modules are stallable (a weak condition to ask them to obey) and guarantees that systems made of functionally correct modules, behave correctly independently of channel latencies. The theory of latency-insensitive protocols is the foundation of a correct-by-construction methodology for integrated circuit design that handles latency's increasing impact on nanometer technologies and facilitates the assembly of IP cores for building complex SOCs, thereby reducing the number of costly iterations during the design process. Thanks to the generality of its principles, latency-insensitive design can be possibly applied to other research areas like distributed deployment of embedded software.

Professor Alberto L. Sangiovanni-Vincentelli
Dissertation Committee Chair

Alla mia cara Mê

Contents

List of Figures

List of Tables

Acknowledgements

"Blind hopes in them I made to dwell."
AESCHYLUS. PROMETHEUS BOUND.

During a late winter afternoon, back in December 1994, I met for the first time professor Alberto Sangiovanni-Vincentelli. I was a visiting undergraduate student from the University of Bologna, who spoke broken English while trying to cope with the diverse challenges of UC Berkeley Exchange Abroad Program. The meeting lasted less than ten minutes and changed my life. Eighteen months later, I would be starting Graduate School at Berkeley, thus guaranteeing to myself the luxury of working together with Alberto for all these years and, hopefully, for many more to come.

I could fill pages writing about Alberto's professional gifts. Gifts that, naturally, turn out to be mainly *presents* for his students: the gift of understanding (predicting seems often a more proper word) which research problems are the key ones to be addressed; the gift of combining a rich scientific talent with uncommon business intuition; the gift of putting the same careful attention to all the stages of a project, from involving the students in the drafting of the research proposal to refining the slides (and the voice intonation!) for their presentations; the (rare) gift of attacking each research problem with the same passion as if it were the first one [1]; the (even rarer) gift of constantly surrounding himself with brilliant collaborators, thus enabling the development of a unique work environment as the Berkeley CAD Group has been for over twenty years. In the end, however, it is Alberto's personal gifts that make it invaluable for his students (and for many less fortunate students, enviable) to work with him: Alberto's genuine interest for the balance between the student's personal and professional development is certainly uncommon, and, in my opinion, likely unmatched. That winter afternoon back in 1994 during our short conversation, I learned, as Musil teaches, *to see a possible experience as a project, something yet to be invented.*

Prof. Robert Brayton was my first teacher at UC Berkeley, when I attended his Logic Synthesis course as an undergraduate student, and has been like a second advisor throughout my master's thesis. As Tiziano Villa once perfectly wrote, Bob remains "a model of

[1] A personal memory should be recorded here, because I suspect that there are not too many advisors who are ready to spend the night after Christmas supper exchanging the drafts of a theorem proof via transatlantic e-mail with a young graduate student who is writing his first paper.

dedication to scholarship and gentleman's style." Since the beginning of graduate school I looked at Prof. Richard Newton as a source of true enthusiasm and inspiring vision. I thank him for his comments on my work and for being part of my Qualifying Committee. Similarly, I thank the other two committee members Prof. Jan Rabaey and Prof. John Freeman. I also want to thank Prof. John Danner, who Socratically taught the *Workshop in Entrepreneurship* at the Haas School of Business during Fall 1999: truly one of the best teaching efforts I experienced at Berkeley.

Besides the professors, however, I believe that what makes graduate school special is the opportunity to learn from senior students. I have been blessed to learn from many of them and I have been trying, in return, to teach something to as many. In fact, I take this occasion to offer my final advice to a first year graduate student: *get a mentor, two is not enough, three is not too many.* During my first couple of years at Berkeley I had the fortune to work with Tiziano Villa (who introduced me to the world of academic research), Alex Saldanha (who introduced me to the world of industrial research), Evguenii Goldberg (who gave me a brilliant idea to work on for my master's thesis), Timothy Kam (whose elegant software packages were the first I read, when I learned that reading good code is the basis for writing good code), and Arlindo Oliveira (with whom I derived my first theorem proof). I just said my final advice, but here I have another one: do summer internships and collaborate with researchers at other institutions. I was lucky enough to have the possibility of spending four consecutive summers at Cadence Berkeley Laboratories, working together with such talented researchers as Alex Kondratyev, Luciano Lavagno, Ken McMillan, and Yoshinori Watanabe; and I was very lucky to collaborate with other talented researchers "across the ocean" who also influenced my research: Albert Benveniste, Benoît Caillaud, and Paul Caspi. And here a final (really final, now) advice: do different things. I believe that Graduate School is meant for engaging in many diverse projects and not for working only on a single problem with the goal of accelerating the coming of that day when your dissertation is ready for the famous three signatures. In my case, the time spent in 1999 attending classes at the Haas School of Business and working together with a team of MBA students for the Business Plan Competition, helped me to broaden the perspectives of my work and made me grow professionally.

When you are completing the following, traditional, *mandatory*, endless list of names of friends and colleagues, you finally suspect that you may have spent just a few too many nights at Cory Hall. In any case, it is always a pleasure to recall fellow CAD Group students together with other members of that unique community which is UC Berkeley: Arthur Abnous, Joern Altmann, Felice Balarin, Alvise Bonivento, Eylon Caspi, Edoardo Charbon, Max Chiodo, Philip Chong, Luca Daniel, Fernando De Bernardinis, Alberto Ferrari, Varghese George, Naji Ghazal, Wilsin Gosti, Heloise Hse, Joe Higgins, Harry Hsieh, Adrian Isles, Sunil Khatri, Desmond Kirkpatrick, Christoforos Kozyrakis, Yuji Kukimoto, Freddy Mang, Amit Merothra, Trevor Meyerowitz, Paolo Miliozzi, Fan Mo, Rajeev Murgai, Amit Narayan, Alessandra Nardi, Georges Pappas, Roberto Passerone, Claudio Pinello, Alessandro Pinto, Vandana Prabhu, Mukul Prasad, Shaz Quadeer, Jacques-Christophe (Chris) Rudell, Marco Sabatini, Marco Sgroi, Niraj Shah, Farhana Sheikh, Narendra Shenoy, Michael Shilman, Tom Shiple, Vigyan Singhal, Subarnarekha Sinha, Mark Spiller, Lixin Su, Iason Vassiliou, Ken Yamaguchi, James Shin Young, and Stefano Zanella. Also, I would like to thank Mary Byrnes and Ruth Gjerde, whose kindness and professionalism make a trip to the Office of Graduate Matters always a pleasure, Brad Krebs, whose skills as system manager saved my work more than once, and, finally, Lorie Brofferio, Susan Gardner, Flora Oviedo, and Jennifer Stone for their administrative assistance. Finally, I am grateful to SRC, GSRC, and NSF which supported me for various research projects throughout graduate school as well as to Cadence Design Systems and Intel Corporation.

Still, and not surprisingly for an Italian, I must say that my biggest gratitude goes to my family, *la famiglia!* Without them, I simply wouldn't be here. Hence, without further ado:

Un forte abbraccio va a mio padre, Giulio Cesare, a mia madre, Anna, e ai miei cari fratelli, Fabio e Marco.

Um abraço com carinho para minha querida familia no Brasil: Emilia Yoko, Max e Denio.

My little princesses Kiara Tainá and Maira Paloma, the sweetest pretexts to postpone the completion of the present effort.

La mia cara Meika Alessandra, who paces the world around me in all directions, thus rendering it its equilibrium and harmony.

To pass freely through open doors, it is necessary to respect the fact that they have solid frames. This principle, by which the old professor had lived, is simply a requisite of the sense of reality. But if there is a sense of reality, and no one will doubt that it has its justifications for existing, then there must also be something we can call a sense of possibility. Whoever has it does not say, for instance: Here this or that has happened, will happen, must happen; but he invents: Here this or that might, could, or ought to happen. If he is told that something is the way it is, he will think: Well, it could probably just as well be otherwise. So the sense of possibility could be defined outright as the ability to conceive of everything there might be just as well, and to attach no more importance to what is than to what is not. The consequences of so creative a disposition can be remarkable, and may, regrettably, often make what people admire seem wrong, and what is taboo permissible, or, also, make both a matter of indifference. Such possibilists are said to inhabit a more delicate medium, a hazy medium of mist, fantasy, daydreams, and the subjunctive mood. Children who show this tendency are dealt with firmly and warned that such persons are cranks, dreamers, weaklings, know-it-alls, or troublemakers. Such fools are also called idealists by those who wish to praise them. But all this clearly applies only to their weak subspecies, those who cannot comprehend reality or who, in their melancholic condition, avoid it. These are people in whom the lack of a sense of reality is a real deficiency. But the possible includes not only the fantasies of people with weak nerves but also the as yet unwakened intentions of God. A possible experience or truth is not the same as an actual experience or truth minus its "reality value" but has - according to its partisans, at least - something quite divine about it, a fire, a soaring, a readiness to build and a conscious utopianism that does not shrink from reality but sees it as a project, something yet to be invented. After all, the earth is not that old, and was apparently never so ready as now to give birth to its full potential.

R. MUSIL.

Chapter 1

Introduction

In which what has been is not overlooked and what will follow is anticipated, at least partially.

PARADIGMS are *"accepted examples of scientific practice—examples which include law, theory, application, and instrumentation together—[that] provide models from which spring particular coherent traditions of scientific research."* This at least according to Kuhn in his classic, and influential, 1962 book [137]. The informal definition, which has been sometimes criticized for being too fuzzy, is centered around the English word that best translates the original Greek παράδειγμα (*paradeigma*), i.e. *example*. Therefore, paradigms are examples. These examples gain their value, which is ultimately a practical value (*"to provide models"*), from offering a combination of a diverse body of information (*"law, theory, application, instrumentation"*). As such, Kuhn's definition applies well to engineering design, particularly in the field of hardware and software systems. In their work engineers naturally follow fundamental laws and theories, but, as they strive to build their systems on time, they regularly find support in those practices, methods and tools which have been applied repeatedly and successfully before.

The present dissertation is very much about the practical importance of a paradigm in designing electronic systems, the paradigm of synchrony, and about the apparent crisis that this paradigm is facing. Though aware that *"all crises begin with the blurring of a paradigm"* and that *"a crisis may end with the emergence of a new candidate for paradigm"* [137], I make an attempt at presenting the old candidate as still a valid one, at least partially.

1.1 The Synchronous Paradigm

The *synchronous paradigm* is ubiquitous in electrical engineering and computer science. It is the basis of digital integrated circuit design, it is used in building discrete-time dynamical control systems, and it is the foundation of programming languages and design environments for real-time embedded systems.

With the synchronous paradigm, a complex system is represented as a collection of *interacting concurrent components* whose state is updated collectively in one instantaneous step. The system consists of a composition of sequential functional processes and evolves through a sequence of *atomic reactions*: at each reaction all processes, simultaneously, read the values of their input variables and use them, together with the values of their state variables, to compute new values for both their state and output variables; between two successive reactions the communication of the computed values across the processes occurs via instantaneous broadcasting.

The *synchronous hypothesis* is precisely the idea that at each reaction the computation in the functional modules and the subsequent communication of the computed values across modules *do not overlap* [1]. In the synchronous paradigm "time" progresses in lock-step, one reaction after the other. The idea of measuring time is confined to the concept of a *virtual*, or *logical*, clock, whose ticking indexes the totally ordered sequence of reactions by associating a new *timestamp* to each reaction (i.e. the set of timestamps coincides with the set of natural numbers).

The power of the synchronous paradigm lies essentially in its simplicity. It is an intuitive, but formal, model of computation [144] that offers many advantages:

- it simplifies the modeling of deterministic concurrent systems;
- it enables the incremental design of complex systems in a modular and hierarchical fashion;
- it facilitates the design process by separating functionality from the notion of time;

[1]Some researchers talk of a *zero-time* step. This is misleading since it is not a matter of measuring computation or communication time, but simply to order subsequent reactions and maintain separated communication from computation.

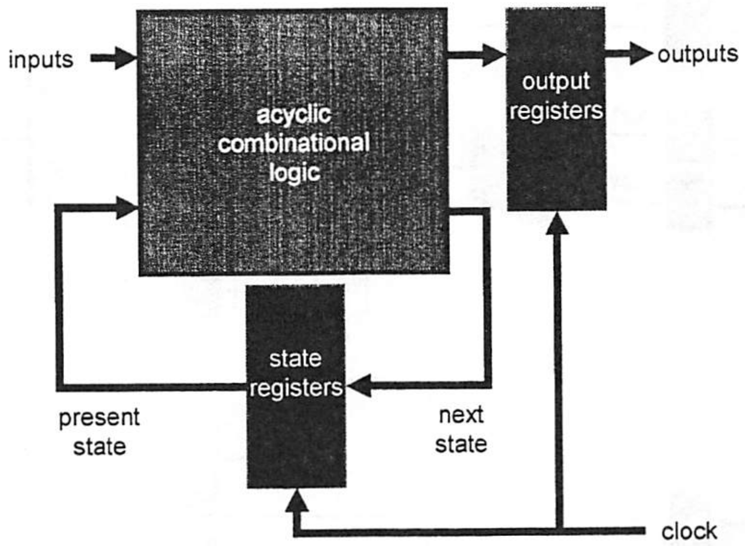

Figure 1.1: Diagram representing a sequential module at the register-transfer level.

- it encourages abstraction and reuse by leading to design specifications that are independent from the details of the particular implementation technology;
- it eases the development of design-automation tools for specification, validation, and synthesis;

1.1.1 Synchronous Paradigm and Hardware Design

In digital hardware design, methodologies and tools based on the synchronous paradigm have made it possible, over the last three decades, to build integrated circuits (IC) that are exponentially more complex and faster. Today's chips are built assembling hundreds of millions of transistors whose concurrent operations are tightly controlled by the beat of a single clock signal (the *physical*, or *real*, clock). Transistors and logic gates, however, are abstracted away during most stages of the design process. The core of the design effort occurs at the *register-transfer level (RTL)* where designers are assisted by hardware-description languages (HDL).

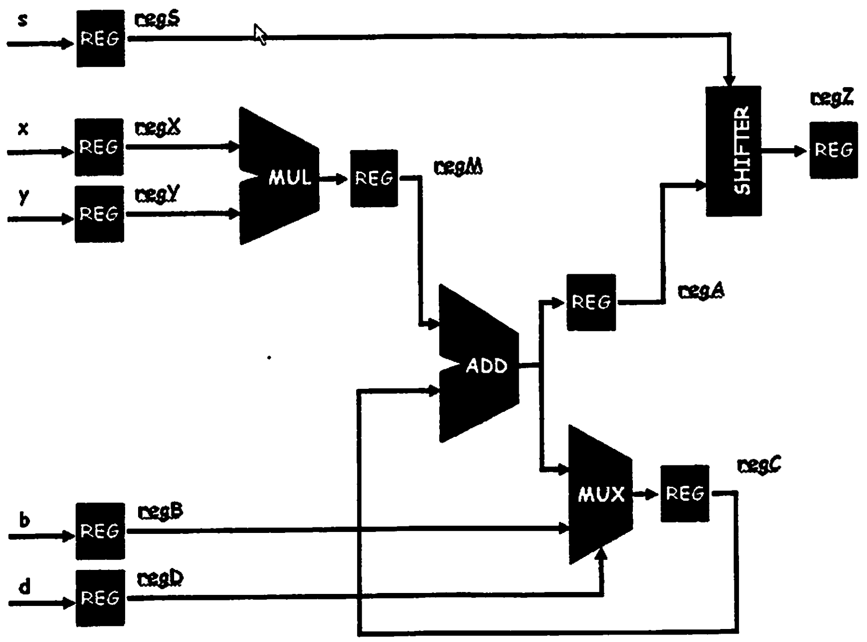

Figure 1.2: RTL block diagram of a MAC circuit.

Figure 1.1 illustrates a sequential module, the basic RTL building block and the ultimate result of applying the synchronous paradigm to IC design. The acyclic combinational logic implements the functionality of the module (an arbitrarily complex arithmetic or logic function) while the registers (memory elements controlled by the clock) store the values of the state and output variables over time [193, 216]. A sequential module is the direct implementation of a finite state machine (FSM) [2], which is the model of computation commonly used to specify control blocks in IC design. Also, any pipelined data-path can be seen as a cascade of sequential modules. Using HDL languages, designers write software programs to specify the functionality of each module as well as their (possibly hierarchical) composition.

Example A multiplier-accumulator (MAC) is a very common digital circuit, because it facilitates the implementation of an operation like $\sum x(n) \cdot y(n-k)$, which is ubiquitous

[2]Strictly speaking the diagram of Figure 1.1 represents the basic implementation of a Moore machine. Removing the output register gives the basic implementation of a Mealy machine [124, 240].

	Input					*Internal*			*Output*
n	*regD*	*regB*	*regY*	*regX*	*regS*	*regM*	*regC*	*regA*	*regZ*
1	0	–	1	2	0	0	0	0	–
2	0	–	0	1	0	2	0	0	–
3	0	–	3	1	0	0	2	2	–
4	0	–	1	2	0	3	2	2	2
5	0	–	2	4	0	2	5	5	2
6	1	10	1	1	0	8	7	7	5
7	0	–	3	2	0	1	10	15	7
8	0	–	1	3	0	6	11	11	15
9	0	–	1	2	0	3	17	17	11
10	0	–	–	–	0	2	20	20	17
11	0	–	–	–	0	–	22	22	20
12	0	–	–	–	0	–	–	–	22

Table 1.1: Example of RTL behavior of the MAC circuit of Figure 1.2.

in filters and vector arithmetic [110]. Fig. 1.2 illustrates a pipelined implementation of a MAC circuit: *inX* and *inY* are the input values to be multiplied, *inD* forces the accumulator register to be preset to the value of *inB*, *inS* controls the shift operation, and, finally, *regZ* is the output representing the sequence of indexed partial sums. The block diagram of the MAC can be decomposed in three sub-circuits and each of them can be designed as a separate RTL module. The sequential behavior of this circuit is captured by the following equations, written in "pseudo-HDL" with $n \in \mathbb{N}$ denoting the timestamp:

$$\begin{aligned}
regM_{n+1} &= regX_n \cdot regY_n \\
regA_{n+1} &= regC_n + regM_n \\
regC_{n+1} &= \begin{cases} regC_n + regM_n & \text{if } regD = 0 \\ regB_n & \text{if } regD = 1 \end{cases} \\
regZ_{n+1} &= shift(regA_n, regS_n)
\end{aligned}$$

Table 1.1 illustrates a possible RTL behavior of the MAC circuit spanning 12 clock cycles. The performance of the circuit is dictated by the delay of the combinational logic of its slowest component, which, in this case, is likely to be the multiplier. ■

The key point in RTL design is the separation of the design and validation of the functional behavior of the system from the analysis and optimization of its performance. The longest combinational path inside a module (*critical path*) dictates the minimum clock period ψ_i that makes it operate correctly. Therefore, given a target period ψ for the nominal clock signal, the task of designing a large digital circuit can be decomposed in sub-tasks aimed to design smaller RTL modules such that $\psi_i < \psi$ for each module i. The modules can be specified, simulated, implemented, and verified independently from each other based only on the desired input/output functionality and the expected value of ψ. The separation of functional design from performance analysis consists of the following: a functionally correct design is obtained by simply assembling all the modules, while its speed is given by the slowest module. In other words, once all modules are composed, the overall circuit works correctly as far as it is running with a clock signal having a period $\psi \geq \max_i\{\psi_i\}$. The effectiveness of this strategy is based on the assumption that the delay of any path connecting two modules (inter-module delay) is negligible, or at most comparable, to the delay of the combinational paths inside the slowest module in the system (intra-module delay). This has been the case for thirty years of progression of semiconductor process technologies as average interconnect inter-module delays have remained insignificant with respect to logic gate delays. Each new process generation has challenged designers to integrate twice the number of transistors running at twice the speed of the previous one, and designers have successfully done so by relying on the simplicity of the synchronous paradigm to build ever more powerful integrated circuits.

The other advantages of the synchronous paradigm mentioned above also translate in the hardware design environment. In particular: the design of the circuit as a deterministic concurrent system is simplified (different modules of the same system can be designed independently and simultaneously by different designers); RTL design is inherently incremental and hierarchical (e.g., portions of the circuit can be redesigned to improve their performance without touching the rest of the system); the RTL design of a module is independent from the particular semiconductor technology used to build the circuits; predesigned RTL modules can be reused for different projects; and, finally, designers can take advantage of a rich offering of commercial computer-aided design (CAD) tools, developed over the last twenty years, for RTL specification, simulation, synthesis, and validation,

1.1.2 Synchronous Paradigm and Embedded Software

Synchronous programming languages [16, 17, 99, 100] like ESTEREL, LUSTRE, and SIGNAL represent powerful instruments for the specification of complex real-time, safety-critical, embedded control systems (e.g., flight-control systems in flight-by-wire avionics and anti-skidding or anti-collision equipment in automotive electronics), which are becoming pervasive in today's society. Designers of such systems can rely on the solid mathematical foundation of synchronous languages to specify and formally validate their designs. They can also use the compilers and code generators that have been developed over the last thirteen years to compile these concurrent programs into embedded software code like executable C or JAVA.

Synchronous languages combine the simplicity of the synchronous hypothesis with the power of deterministic concurrency in functional specification. In doing so, they have their foundation in the synchronous paradigm, which guarantees a common formal semantics. Their shared *synchronous programming model* can be expressed by the following "pseudo-mathematical" statements [13, 15] [3]:

$$\begin{aligned} P &\equiv R^{\omega} \\ P_1 || P_2 &\equiv (R_1 \wedge R_2)^{\omega} \end{aligned}$$

where P, P_1, P_2 denote synchronous programs, R, R_1, R_2 denote the sets of all the possible reactions of the corresponding programs, and the superscript ω indicates non-terminating iterations. The first expression captures the essence of the synchronous paradigm: a synchronous program P evolves according to an infinite sequence of atomic reactions. The second expression denotes the parallel composition of two components as the conjunction of the reactions for each component. In other words, components communicate via shared variables, whose value they must agree upon at each reaction. Hence, parallel composition via conjunction of reactions implies that communication among components is performed via instantaneous broadcast.

Although they share a common mathematical semantics, synchronous languages interpret the synchronous paradigm slightly differently as each language targets its own distinc-

[3] In [15], Benveniste *et. al* discuss how this pseudo-mathematical formulation captures also the composition of block diagrams in control engineering as well as the synchronous product of automata.

```
initialize memory elements;
for each input event do
    compute outputs;
    update memory elements;
end;
```

```
initialize memory elements;
loop each clock tick
    read inputs;
    compute outputs;
    update memory elements;
end;
```

Figure 1.3: Schemes of synchronous execution [10].

tive application area: LUSTRE and ESTEREL follow a *strictly synchronous* approach in their focus on computation-dominated and control-dominated systems, respectively; SIGNAL is a multi-clock language targeting *open systems*, i.e. systems where each component must be designed without knowing the details of its operational environment, including the activity of the other components. In a strictly synchronous model, each variable in the system presents a value at each reaction. This is, for instance, the model for synchronous hardware discussed in the previous section. In a multi-clock synchronous model some variables may be absent at certain reactions. This feature is useful when modeling open systems because it allows us to represent the fact that some components are active while others are silent and synchronization occurs only occasionally. Being a synchronous language, SIGNAL encodes the absence of a variable with the special symbol $\perp$, thereby providing designers with the ability to write programs where decisions based on absence can also be made (another distinctive feature of the synchronous paradigm).

Figure 1.3 shows two typical synchronous execution schemes: event-driven on the left and sample-driven on the right [10]. The bodies of the two loops are examples of atomic reactions: at each reaction, a program sequentially reads input variables, computes output variables and updates its internal register. Moreover, all processes in the system do so concurrently and simultaneously. The execution is then repeated for the next reaction. Data computation (within a reaction) and data communication (across reactions) do not overlap. Clearly, the model of computation of a synchronous program is equivalent to the one of the RTL sequential module illustrated in Figure 1.1. Embedded-software engineers write complex synchronous programs by composing simpler programs having the structure of Figure 1.3 very much in the same way as digital-hardware engineers design complex circuits by assembling simpler RTL modules [4].

The advantages of the synchronous paradigm naturally translate into synchronous programming. In particular [10, 15, 100], the design of a real-time embedded system as a deterministic concurrent system is simplified (synchronous programs are concurrent and deterministic, differently from parallel languages which are based on asynchronous execution schemes where the competition of different processes for the same resource is resolved nondeterministically); synchronous programming is inherently modular; time and functionality are separated (the compiler takes care of processing functional concurrency to derive embedded code, thereby allowing critical applications to be deployed without the need for any operating system scheduler); synchronous languages make formal verification of programs feasible (the synchronous parallel composition greatly reduces the state-explosion problem, compared to the asynchronous interleaving approach to concurrency); and, finally, although synchronous languages have been around for only thirteen years, there is already a substantial amount of industrial offering in terms of tools and design environments.

In summary, synchronous languages are further evidence of the success of the synchronous paradigm. In the words of Benveniste *et. al.*: *"the paradigm of synchrony has emerged as an engineer-friendly design method based on mathematically sound tools"* [15].

1.1.3 The Crisis of the Synchronous Paradigm

If the crisis of a paradigm begins with its blurring, the synchronous paradigm may soon be facing one, paradoxically in application areas where it has been successful so far: integrated circuit design and embedded software development.

In general, while the synchronous hypothesis strongly simplifies system specification, the problem of deriving a correct and efficient physical implementation from it still remains. The difficulty of this problem grows dramatically when the final implementation has a

[4]Synchronous parallel composition may suffer from the *combinational cycle problem*, i.e. the creation of cyclic instantaneous dependency between variables. In hardware design this may occur during the composition of Mealy machines. A discussion of the various methods to handle this issue can be found in [15].

distributed nature that poorly matches the synchronous hypothesis due to large variance in computation and communication times and to the challenge of maintaining a global notion of time.

This is increasingly the case for many important classes of embedded software applications in avionics, automotive electronics, and industrial-plant control where multiple processing elements operating at different clock frequencies are distributed on an extended area and are connected via communication media such as busses (e.g., CAN for automotive applications, ARINC for avionics, and Ethernet for industrial automation) or serial links [10, 14].

And it is also the case for integrated circuit design since the advent of nanometer technologies: as hundreds of millions of transistors can be integrated on a single die, the electronic chip becomes a distributed system with interconnect delays that are not only up to an order of magnitude larger than the switching delays of the logic gates but also extremely difficult to estimate in advance [39].

Hence, the crisis of the synchronous paradigm starts as the consequence of a spreading gap between the *synchronous hypothesis of the specification* and the *distributed reality of the implementation*. On one side, to assume instantaneous communication via broadcasting when it is more likely that the concurrent processes will be eventually implemented as distributed components may lead to poor design specifications. On the other side, even if it is still possible to take a synchronous specification and enforce a synchronous design style on the distributed implementation, the final result may likely be a system that either underperforms or performs wasting too many resources: in both cases a suboptimal design.

But has this crisis really started? After all, the large majority of today's digital chips are still synchronous circuits controlled by a single global clock. And the synchronous paradigm continues to play a fundamental role in the development of embedded systems not only through the adoption of synchronous languages but also with the increasing success of design environments like SIMULINK® & STATEFLOW® [162, 170], which largely benefit from the simplicity of the synchronous hypothesis. Nevertheless it is a fact that the design of high-performance integrated circuits is becoming increasingly more expensive and difficult, and that the demand for more formal methods in the design of distributed embedded systems continues to grow.

It may be debatable whether the crisis of the synchronous paradigm started, but it is clear that it has not ended. The proof is that a new candidate paradigm has not emerged yet. The present dissertation is an attempt to end this apparent crisis with a compromise. Being a compromise, it may eventually develop in either of the following directions: as a confirmation of the synchronous paradigm or as a candidate for a new paradigm.

The thesis presented here is that "*correct-by-construction methods combining the benefits of synchronous specification with the efficiency of asynchronous implementation are the key to design moderately distributed complex systems, i.e. systems composed of tightly interacting distributed concurrent processes.*"

The main motivation behind this thesis is the desire to leverage the traditional tools and practices of synchronous design in the specification and optimization of these systems, while targeting efficient final implementations that are distributed in nature. I argue that the synchronous paradigm is still valid for designing systems whose functional correctness depends on the continuous interaction of several concurrent processes that communicate data more slowly than they process them. These are "moderately distributed" as opposed to "massively distributed", or simply distributed, systems where multiple components operate mostly in an autonomous fashion and their occasional, if any, interaction occurs without stringent time constraints. For instance, a system like the World Wide Web is certainly distributed, but its "correctness" does not depend on the tight interaction of all its components. Actually, the more independently its components can operate, the higher is its "performance" as measured by its offering of prompt and reliable services of various natures.

Instead, the functional correctness and the performance of a one billion transistor system-on-chip or of a drive-by-wire system continuously depends on the correct behavior of each distributed component as well as on their interactions. Consequently, communication and synchronization issues become paramount. Designers of these systems need a sound approach to handle the notion of global state and time, as well as efficient techniques to implement robust distributed communication schemes.

While a solution to the first challenge is found in the synchronous paradigm, it is asynchronous mechanisms such as handshake protocols that will provide answers to the second.

A third challenge is implicit: the need for design methods that can serve as a bridge between these two worlds. These methods must be *correct-by-construction* due to the high complexity of the systems being designed and the critical nature (business-critical, safety-critical) of their applications. Correct-by-construction methods formally guarantee the preservation of essential system properties during any successive refinement step of the design, from the original specification to the final implementation.

The main contribution of this dissertation—Latency-Insensitive Design—includes both a formalization of these ideas (the *theory of latency-insensitive protocols*) and an application to the case of integrated circuit design with nanometer technologies (the *correct-by-construction latency-insensitive design methodology*). In the future, thanks to the generality of its principles, Latency-Insensitive Design can possibly be applied to other research areas, like distributed deployment of embedded software.

1.2 Outline of the Dissertation

This dissertation is organized as follows:

In Chapter 2 I describe the background scenario that motivated my research work and the principles that guided my endeavor. I explain how, as it enters the realm of nanometer process technologies, the semiconductor industry is facing an exacerbation of two problems—productivity gap and timing-closure—which dramatically increase the complexity of designing high-performance integrated circuits. I argue that to manage the complexity of system-on-chip design and to address the challenges of gigascale integration it is necessary to develop new CAD tools in the context of innovative design methodologies that have their foundation on the *principle of orthogonalization of concerns* and the *principle of correct-by-construction design.*

The theory of latency-insensitive protocols is first introduced informally and then presented in formal detail in Chapter 3. Latency-insensitive designs are synchronous, distributed systems that are built by composing functional sequential modules that exchange data on communication channels according to a latency-insensitive protocol. The goal of the protocol is to guarantee that a system composed of functionally correct modules behaves correctly independently of the channel latencies. At the core of the theory lie the notions of *latency equivalence* and *patience*, which are proved to be compositional properties. I also explain the relationship between patience and stallability, and introduce the concepts of *relay station* and *shell* process, which are the building blocks used to transform any synchronous system into a latency-insensitive one. The chapter concludes with a discussion of related work. Special emphasis is put in the comparison of latency-insensitive design with asynchronous design as well as between the theory of latency-insensitive protocol and the theory of desynchronization of synchronous programs.

In Chapter 4 I present a correct-by-construction methodology for latency-insensitive design of systems-on-chip (SOC) with nanometer technologies. The methodology handles the increasing impact of global interconnect delays on integrated circuit design and it facilitates the reuse of intellectual-property (IP) cores for building complex systems-on-chip, thereby reducing the number of costly iterations in the design process. Specifically, the application of latency-insensitive design to integrated circuits presents two main advantages towards the productivity-gap and the timing-closure problems: (1) it facilitates the assembly of pre-designed components (IP cores), that, as long as they are stallable, can be automatically encapsulated within a shell—which interfaces them with the communication protocol—without changing their internal structure; and (2) it enables the *a-posteriori* automatic pipelining of long wires through the insertion of relay stations on the communication channels. I give an operational description of the main building blocks of a latency-insensitive communication architecture (channels, relay stations, and shells) and provide a reference RTL implementation based on the concept of back-pressure. As a case study, I report on the RTL latency-insensitive design of PDLX, a microprocessor with out-of-order and speculative execution.

Performance analysis of latency-insensitive systems is the subject of Chapter 5. I show how latency-insensitive systems can be modeled using *marked graphs*, a subclass of Petri nets, and I provide two constructive models that address different styles of implementation for the building blocks of a latency-insensitive protocol. Both models can be used to compute exactly and efficiently the impact of the insertion of relay stations to pipeline long communication channels between shell processes. Ultimately, the highest processing throughput that can be sustained by a latency-insensitive system depends only on its computation structure. I prove that the combination of the back-pressure mechanism and the introduction of input queues of length two within the shells is sufficient to support the maximum sustainable throughput in a system built under the singular assumption of the stallability of its components.

In Chapter 6 I discuss techniques to optimize the performance of a latency-insensitive system. Building on the modeling results of the previous chapter, I clarify how the *local* insertion of a relay station on any given channel can have a *global* impact on the overall system performance. I then describe simple criteria to optimize channel pipelining based on the analysis of the system topology. To explore alternative design implementations, I define the concept of *recycling*, i.e. the combined application of three design transformations: inserting relay stations, moving them across shell-core pairs, and redrawing the boundaries of the shells around the cores. To optimize the application of recycling and find the right balance between communication and computation latencies, I define the *cycle balancing* problem and present, in the appendix, an algorithm to solve it. This is the basis for developing an interactive design framework for SOCs that is centered on the optimization of the average-case performance of the system, driven by global throughput metrics, as opposed to the worst-case, driven by the minimization of local critical-path timing violations.

Recycling can be seen also as an extension to system-level design of retiming, a classic gate-level optimization techniques. In Chapter 7 I discuss the combination of retiming (a sequential circuit) and recycling (a network of sequential circuits). I provide an analytical model to guide the simultaneous application of these two techniques. This model identifies the conditions under which an optimally retimed synchronous circuit can be further accelerated and estimates the amount of the additional performance gain. Furthermore, I present a simple case study to illustrate how recycling enables high-level design exploration through the reuse of components from a library of pre-designed IP cores.

I conclude with Chapter 8 by summarizing the main contributions of the present work and by outlining the most promising avenues for future research. In particular I suggest that the ideas presented in this dissertation, as well as the general principles behind them, can be used to guide research efforts in other important areas such as: the deployment of synchronous embedded software on heterogeneous distributed architectures and the design of distributed systems that are *globally robust and locally flexible*.

Chapter 2

Background

In which two definitive principles are ultimately given, others will follow.

ELECTRONIC system design is undergoing a revolution that is forcing us to change the traditional engineering practices in order to sustain the outstanding pace of progress of the information technology industry. The advent of nanometer process technologies makes available hundreds of millions of transistors for the design of an entire system on a single chip (system-on-chip). However, designs at this level of sophistication expose issues that were barely visible at previous levels of integration, thus exacerbating both the design productivity gap and the timing closure problem.

In the present chapter, I describe the background of the dissertation and highlight the motivations that prompted this research effort and the principles that guided me throughout my research. The chapter has a threefold structure. In the first two sections I analyze two critical challenges that the designers of electronic systems are facing today: the complexity of system-on-chip design and the impact of latency on nanometer technologies. In the third and final section I argue that the development of design methodologies aimed at assisting designers to address these challenges must be based on the combined application of two principles: the principle of orthogonalization of concerns and the principle of correct-by-construction design.

2.1 The Complexity of System-on-Chip Design

"I tried to illustrate this number, ten to the eighteenth, and I've used raindrops falling on California. E.O. Wilson, a noted Harvard biologist and expert on ants, estimates that the number of ants in the world is between ten to the sixteenth and ten to the seventeenth. So for years I used that. Now each ant has to carry ten to a hundred transistors." These are the words used by INTEL® co-founder, Gordon Moore, during his keynote address at the 50th anniversary of the International Solid-State Circuits Conference while commenting on the estimated number of transistors produced during the year 2002 by the semiconductor industry [174].

Figure 2.1, taken from [173], shows the growth in the estimated number of transistors shipped each year, a growth by the factor of nearly eight orders of magnitude over the last thirty years. This translates into an average compound annual growth of 78%, including several years during the 1970s and 1980s when it exceeded 100%. Fueling this amazing growth is the exponential growth in the number of transistors per integrated circuit, which has been doubling roughly every two years. This phenomenon is now widely known as *Moore's law*, from a 1965 paper [172] in which Moore made the empirical observation that the number of components on semiconductor chips with lowest per-component cost doubles roughly every twelve months, and he predicted that the trend would continue for at least another ten years. In truth, the trend has lasted up to the present and it is now expected to last for at least another decade, thus enabling the integration of more than one billion transistors on a single die before the end of this decade [117]. In the meantime, the semiconductor industry has gone from nothing to become an industry with $200 billion in annual revenue and supporting a trillion-dollar electronics industry [173].

Ironically, however, as it enters the realm of nanometer technology processes [1], this industry is in serious risk of "hurting itself" even before it reaches the physical barrier of atomic dimension. As discussed in the following pages, major challenges such as the complexity of designing a system-on-chip with a billion transistors and the impact of interconnect delay threaten the outstanding pace of technology progress that has shaped the

[1]Nanometer technologies refer to processes at the 90*nm technology node* and below [117]. These are also referred to as deep sub-micron (DSM) technology processes.

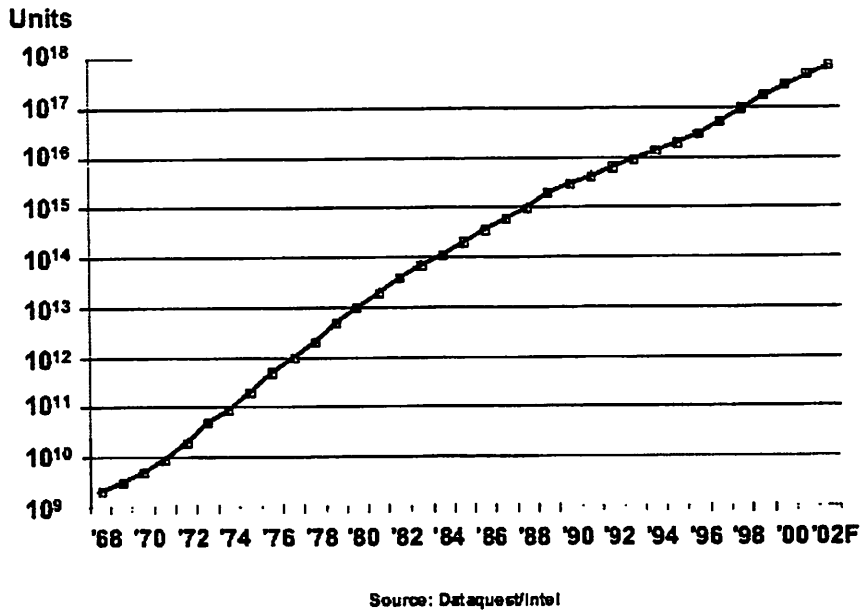

Figure 2.1: Transistors shipped per year [173].

semiconductor industry as no other before.

2.1.1 The Design Productivity Gap

Time-to-market pressure, design complexity and cost of ownership for masks are driving the electronics industry towards more disciplined design styles that favor design re-use and correct-the-first-time implementations. However, traditional computer-aided design (CAD) tool flows are still inadequate to provide support to reach these goals. In [219], Spirakis uses data collected at INTEL® over a period of seven years (1996-2003) to draw the following picture:

- design complexity has grown by a factor of ten: the number of transistors integrated on a single chip have increased from ten million to one hundred million;
- the number of lines of register-transfer level (RTL) specification has grown from 400 thousand to one million;

- the number of pre-silicon design bugs has grown from 500 to eight thousand;
- the team of engineers working on the design validation has doubled in size;
- the slowness of the signal integrity convergence is indicated by the many project months it takes, while more than a thousand RTL lines are modified to enable design progress.

Spirakis' conclusion is clear: *"all of the above clearly indicate that the design tools and methodologies have not progressed in their capacity and efficiency as fast as the design complexity has—and thus have become the bottleneck to the growth of the VLSI market"*.

The phenomenon of CAD tools and methodologies lagging behind the capabilities offered by the continuous progress of Moore's law is commonly referred to as the *design productivity gap*. The 2001 Technology Roadmap for Semiconductors confirmed the gravity of the situation reporting that the available raw transistor count increases by 58% per year, while the designers' capability to design them grows by only 21% per year [210]. Consequently, *"the cost of designing a transistor increases exponentially relative to the cost of the raw transistor"* [123].

The main cause of this problem is that CAD tool flows are built as a juxtaposition of independently conceived stages. Each stage manipulates a representation of the final design, applying certain transformations to it before it goes through the next stage. The exchange of information between stages is based on standard data formats where functional specification, structural information and performance-related annotation are intertwined. These data formats are autonomously interpreted by the tools at each stage. The tools have different purposes (functional manipulation, structural transformation, performance optimization) and, based on their interpretation of the current status of the design, make decisions with global effects that can be difficult to reverse. As a consequence, numerous time-consuming iterations between subsequent stages of the design flow are often necessary. For instance, due to the increasing impact of second-order physical effects in nanometer technologies, designers are forced to iterate many times between HDL specification and layout, since logic synthesis uses statistical delay models that badly estimate the impact of post-layout wire load capacitance and since the netlist lack of structure prevents manual intervention on the layout (see Section 2.2.3).

The general problem is that the design flow does not include formal methods to propagate (and refine) design constraints from the early stages of the process, where the specifications are set, down to the last stages where the final implementation is derived. Similarly, there is a lack of formal methods to uniformly supply the early stages with those technology and performance parameters that are imposed by the adoption of a given process technology and the choice of a specific library of pre-design components. The ultimate consequence is that designers struggle to:

- orient themselves among the various design representations,
- understand the tools' combined behavior, and
- track the evolution of *their* design.

2.1.2 Intellectual Property (IP) Reuse

As the consumer demand for ever more sophisticated electronic products continues to grow, the effective reuse of existing intellectual property design modules (aka IP cores) is commonly perceived as essential to bridge the design productivity gap and allow the completion of a reliable design within the tight constraints of time-to-market. Originally, IP cores were mostly functional blocks built during previous design generations within the same company. Today, more and more frequently IPs are optimized modules marketed as off-the-shelf components by specialized vendors. System-on-chip (SOC) and system-in-package (SIP) designs that incorporate building blocks from multiple sources are supplanting in-house, single-source chip designs [81].

According to the 2003 International Technology Roadmap for Semiconductors (ITRS), reusable, high-level, functional IP blocks offer the potential for productivity gains estimated to be at least of 200% [117]. Hence, it is not surprising that many semiconductor companies share the desire to broaden IP reuse and openly complain about the poor performance of the electronic design automation (EDA) companies in meeting this challenge [189]. In fact, event though design reuse has been set as a requirement in the last several editions of the ITRS, it has yet to permeate the system design process. It is perceived, however, to be a critical requirement for the successful transition from the 90*nm* to the 65*nm* technology [117].

An IP core must be both flexible to collaborate with other modules within different environments, and independent from the particular details of one among many possible implementations. The prerequisite for an easy trade, reuse, and assembly of IP cores is the capability of assembling pre-designed components with little or no effort. The consequent challenge is the ability of addressing the communication and synchronization issues that naturally arise while assembling pre-designed components.

2.2 The Impact of Latency on Nanometer Design

This section looks at the reverse side of the coin, namely how nanometer technologies are forcing the semiconductor industry to experience a paradigm shift from computation- to communication-bound design, where the number of transistors that a signal can reach in a clock cycle—not the number that designers can integrate on a chip—drives the design process. In particular, it is the impact of interconnect delay, which grows with each process generation, that challenges both well-established chip architectures and commonly adopted design methodologies.

2.2.1 From Computation- to Communication-Bound Design

According to the 2001 Technology Roadmap for Semiconductors, the historical half-pitch technology gap between microprocessors/ASIC chips and DRAM will disappear by the year 2005 [4]. As the semiconductor industry proceeds into nanometer technologies, the 2002 130*nm* microprocessor half-pitch will shrink to the projected 32*nm* in 2013, thus allowing the integration of more than one billion transistors on a single die. At the same time, the on-chip local clock frequency is predicted to rise from today's $1-2Ghz$ to $19-20Ghz$. The "interconnect problem", however, threatens to become an impassable roadblock.

Meindl has explained how, during the past decade, interconnects have replaced transistors as the dominant determiner of chip performance by imposing primary limits on

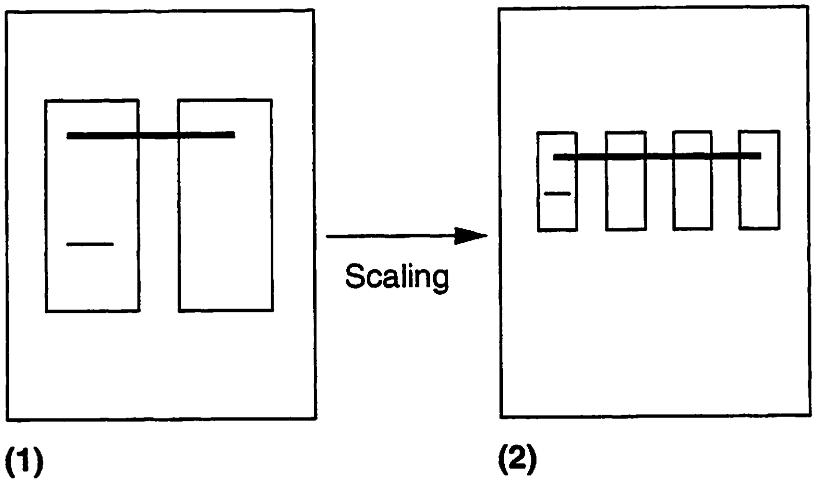

Figure 2.2: Local wires scale in length; global wires do not [112].

latency, energy dissipation, signal integrity and design productivity for gigascale integration (GSI) [167]. Despite the increase in number of metal layers and in aspect ratio, the *resistance-capacitance (RC) delay* of an average metal line with constant length is getting worse with each process generation [20, 74]. The current migration from aluminum to copper metallization is compensating this trend by reducing the interconnect resistivity. The introduction of low-κ dielectric insulators may also alleviate the problem, but these one-time improvements will not suffice in the long run as feature size continues to shrink [19, 87]. The increasing RC delays, combined with the increases in operating frequency, die size, and average interconnect length, cause interconnect delay to become the largest fraction of the clock cycle time [19].

In 1997 COMPUTER magazine published a study by Matzke containing a gloomy forecast of how on-chip interconnect latency (predicted to soon measure in the tens of clock cycles) will hamper Moore's Law [163]. Since then, researchers have been fiercely debating about the real magnitude of the wire delay impact and about the consequent need of revolutionizing established design methodologies already challenged by the timing-closure problem. The core of the debate has been on the scaling properties of interconnect wires relative to gate scaling [183]. On one side, some researchers share an optimistic view based on the fact that wires that scale in length together with gate lengths offer approximately a constant resistance and a falling capacitance. Hence, as long as a modular design approach is adopted and functional modules of up to 50,000 gates are treated as the main components, these researchers' position is that the current design flows are capable of sustaining the challenges of nanometer design [223, 224, 225, 226]. On the other side of the debate, researchers argue that the previous argument does not account for the presence in SOCs of many "global" wires that cannot scale in length because they need to span across multiple modules to connect distant gates [112]. As illustrated in Figure 2.2, which is taken from [112], the impact of technology scaling on devices, local wires and global wires is not the same: a newer technology process enables an higher level of integration as more (and smaller) devices and modules are accommodated on the chip. Similarly, local wires connecting devices within a module shrink nicely with the module. However, global wires connect devices located in different modules do not shrink because they need to span significant portions of the die. In an updated study [113] the same authors report that local and global wires are degrading relative to gates, by one and three orders of magnitude respectively, although they argue that a careful use of simple buffering circuits (a single inverter or two back-to-back inverters) can reduce these degradations to a factor of $40x$ and $2-3x$ (over nine generations) respectively. The key observation behind these numbers is that the increase of wire *aspect ratios* caps at 2.2 while their resistance continues to grow quickly under scaling. The need for extensive buffering is confirmed in another study [206, 207], where projections of historical scaling trends lead to the disturbing prediction that within a few process generations functional modules will have even 70% percent of their cell count dedicated to interconnect buffering.

As Table 2.1 illustrates, the intrinsic interconnect delay of a 1-mm length wire for a 35-nm technology will be longer than the transistor delay by two orders of magnitude [74]. Similar results have been presented in [167]. Matzke's prediction is that with nanometer technologies a signal will need more than ten clock cycles to traverse the entire chip area [163]. Agarwal *et al.* argue that, even under the best conditions, the latency to transmit a signal across the chip in a top-level metal wire will vary between 12 and 32 cycles, depending on the clock rate, while only a fraction of the chip area between 0.4% and 1.4% will be reachable in one clock cycle [3]. In fact, while the number of gates reachable in

Technology	MOSFET switching delay	Min. scaled, $1mm$ wire intrinsic delay	Reverse scaled, $1mm$ wire intrinsic delay	Ratio of wire size to minimum lithographic size
$10\mu m(Al, SiO_2)$	$\sim 20ps$	$\sim 5ps$	$\sim 5ps$	1
$0.1\mu m(Al, SiO_2)$	$\sim 5ps$	$\sim 30ps$	$\sim 5ps$	1.5
$35nm(Cu, low-\kappa)$	$\sim 2.5ps$	$\sim 250ps$	$\sim 5ps$	4.5

Table 2.1: Comparing interconnect and transistor scaling properties [74].

a cycle will not change significantly and the on-chip bandwidth will continue to grow, the percentage of the die reachable within one clock cycle is inexorably and dramatically decreasing: *"we are reaching or have reached a point where more gates can fit on a chip than can communicate in one cycle"* [112]. Hence, instead of being traditionally bound by the number of transistors that can be integrated on a single die (*computation bound*), designs will be bound by the amount of state and logic that can be reached within the required number of clock cycles (*communication bound*). The impact of these trends on system architectures and design methodologies are the subject of the next sections.

2.2.2 The Role of Latency in the Design of the One Billion Transistor Microprocessor

In the case of microprocessors, the evolution towards communication-bound designs implies that the amount of state reachable in a clock cycle, and not the number of transistors, becomes the major factor limiting the growth of instruction throughput (IPC). Furthermore, the increasing interconnect latency will particular penalize current memory-oriented microprocessor architectures that strongly rely on the assumption of low-latency communication with structures such as caches, register files and rename/reorder tables. Recent studies employing cache delay analysis tools (that account for cache parameters as well as technology generation figures) predict that in a $35nm$ design running at $10Ghz$, accessing even a $4KB$ level-one cache will require 3 clock cycles [3]. In fact, this trend is already started, as the architects of the ALPHA 21264 adopted clustered functional units (with a one-cycle penalty for communicating results between them) and a partitioned register file to keep offering high computational bandwidth despite larger wire delays [233]. Similarly, in its hyper-pipelined NETBURST® micro-architecture, the INTEL® PENTIUM®4 microprocessor presents two so-called *drive stages* that are purely dedicated to instruction distribution and data movement [91, 111, 201].

Exposing interconnect latency to the micro-architecture and possibly to the instruction-set architecture (ISA) level will be key to control the system performance. Several research teams have already started investigating this idea [112, 130, 135, 157, 178, 241]. A general trend is the move towards more parallel architecture. Indeed, is likely that the one-billion transistor microprocessor will be a multi-computer. In [69] Dally and Lacy sketch the architecture of a "multi-computer chip" to be built using an hypothetical 2009 CMOS processor-DRAM technology. About 20% of the area of this fine-grain machine is devoted to a number of simple, powerful processors. The multiprocessor is divided in 64 tiles (each containing a processor and a memory), where the round-trip memory-processor intra-tile communication latency is 2 cycles ($1ns$) while the worst case latency for a communication corner-to-corner with the most distant memory is 56 cycles. In particular, Dally and Lacy point out that:

1. on-chip communication bandwidth is not an issue due to the many wiring tracks;

2. unlike modern multiprocessors with their all-or-nothing locality, latency varies continuously with distance;

3. latency is controlled by placing data near their point of use, not just at their point of use.

2.2.3 The Timing-Closure Problem

The traditional design flow for digital integrated circuits design is centered around two independent steps that are performed with different kinds of computer-aided design (CAD) tools [2]:

1. *logic synthesis* [25, 24, 168] tools automatically derive a netlist of standard cells from a functional specification written in a hardware-description language (HDL), such as VERILOG [234] or VHDL [5] (*RTL design*);

2. *place & route* tools [202, 214] automatically produce the final layout by processing the standard-cell netlist (*physical design*).

As technology scaling proceeds in the DSM realm, the effectiveness of this well-established design flow continues to decrease due to the exacerbation of the *timing-closure problem*: the designers of semi-custom integrated circuits are forced to iterate many times between HDL and layout, because (1) the two steps are performed independently, (2) logic synthesis uses statistical delay models that badly estimate the impact of post-layout wire load capacitance, and (3) the netlist's lack of structure prevents manual intervention on the layout [58, 68, 126, 188]. Furthermore, HDLs allow poor control on physical design and the output of logic synthesis is not robust with respect to small variations in the HDL specification. Consequently, it is very hard to develop new CAD tools that are able to make the two steps interact effectively and converge quickly to an optimal solution.

2.2.4 Coping with Volatile Latency

High-end microprocessor designers have traditionally anticipated the challenges that ASIC [51, 216] designers will face with the next process generation. As discussed in Section 2.2.2, latency is increasingly affecting the design of state-of-the-art microprocessors and will be among the main forces shaping the billion-transistor computer architecture [29] expected before the end of this decade. Hence, it is not hard to predict that interconnect latency is destined to have a significant impact also on the design of the communication architecture among the modules of a systems-on-chip. This impact translates into a major repercussion on the effectiveness of the current design methodologies and CAD tools. In fact, it is very difficult to estimate the actual interconnect latency at the early stages of

[2]For an industrial perspective on the current IC design flow, including a more detailed discussion on the various steps and the future challenges for the EDA industry see [46]

the design process because it is affected by several phenomena such as process variations, cross-talk, and power-supply drop variations. Furthermore, their combined effect may vary on different chip regions and across different periods during chip operation. Hence, to find the exact delay value for a global wire may often be impossible and to rely on conservative estimations to establish value intervals may likely lead to sub-optimal designs.

Recently proposed design flows advocating new CAD tools that couple logic synthesis and physical design will suffer from the impracticality of accurately estimating latency of global wires [39]. Indeed, logic synthesis tools are presently suffering from a number of drawbacks:

- *logic synthesis tools are inherently unstable:* small variations on the HDL input specification (that the designers may have to make to fix a slow path) lead to major variations on the output netlist and, consequently, on the final layout;

- *logic synthesis tools are based on the synchronous design methodology:* this methodology is the foundation of the design flows for the majority of commercial chips today, but, if left unchanged, will lead to an exacerbation of the timing closure problem for tomorrow's design flows.

The main assumption in synchronous design is that the delay of each combinational path (i.e. each signal path leaving a latch and traversing only combinational logic and wires to reach another latch) is smaller than the clock period of the system. Hence, the slowest combinational path (*critical path*) dictates the maximum operating frequency for the system. However, it is often the case that the desired operating frequency is fixed as a design constraint. Then, once the final layout is derived, every path having a delay longer than the desired clock period simply represents a *design exception* that needs to be fixed. There are different ways to fix all these exceptions: from wire buffering and transistor re-sizing to re-routing wires, re-placing modules and even re-designing entire portions of the system. Re-placing, re-routing and re-designing clearly do not help alleviating the timing closure problem. Buffering is an efficient technique, but it carries precise limitations because there is a maximum number of buffers that can be inserted on a given wire in order to reduce its delay [7, 192]. Then, as last resort, designers are often forced to break long wires by inserting storage elements (latches, flip-flops, ...) [132, 193], similarly to the insertion of new stages in a pipeline. This operation trades off fixing a wire exception with increasing its latency by one or more clock cycles and will become pervasive in nanometer designs, where most global wires will be heavily pipelined to begin with (*wire pipelining*). Again, confirmation of this fact can be found in some recent high-performance microprocessor designs:

- in [164], McInerney *et al.* report that about 85% of the 13,000 top-level nets of the INTEL® ITANIUM® microprocessor require one or more repeaters of some type to meet the frequency goal for the chip tape out and that many of the nets *"require solutions that involve clocked and enabled elements such as latches and flip-flops"*;
- in [220], Sprangle and Carmean report a personal communication with Sager on wire pipelining saying that *"there are plenty of places in the* INTEL® PENTIUM®4 *where the wires were pipelined"*.

Latency, measured in clock cycles, between the various components of a SOC varies considerably based on their reciprocal distances, even without considering the need of increasing it to further pipeline long global wires. Inserting storage elements (*stateful* repeaters) has generally a different impact on the surrounding control logic in comparison to inserting transistor buffers (*stateless* repeater). If the interface logic of two communication components has been designed assuming a certain latency, then it needs to be redesigned to account for additional pipeline stages, with serious consequences on design productivity. In [209], Scheffer list a series of important issues raised by the insertion of stateful repeaters:

- inserting, moving and removing stateful repeaters are delicate actions that engineers must perform manually at the layout level; further they have to track these changes at RTL level (by modifying the HDL code) to be able to continue to validate the design;
- each insertion/removal of stateful repeaters invalidates the existing simulation vectors (and expected results);

- proving that the design remains correct after performing interconnect pipelining is difficult also because tools for functional equivalence checking cannot be applied across this design transformation;
- designing RTL and deriving the layout are no longer independent stages of the design flow.

According to Saxena *et al.*, the scenario can only get worse in the future as *"repeaters, which are already a problem at the full-chip level, will become critical at the block level also"* [207]. Still, they point out that *mispredicted cycle latency* of global interconnects is more damaging than such mispredictions within blocks due to the major impact on the overall chip assembly. Although improved estimation techniques may be developed in the future, the designers of the microarchitecture will have to deal with multi-cycle estimation errors for global communication paths. They will certainly dedicate the best resources (top metal layers, repeaters) to the most critical paths, but routing constraints will prevent them from guaranteeing a certain cycle-latency for all the global paths. Therefore, there is a need to develop new design methodologies that (1) guarantee the functional correctness of the micro-architecture design while supporting *cycle latency ranges* for interdependent sets of global interconnects and (2) help analyze the sensitivity of its performance to variations within these ranges. In the words of Saxena *et al.* *"this will allow intelligent late stage* correct-by-construction *cycle-latency optimizations for improved performance without invalidating earlier microarchitectural performance simulations or machine correctness, thus avoiding downstream surprises because of infeasible cycle latency constraints"* [207].

2.3 The Combination of Two Design Principles

To manage the complexity of system-level design and to address the challenges of gigascale integration it is necessary to develop new CAD tools in the context of innovative design methodologies. The principle of orthogonalization of concerns and the principle of correct-by-construction design will guide us in this process.

The *principle of orthogonalization of concerns* advocates the separation/decomposition of the various aspects of design along orthogonal axes to allow a more effective exploration of alternative solutions. This can be naturally applied at different stages of the design process as well as at different levels of granularity. Examples of orthogonal axes are:

- *specification vs. implementation*: the definition of the design specification versus the derivation of a particular implementation (among the many possible ones);
- *computation vs. communication*: the specification of the computational features of the design components versus the specification of the communication architecture among the components;
- *functionality vs. performance*: the design and validation of the functional behavior of a possible implementation versus the analysis and tuning of its performance metrics.

The *principle of correct-by-construction design* states that a complex system is built through a (possibly hierarchical) sequence of precise steps during which simpler components are assembled to derive more complex ones whose key properties are guaranteed to be correct by the very nature of the assembling step. The process of deriving the final implementation from the original specification becomes a sequence of *refinement steps* from higher levels of abstraction to more and more detailed ones. The design representation grows at each level of the design process as alternative implementation solutions are considered and decisions are made. The difference from common top-down design flows is that each refinement step is performed according to formal guidelines that guarantee the preservation of key properties of the design, while designers are free to explore alternative solutions to optimize it under other criteria. Naturally, to continuously preserve certain key properties may restrict the searchable solution space. But this is precisely the point as correct-by-construction design means *trading off optimality for robustness* for those variables that have a dimension that is impractical, if not impossible, to explore.

Observe that the formalization of the refinement steps is made feasible by the possibility of focusing on each design property separately. The importance of combining the application of the two principles naturally follows. A testimony for the effectiveness of this approach is the theory of latency-insensitive protocols, which is the subject of Chapter 3.

Chapter 3

Theory of Latency-Insensitive Protocols

In which patience, once again, manifests itself as the most precious virtue.

THE theory of latency-insensitive protocols represents the foundation of a correct-by-construction methodology to design complex systems by assembling pre-designed components. Latency-insensitive designs are synchronous, distributed systems and are built by composing functional modules that exchange data on communication channels according to a latency-insensitive protocol. The protocol works on the assumption that the modules are stallable, a weak condition to ask them to obey. The goal of the protocol is to guarantee that a system composed of functionally correct modules behaves correctly independently of the channel latencies. This allows increasing the robustness of a design implementation, because any delay variations of a channel can be "recovered" by changing the channel latency while the overall system functionality remains unaffected. As a consequence, an important application of the proposed theory is the latency-insensitive design methodology to build systems-on-chip with nanometer technologies. This application is the subject of Chapter 4.

I presented the first results of the theory of latency-insensitive protocols at the 11th International Conference on Computer-Aided Verification in July 1999 [36]. A more complete exposition of the theory was published on the September 2001 issue of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems [37].

3.1 An Informal Presentation

The principle of correct-by-construction design and the principle of orthogonalization of concerns are the two pillars of the theory of latency-insensitive protocols. Section 2.3 anticipates the critical importance of applying the two principles in a combined effort. The theoretical results presented in this chapter are a confirmation of this point.

Latency-insensitive protocols are a mechanism to *formally separate* communication from computation by specifying a system as a collection of computational processes that exchange data by means of communication channels. The communication is governed by an abstract protocol, whose main characteristic is to be insensitive to the latencies of the channels. The theory may be applied as a rigorous basis to design complex systems by simply composing pre-designed and verified components so that the composition satisfies, *formally and by construction*, the required properties of synchronization and communication. Also, the theory naturally enables the orthogonalization of the system specification from the derivation of one among many possible implementations. The designers of a latency-insensitive system can focus first on specifying the overall system and then on choosing the best components for the implementation. While doing so they do not need to worry about communication details such as data synchronization and transmission latency. The latency-insensitive protocol takes care of these. Further, as the designers explore the design space and consider alternative implementations for the various parts of the system, they can rely on the fact that the specific implementation of the latency-insensitive protocol is automatically generated.

The following sections contain a formal presentation of the theory of latency-insensitive protocols and the concept of latency-insensitive design. Here, I give an informal summary of the content of this chapter.

Section 3.2 contains the definitions of latency equivalence and patient processes that lie at the core of the theory. A latency-insensitive protocol controls the communication among the components of a *patient system*, i.e. a system of patient processes. Two systems are latency equivalent if on every channel they present the same data streams, i.e. the same ordered sequence of data, but, possibly, with different timing. A synchronous system can be modeled as a set of processes communicating by exchanging signals on a set of point-to-point channels. A patient system is a synchronous system whose functionality depends only on the order of the events of each signal and not on their exact timing. A latency-insensitive protocol guarantees that a patient system, if composed of functionally correct modules, behaves correctly independently from the delays of the channels connecting the modules.

In Section 3.3 I show that the notions of patience and latency equivalence are compositional by proving the following theorems:

1. the intersection of two patient processes is a patient process (Theorem 3.2);
2. given two pairs of latency-equivalent patient processes, their pairwise intersections are also latency equivalent (Theorem 3.3);
3. for all pairs of strict processes P_1, P_2 and patient processes Q_1, Q_2, if P_1 is latency equivalent to Q_1 and P_2 is latency equivalent to Q_2 then their pairwise intersections are latency equivalent (Theorem 3.4).

As a consequence, I derive the major result of the theory: *if all processes in a strict system are replaced by corresponding latency-equivalent patient processes then the resulting system is patient and latency equivalent to the original one.* Then, I define the notion of *relay station*, illustrate its main properties, and I show how the latencies of the communication channels in a system of patient processes can be adapted through the insertion of relay stations. A relay station, being a buffer process of unit latency and twofold storage capacity, represents the optimal building block for the construction of patient channels. Patient channels are the abstraction to model communication media between patient processes. In particular, they can be used to formalize the design practice of wire pipelining whose increasing importance is discussed in Section 2.2.4.

In Section 3.4 I discuss first the condition under which a generic strict system can be transformed into a patient one, i.e. its components must be *stallable*. Stallability is instrumental in order to enable latency-insensitive design because to require the direct design of patient processes would be too demanding from a practical viewpoint, while stallability is a weaker condition to ask from functional processes. I also prove that every stallable process can be encapsulated into a so-called *shell process*, which acts as an interface towards a latency-insensitive protocol. Then, I delineate how the present theory leads to the notion of *latency-insensitive design methodology*, which provides a formal way to *orthogonalize* computation and communication. In fact, I can build systems by assembling functional *core processes* (which can be arbitrarily complex as far as they satisfy the stalling assumption) and shell processes (which interface the cores with the channels by "speaking" the latency-insensitive protocol), while knowing that the latency of the communication among them may vary arbitrarily without affecting the functionality of the system. In other words, regardless of the number and the complexity of the components, the functionality of the system is guaranteed in a *correct-by-construction* fashion. As a specific application, I outline the structure of a latency-insensitive design methodology for SOC design, which is the subject of Chapter 4.

Finally, in Section 3.5 I comment on the relationships between the theory of latency-insensitive protocols and the work of other researchers in the fields of asynchronous design, high-level synthesis and theory of desynchronization. The relationships between latency-insensitive design and retiming are left as the subject of Chapter 7.

3.2 Latency Insensitivity

To develop the theory formally I adopt the *tagged-signal model*, a denotational framework that was proposed by Lee and Sangiovanni-Vincentelli to represent complex systems as collections of signals and processes [144].

3.2.1 The Tagged-Signal Model

Given a set of *values* $\mathcal{V}$ and a set of *tags* $\mathcal{T}$, an *event* is a member of $\mathcal{V} \times \mathcal{T}$. A *signal* s is a set of events. The set of all N-tuples of signals is denoted $\mathcal{S}^N$. A *process* P is a subset of $\mathcal{S}^N$. A particular N-tuple $\mathbf{s} \in \mathcal{S}^N$ satisfies the process if $\mathbf{s} \in P$. A N-tuple $\mathbf{s}$ that satisfies a process is called a *behavior* of the process. Thus, a process is a set of possible behaviors [1]. A *composition of processes* (also called a *system*) $\{P_1, \dots, P_M\}$, is a new process defined as the intersection of their behaviors $P = \bigcap_{m=1}^{M} P_m$. Since processes can be defined over

[1]For $N \geq 2$, processes may also be viewed as a relation between the N signals in $\mathbf{s} = (s_1, \dots, s_N)$.

different sets of signals, to form the composition it is necessary to extend the set of signals over which each process is defined to contain all the signals of all processes. Note that the extension changes the behavior of the processes only formally.

Let $J = (j_1, \ldots, j_h)$ be an ordered set of integers in the range $[1,N]$. The *projection* of a behavior $b = (s_1, \ldots, s_N) \in \mathcal{S}^N$ onto $\mathcal{S}^h$ is $proj_J(b) = (s_{j_1}, \ldots, s_{j_h})$. The projection of a process $P \subseteq \mathcal{S}^N$ onto $\mathcal{S}^h$ is $proj_J(P) = (\mathbf{s}' \mid \exists \mathbf{s} \in P \wedge proj_J(\mathbf{s}) = \mathbf{s}')$. A *connection* C is a particularly simple process where two (or more) of the signals in the N-tuple are constrained to be identical: for instance, $C(i,j,k) \subset \mathcal{S}^N \; : \; (s_1, \ldots, s_N) \in C(i,j,k) \Leftrightarrow s_i = s_j = s_k$, with $i,j,k \in [1,N]$.

An *input* to a process $P \subseteq \mathcal{S}^N$ is an externally imposed constraint $P_I \subseteq \mathcal{S}^N$ such that $P_I \cap P$ is the total set of acceptable behaviors. The set of all possible inputs $I \subseteq \mathcal{S}^n$ is a further characterization of a process. Given a process P, if $I = \{\mathcal{S}^N\}$ then the set of acceptable behaviors is $\{\mathcal{S}^N\} \cap P = P$ and the process does not have input constraints (the process is *closed*). Commonly, one considers processes having input signals and output signals. In this case, given process P, the set of signals can be partitioned into three disjoint subsets by partitioning the index set as $\{1, \ldots, N\} = I \cup O \cup R$, where I is the ordered set of indexes for the input signals of P, O is the ordered set of indexes for the output signals and R is the ordered set of indexes for the remaining signals (also called irrelevant signals with respect to P). A process is *functional* with respect to (I,O) if for all behaviors $b \in P$ and $b' \in P$, $proj_I(b) = proj_I(b')$ implies $proj_O(b) = proj_O(b')$. Hence, given a function $F : \mathcal{S}^{|I|} \rightarrow \mathcal{S}^{|O|}$, a functional process P is completely characterized by the tuple (F,I,O). A process P is *determinate* if for any input $I' \in I$ then either $|I' \cap P| = 1$ or $|I' \cap P| = 0$. Otherwise, it is *non-determinate*.

Two events e_1, e_2 are *synchronous* ($e_1 \approx e_2$) when they have the same tag, i.e. $e_1 \approx e_2 \Leftrightarrow tag(e_1) = tag(e_2)$. Two signals s_1, s_2 are synchronous ($s_1 \approx s_2$) when for each event of s_1 there is a synchronous event in s_2 and vice versa, i.e.:

$$s_1 \approx s_2 \Leftrightarrow \Big(\forall e_i \in s_1, \exists e_j \in s_2, \; e_i \approx e_j \Big) \wedge \Big(\forall e_k \in s_2, \exists e_l \in s_1, \; e_k \approx e_l \Big)$$

Therefore, synchronous signals share the tag set. The definitions of two synchronous be-

	tag	:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
	w	:	1	0	1	0	1	0	1	0	...
P:	y	:	0	2	2	6	6	10	10	14	...
	z	:	0	0	4	0	8	4	12	8	...
	w	:	1	0	1	0	1	0	1	0	...
Q:	x	:	1	3	5	7	9	11	13	15	...
	y	:	0	2	2	6	6	10	10	14	...
	w	:	1	0	1	0	1	0	1	0	...
R:	x	:	1	3	5	7	9	11	13	15	...
	z	:	0	0	4	0	8	4	12	8	...

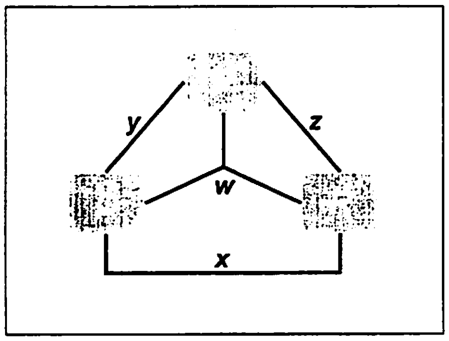

Figure 3.1: The synchronous system of Example 3.2.1 and its behavior.

haviors b_1, b_2 and two synchronous processes P_1, P_2 naturally follow:

$$\begin{aligned} b_1 \approx b_2 &\Leftrightarrow \forall s_i \in b_1, \forall s_j \in b_2, s_i \neq s_j, \ (s_i \approx s_j) \\ P_1 \approx P_2 &\Leftrightarrow \forall b_i \in P_1, \forall b_j \in P_2, \ (b_i \approx b_j) \end{aligned}$$

A stand-alone behavior b is synchronous when $b \approx b$. A stand-alone process P is synchronous when $P \approx P$. In any behavior of a *synchronous system*, every signal is synchronous with every other signal and, equivalently, for each tag a signal has *exactly one* corresponding event: $\forall s \in b, \forall t \in \mathcal{T}, \left(\exists! e \in s \ \ tag(e) = t\right)$.

Example The diagram of Figure 3.1 represents the unique behavior of a synchronous system that is the result of the composition of three processes P, Q, and R. Signal w, a binary, is shared by all processes, while the remaining signals, integers x,y, and z, are shared in pairwise manner. In Figure 3.1, the signals are purposely represented by simple lines and not arrows. In fact, by observing only the event sequences we can not say which input/output relations exist among the system processes. ■

The definition of asynchrony as used in the literature is vague: some use the term to indicate any systems that is *not* synchronous, others are more restrictive. In the tagged-signal mode, two events e_1, e_2 are *asynchronous* ($e_1 \simeq e_2$) if they have different tags, i.e. $e_1 \simeq e_2 \Leftrightarrow tag(e_1) \neq tag(e_2)$. Two signals s_1, s_2 are asynchronous ($s_1 \simeq s_2$) when:

$$s_1 \simeq s_2 \Leftrightarrow \left(\forall e_i \in s_1 \ \nexists e_j \in s_2 \ \ e_i \approx e_j\right)$$

	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	...
	w_a :	1				0				1				0				1			...
P_a :	y_a:			0				2				2				6				6	...
	z_a :				0				0				4				0				...
	w_a :	1				0				1				0				1			...
Q_a :	x_a :		1				3				5				7				9		...
	y_a :			0				2				2				6				6	...
	w_a :	1				0				1				0				1			...
R_a :	x_a :		1				3				5				7				9		...
	z_a :				0				0				4				0				...

Asynchronous signals have disjoint tag sets. The definitions of asynchronous behaviors b_1, b_2 and asynchronous processes P_1, P_2 follow:

$$b_1 \simeq b_2 \quad \Leftrightarrow \quad \forall s_i \in b_1, \forall s_j \in b_2,\ s_i \simeq s_j$$

$$P_1 \simeq P_2 \quad \Leftrightarrow \quad \forall b_i \in P_1, \forall b_j \in P_2,\ b_i \simeq b_j$$

A stand-alone behavior b is asynchronous when $b \simeq b$. A stand-alone process P is asynchronous when $P \simeq P$. In a behavior of an asynchronous system, every signal is asynchronous with every other signal and, equivalently, for each tag there is one and only one event across all signals: $\forall t \in, \Big((\exists ! e \in \bigcup_i s_i tag(e) = t \Big)$.

Example The following diagram represents the unique behavior of the asynchronous system $S_a = P_a \cap Q_a \cap R_a$. Processes P_a, Q_a, and R_a communicate by sharing signals (as it is the case for synchronous systems), but signals do not share tags. ■

In a *timed system* the set $\mathcal{T}$ of tags, also called *timestamps*, is a totally ordered set. The ordering among the timestamps of a signal s induces a natural order on the set of events of s. A functional process is *(strictly) causal* if two outputs can only differ at timestamps that (strictly) follow the timestamps when the inputs producing these outputs show a difference. More formally, if d is a metric on the set $\mathcal{S}^N$ of N-tuples of signals [2], then a functional process $P = (F, I, O)$ is *causal* when

$$\forall s_i, s_j \in \mathcal{S}^{|I|} \ \Big(\ d(F(s_i), F(s_j)) \leq d(s_i, s_j) \ \Big)$$

[2]For instance, in [144] it is considered the Cantor metric $d(s_i, s_j) = \sup\{\frac{1}{2^t} \mid s_i(t) \neq s_j(t), t \in \mathcal{T}\}$.

A functional process $P = (F, I, O)$ is *strictly causal* when

$$\forall s_i, s_j \in S^{|I|} \left(d(F(s_i), F(s_j)) < d(s_i, s_j) \right)$$

3.2.2 Informative Events and Stalling Events

A *latency-insensitive system* is a synchronous timed system whose set of values $\mathcal{V}$ is equal to $\Sigma \cup \{\tau\}$, where Σ is the set of *informative symbols* which are exchanged among modules and $\tau \notin \Sigma$ is a special symbol, representing the absence of an informative symbol. The absence of an informative symbol may result from either lack of valid data to transmit or *back-pressure*, i.e. a request to delay a transmission coming back from a down-link process. From now on, all signals are assumed to be synchronous. The set of timestamps is assumed to be in one-to-one correspondence with the set $\mathbb{N}$ of natural numbers. An event is called *informative* if it has an informative symbol ι_i as value [3]. An event whose value is a τ symbol is said to be a *stalling event* (or τ *event*) [4].

Definition 3.1 *$\mathcal{E}(s)$ denotes the set of events of signal s while $\mathcal{E}_\iota(s)$ and $\mathcal{E}_\tau(s)$ are respectively the set of informative events and the set of stalling events of s. The k-th event (v_k, t_k) of a signal s is denoted $e_k(s)$. $\mathcal{T}(s)$ denotes the set of timestamps in signal s, while $\mathcal{T}_\iota(s)$ is the set of timestamps corresponding to informative events.*

Processes exchange "useful" data by sending and receiving informative events. Ideally only informative events should be communicated among processes. However, in a latency-insensitive system, a process may not have data to output at a given timestamp, thus requiring the output of a stalling event at that timestamp. Alternatively, it may happen that a down-link process that is not ready to receive new data requests the up-link process to avoid sending them and, as a consequences, the latter reacts by emitting a stalling event (*back-pressure*).

[3] I use subscripts to distinguish among the different informative symbols of Σ : $\iota_1, \iota_2, \iota_3, \ldots$

[4] The role of the τ event is similar to the one played by the *absence* symbol $\perp$ in the synchronous language SIGNAL [16]. See also Section 3.5.5

Definition 3.2 *The set of all sequences of elements in $\Sigma \cup \{\tau\}$ is denoted by Σ_{lat}. The length of a sequence σ is $|\sigma|$ if it is finite, otherwise it is infinity. The empty sequence is denoted as ε and, by definition, $|\varepsilon| = 0$. The i-th term of a sequence σ is denoted σ_i.*

Definition 3.3 *Function $\sigma : \mathcal{S} \times \mathcal{T}^2 \to \Sigma_{lat}$ takes a signal $s = \{(v_0, t_0), (v_1, t_1), ..\}$ and an ordered pair of timestamps (t_i, t_j), $i \leq j$, and returns a sequence $\sigma_{[t_i,t_j]} \in \Sigma_{lat}$ such that (s.t.) $\sigma_{[t_i,t_j]}(s) = v_i, v_{i+1}, \ldots, v_j$ [5]. The sequence of values of a signal is denoted $\sigma(s)$. The infinite subsequence of values corresponding to an infinite sequence of events, starting from t_i is denoted $\sigma_{[t_i,\infty]}(s)$.*

Example If $s = \{(\iota_1, t_1), (\iota_2, t_2), (\tau, t_3), (\iota_2, t_4), (\iota_1, t_5), (\tau, t_6)\}$ then [6]:

$$\begin{aligned} \sigma(s) &= \iota_1\ \iota_2\ \tau\ \iota_2\ \iota_1\ \tau \\ \sigma_{[t_2,t_4]}(s) &= \iota_2\ \tau\ \iota_2 \\ \sigma_{[t_5,t_5]}(s) &= \iota_1 \end{aligned}$$

and respectively, $|\sigma(s)| = 6$, $|\sigma_{t_2,t_4}(s)| = 3$, $|\sigma_{t_5,t_5}(s)| = 1$. ■

The following filtering operators are defined to manipulate sequences of values.

Definition 3.4 *$\mathcal{F}_\iota : \Sigma_{lat} \to \Sigma^\star$ returns a sequence $\sigma' = \mathcal{F}_\iota[\sigma]$ s.t.*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s) & \textit{if } \sigma_{[t_i,t_i]}(s) \in \Sigma \\ \varepsilon & \textit{otherwise} \end{cases}$$

Definition 3.5 *$\mathcal{F}_\tau : \Sigma_{lat} \to \{\tau\}^\star$ returns a sequence $\sigma' = \mathcal{F}_\tau[\sigma]$ s.t.*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s) & \textit{if } \sigma_{[t_i,t_i]}(s) = \tau \\ \varepsilon & \textit{otherwise} \end{cases}$$

Example If $\sigma(s) = \iota_1\ \iota_2\ \tau\ \iota_2\ \iota_1\ \tau$, then $\mathcal{F}_\iota[\sigma(s)] = \iota_1\ \iota_2\ \iota_2\ \iota_1$ and $\mathcal{F}_\tau[\sigma(s)] = \tau\ \tau$. ■

Obviously, $|\sigma(s)| = |\mathcal{F}_\iota[\sigma(s)]| + |\mathcal{F}_\tau[\sigma(s)]|$. Latency-insensitive systems are assumed to have a finite horizon over which informative events appear, i.e., for each signal s there is a *greatest timestamp* $T \in \mathcal{T}_\iota(s)$ which corresponds to the "last" informative event. However,

[5] Notice that $\sigma_{[t_i,t_i]}(s)$ denotes the value of the event at t_i.

[6] In this chapter it is assumed: $\forall t_i \in \mathcal{T}(s), \forall t_j \in \mathcal{T}(s)$, $(t_i \leq t_j \Leftrightarrow i \leq j)$.

$$\boxed{\begin{array}{lcl}\sigma(s_1) & = & \iota_1\ \iota_2\ \iota_3\ \iota_1\ \iota_4\ \iota_1\ \iota_2\ \iota_2\ \iota_1\ \iota_3\,\tau\ \tau\ \tau\ \tau\ \tau\ \ldots \\ \sigma(s_2) & = & \iota_1\,\tau\ \iota_2\,\tau\ \iota_2\,\tau\ \iota_3\,\tau\ \tau\ \iota_4\,\tau\ \iota_5\,\tau\ \iota_6\,\tau\ \iota_3\,\tau\ \tau\ \iota_1\,\tau\ \iota_4\,\tau\ \tau\ \tau\ \tau\ \tau\ldots\end{array}}$$

Figure 3.2: A strict signal s_1 and a stalled signal s_2.

to build the present theory, the set of signals of a latency-insensitive system is extended over an infinite horizon by adding a set of timestamps such that all events with timestamp greater than T have τ values.

Definition 3.6 *A signal s is* strict *if and only if (*iff*) all informative events precede all stalling events, i.e.,* iff *there exists a $k \in \mathbb{N}$ s.t.* $|\mathcal{F}_\tau[\sigma_{[t_0,t_k]}(s)]| = 0$ *and* $|\mathcal{F}_\iota[\sigma_{[t_k,t_\infty]}(s)]| = 0$. *A signal which is not strict is said to be* delayed *(or* stalled*).*

Example Figure 3.2 illustrates the sequences associated to two signals presenting 10 informative events each: s_1 is a strict signal with greatest timestamp equal to 10, while s_2 is a stalled signal with greatest timestamp equal to 21. ■

3.2.3 Latency Equivalence

Two signals are latency equivalent if they present the same sequence of informative events, i.e., they are identical except for different delays between two successive informative events. Formally:

Definition 3.7 *Two signals s_1, s_2 are latency equivalent* $s_1 \equiv_\tau s_2$ iff $\mathcal{F}_\iota[\sigma(s_1)] = \mathcal{F}_\iota[\sigma(s_2)]$.

The *reference signal s_{ref}* of a class of latency-equivalent signals is a strict signal obtained by assigning the sequence of informative values that characterizes the equivalence class to the first $|\mathcal{F}_\iota[\sigma(s_1)]|$ timestamps.

Example Figure 3.3 reports the sequences associated to three signals that belong to the same latency-equivalent class: signal s_1 is also the reference signal of the class. ■

Latency-equivalent signals contain the same sequences of informative values, but with different timestamps. Hence, it is useful to identify their informative events with respect to the common reference signal: the *ordinal* of an informative event coincides with its position in the reference signal.

$$\boxed{\begin{array}{rcl}
\sigma(s_1) & = & \iota_1\ \iota_2\ \iota_1\ \iota_2\ \iota_3\ \iota_1\ \iota_2\ \tau\ \tau\ \tau\ \ldots \\
\sigma(s_2) & = & \iota_1\ \iota_2\ \tau\ \tau\ \iota_1\ \tau\ \iota_2\ \iota_3\ \tau\ \iota_1\ \tau\ \iota_2\ \tau\ldots \\
\sigma(s_3) & = & \iota_1\ \iota_2\ \tau\ \iota_1\ \iota_2\ \iota_3\ \tau\ \iota_1\ \iota_2\ \tau\ \tau\ \tau\ \ldots
\end{array}}$$

Figure 3.3: Sequences of values of three latency-equivalent signals.

Definition 3.8 *The* ordinal *of an informative event $e_k = (v_k, t_k) \in \mathcal{E}_\iota(s)$ is defined as*

$$ord(e_k) = |\mathcal{F}_\iota[\sigma_{|t_0,t_k]}](s)| - 1$$

Let s_1 and s_2 be two latency-equivalent signals: two informative events $e_k(s_1) \in \mathcal{E}_\iota(s_1)$ and $e_l(s_2) \in \mathcal{E}_\iota(s_2)$ are said to be corresponding events iff *$ord(e_k(s_1)) = ord(e_l(s_2))$. The* slack *between two corresponding events is defined as $slack(e_k(s_1), e_l(s_2)) = |k - l|$.*

Hence, if s is strict the ordinal of an informative event coincides with its position on $\sigma(s)$. Observe that if s_1 and s_2 are latency-equivalent signals, then corresponding informative events in s_1 and s_2 have the same ordinals (while they may have different timestamps).

The notion of latency equivalence is extended to behaviors in a component-wise manner:

Definition 3.9 *Two behaviors $(s_1, \ldots, s_N)$ and $(s'_1, \ldots, s'_N)$ are latency equivalent* iff $\forall i$ *$(s_i \equiv_\tau s'_i)$. A behavior $b = (s_1, \ldots, s_N)$ is* strict iff *every signal $s_i \in b$ is strict. Every class of latency-equivalent behaviors contains only one strict behavior: this is called the* reference behavior.

Definition 3.10 *Two processes P_1 and P_2 are latency equivalent, $P_1 \equiv_\tau P_2$, if every behavior of one is latency equivalent to some behavior of the other. A process P is strict* iff *every behavior $b \in P$ is strict. Every class of latency-equivalent processes contains only one strict process: the* reference process.

Definition 3.11 *A signal s_1 is latency dominated by another signal s_2, $s_1 \leq_\tau s_2$,* iff *$s_1 \equiv_\tau s_2$ and $T_1 \leq T_2$, with $T_k = \max \{t \mid t \in \mathcal{T}_\iota(s_k)\}, k = 1, 2$.*

Example In the example of Figure 3.3, signal s_3 is dominated by signal s_2 since $T_3 = 9$ while $T_2 = 12$.

Notice that a reference signal is latency dominated by every signal belonging to its equivalence class. Latency dominance is extended to behaviors and processes as in the case of latency equivalence.

3.2.4 Ordering the Set of Informative Events

To develop the present theory it is necessary to define a total order among the events of a behavior. In particular, I introduce an ordering among events that is motivated by causality: events that have a smaller ordinal are ordered before the ones with larger ordinals [7]. In addition, to avoid combinational cycles that may be created by processing events with the same ordinal, I rely on a well-founded order over the set of signals. This order in real-life designs corresponds to input-output combinational dependencies as they can be found, for instance, in the implementation of Mealy finite state machine. The following definition casts this consideration in the most general form possible to extend maximally the applicability of the theory.

Definition 3.12 *Given a behavior $b = (s_1, \dots, s_N)$, symbol $\leq_c$ denotes a well-founded order on its set of signals. The well-founded order induces a* lexicographic order *$\leq_{lo}$ over the set of informative events of b, s.t. for all pairs of events (e_1, e_2) with $e_1 \in \mathcal{E}_\iota(s_i)$ and $e_2 \in \mathcal{E}_\iota(s_j)$:*

$$e_1 \leq_{lo} e_2 \quad \Leftrightarrow \quad \Big[\big(ord(e_1) < ord(e_2)\big) \vee \Big(\big(ord(e_1) = ord(e_2)\big) \wedge (s_i \leq_c s_j) \Big) \Big]$$

The following function returns the first informative event (in signal s_j of behavior b) following an event $e \in b$ with respect to the lexicographic order $\leq_{lo}$.

Definition 3.13 *Given a behavior $b = (s_1, \dots, s_N)$ and an informative event $e(s_i) \in \mathcal{E}_\iota(s_i)$, the function* nextEvent *is defined as:*

$$nextEvent\big(s_j, e(s_i)\big) = \min_{e_k(s_j) \in \mathcal{E}_\iota(s_j)} \Big\{ e(s_i) \leq_{lo} e_k(s_j) \Big\}$$

[7] In a strict process the ordinal is related to the timestamp; it implies that past events do not depend on future events.

A *stall move* postpones an informative event of a signal of a given behavior by one timestamp. The stall move is used to account for long delays along communication channels (i.e. wires on the chip) and to add delays where needed to guarantee functional correctness of the design.

Definition 3.14 *Given a behavior* $b = (s_1, \ldots, s_j, \ldots, s_N)$ *and an informative event* $e_k(s_j) = (v_k, t_k)$*, a* stall move *returns a behavior* $b' = stall\big(e_k(s_j), b\big) = (s_1, \ldots, s'_j, \ldots, s_N)$*, s.t. for all* $l \in \mathbb{N}$*:*

$$\begin{aligned}
\sigma_{[t_0,t_{k-1}]}(s'_j) &= \sigma_{[t_0,t_{k-1}]}(s_j), \\
\sigma_{[t_k,t_k]}(s'_j) &= \tau, \\
\sigma_{[t_{k+l+1},t_{k+l+1}]}(s'_j) &= \sigma_{[t_{k+l},t_{k+l}]}(s_j).
\end{aligned}$$

A *procrastination effect* represents the "effect" of a stall move $stall\big(e_k(s_j), b\big)$ on other signals of behavior b in correspondence of events following $e_k(s_j)$ in the lexicographic order. The processes will "respond" to the insertion of stalls in some of their signals by "delaying" other signals that are causally related to the stalled signals. Given a behavior b for each stall move on events of b there is a corresponding set of behaviors (the *procrastination effect set*).

Definition 3.15 *A* procrastination effect *is a point-to-set map which takes a behavior* $b' = (s'_1, \ldots, s'_N) = stall\big(e_k(s_j), b\big)$ *resulting from the application of a stall move on event* $e_k(s_j)$ *of behavior* $b = (s_1, \ldots, s_N)$ *and returns a set of behaviors* $\mathcal{PE}\Big[stall\big(e_k(s_j), b\big)\Big]$ *s.t.* $b'' = (s''_1, \ldots, s''_N) \in \mathcal{PE}[b']$ iff *the following three conditions hold:*

- $s''_j = s'_j$;
- $\forall i \in [1, N], i \neq j$, $s''_i \equiv_\tau s'_i$ *and* $\sigma_{[t_0,t_{l-1}]}(s''_i) = \sigma_{[t_0,t_{l-1}]}(s'_i)$*, where* t_l *is the timestamp of event* $e_l(s_i) = nextEvent\big(s_i, e_k(s_j)\big)$*;*
- $\exists K$ *finite s.t.* $\forall i \in [1, N], i \neq j, \exists k_i \leq K$, $\sigma_{[t_{l+k_i},\infty]}(s''_i) = \sigma_{[t_l,\infty]}(s'_i)$.

Each behavior in $\mathcal{PE}[b']$ is obtained from b' by possibly inserting other stalling events in any signal of b', but only at "later" timestamps, i.e. to postpone informative event that

$$
\begin{array}{c}
b = \begin{cases} \sigma(s_1) = \iota_1\, \tau\, \iota_2\, \tau\, \iota_3\, \tau\, \iota_4\, \tau\, \tau\, \tau\, \iota_2\, \tau\, \iota_4\, \tau\, \iota_3\, \tau\, \iota_1\, \tau\, \iota_2\, \tau\, \tau\, \tau\, \ldots \\ \sigma(s_2) = \tau\, \iota_5\, \tau\, \iota_1\, \tau\, \iota_6\, \tau\, \iota_4\, \tau\, \tau\, \tau\, \iota_5\, \tau\, \iota_6\, \tau\, \iota_7\, \tau\, \iota_8\, \tau\, \tau\, \tau\, \iota_7\, \tau\, \ldots \end{cases} \\
\textbf{Stall} \quad \Downarrow \quad \textbf{Move} \\
b' = \begin{cases} \sigma(s'_1) = \iota_1\, \tau\, \iota_2\, \tau \boxed{\tau} \iota_3\, \tau\, \iota_4\, \tau\, \tau\, \tau\, \iota_2\, \tau\, \iota_4\, \tau\, \iota_3\, \tau\, \iota_1\, \tau\, \iota_2\, \tau\, \tau\, \tau\, \ldots \\ \sigma(s'_2) = \tau\, \iota_5\, \tau\, \iota_1\, \tau\, \iota_6\, \tau\, \iota_4\, \tau\, \tau\, \tau\, \iota_5\, \tau\, \iota_6\, \tau\, \iota_7\, \tau\, \iota_8\, \tau\, \tau\, \tau\, \iota_7\, \tau\, \ldots \end{cases}
\end{array}
$$

Figure 3.4: Behavior $b' = stall(e_5(s_1), b)$ is obtained stalling the fifth event of signal s_1 of behavior b.

follows $e_k(s_j)$ with respect to the lexicographic order $\leq_{lo}$. Observe that a procrastination effect returns a behavior that latency dominates the original behavior.

At the core of the theory of latency-insensitive protocols lies the notion of patient process. A patient process can take stall moves on any signal of its behaviors by reacting with the appropriate procrastination effects [8].

Definition 3.16 *A process P is* patient iff

$$\forall b = (s_1, \ldots, s_N) \in P,\ \forall j \in [1,N],\ \forall e_k(s_j) \in \mathcal{E}_\iota(s_j),\ \left(\mathcal{PE}\left[stall(e_k(s_j), b)\right] \cap P \neq \emptyset \right)$$

Hence, the result of a stall move on one of the events of a patient process may not satisfy the process, but one of the behaviors of the procrastination effect corresponding to the stall move does satisfy the process.

3.3 Composing Patient Systems

Patience is the key condition for the system components to be combinable according to the present approach. In Section 3.4 I discuss how to make a process patient and how to build complex latency-insensitive systems by composing patient processes. The foundations of these results lie in the following three theorems. These theorems together guarantee that the notions of patience and latency equivalence are compositional.

[8]For instance, if an input event of P is stalled, then some output events of P will be delayed if a down-link process Q requests to delay an output event of P (back-pressure) then future input events of P will be delayed.

3.3.1 Compositionality of Patient Processes

Lemma 3.1 *Let P_1 and P_2 be two patient processes. Let $b_1 \in P_1$, $b_2 \in P_2$ be two behaviors with the same lexicographic order s.t. $b_1 \equiv_\tau b_2$. Then, there exists a behavior $b' \in (P_1 \cap P_2)$, $b_1 \equiv_\tau b' \equiv_\tau b_2$.*

Proof (constructive). Let $b_1 = (r_1, \ldots, r_N) \in P_1$ and $b_2 = (q_1, \ldots, q_N) \in P_2$ be the two behaviors with the same lexicographic order. Since b_1 and b_2 are latency equivalent, each event in b_1 has a corresponding event in b_2 and vice versa (see Definition 3.8). Let $r_1 \equiv_\tau q_1, \ldots, r_N \equiv_\tau q_N$. Let $W = \{w \mid \exists k \in \mathbb{N},\ \exists l \in \mathbb{N}\ \ (\, k \neq l \wedge ord(e_k(r_j)) = ord(e_l(q_j)) = w\,)\}$ be the set of ordinals associated to pairs of corresponding events of b_1 and b_2 whose timestamps differ. Define the distance between behaviors b_1, b_2 as

$$d(b_1, b_2) = \begin{cases} \max\left\{\frac{1}{2^w} \mid w \in W\right\} & \text{if } W \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

This distance is reminiscent of the Cantor metric. Thus, b_1 and b_2 have distance equal to zero if all pairs of corresponding events are *aligned*[9]. In this case, b_1 and b_2 are identical, i.e. they are the same behavior that belongs to $(P_1 \cap P_2)$. Now suppose that $d(b_1, b_2) = \frac{1}{2^{w_1}} \neq 0$: in this case, w_1 is the smallest ordinal among those which are associated to unaligned pairs of corresponding events. Without loss of generality, let $p_1 = (e_k(r_j), e_l(q_j))$ be the pair of corresponding events whose ordinal is equal to w_1 and let $l > k$. Apply a stall move to $e_k(r_j)$ to obtain a new behavior $b_1' = (s_1', \ldots, s_N') = stall(e_k(r_j), b_1) \equiv_\tau b_1$. Obviously, $slack(e_{k+1}(s_j'), e_l(q_j)) = slack(e_k(r_j), e_l(q_j)) - 1$. Note that b_1' is not necessarily a behavior of P_1. However, since P_1 is patient, there exists $b_1'' = (s_1'', \ldots, s_N'') \equiv_\tau b_1$ s.t. $b'' \in \mathcal{PE}\left[stall(e_k(r_j), b_1)\right] \cap P_1$. Since, by definition of procrastination effect, $s_j'' = s_j'$, then also $slack(e_{k+1}(s_j''), e_l(q_j)) = slack(e_k(r_j), e_l(q_j)) - 1$. Since the procrastination effect may postpone only events following $e_k(r_j)$ in the lexicographic order $\leq_{lo}$, then all the pairs of corresponding events of b_1'' and b_2 with ordinal smaller than w are still aligned. Now, there are two possibilities: if $slack(e_{k+1}(s_j''), e_l(q_j)) = 0$, then one more pair has been aligned and $d(b_1'', b_2) < d(b_1, b_2)$; otherwise, this slack can be reduce by 1 through a

[9]A pair of corresponding events is said *aligned* if the events are synchronous, or, according to Definition 3.8, if their slack is 0.

repetition of the same procedure starting from behavior b_1''. In any case, after $l-k$ steps of the procedure outlined above, the result is a behavior $b_1^\star \equiv_\tau b_1$ that satisfies P_1 and s.t. $d(b_1^\star, b_2) < d(b_1, b_2)$, because one more pair of corresponding events has been aligned. An *alignment step* has been completed.

Now, if $d(b_1^\star, b_2) = 0$ then there are no more unaligned pairs, the two behaviors are identical and the lemma is proven since $b_1^\star \equiv_\tau b_1$. Instead, if $d(b_1^\star, b_2) = \frac{1}{2^{w_2}} \neq 0$ then the next unaligned pair p_2 of corresponding events must be considered and a second alignment step must be performed. Note that at the m-th step, after aligning pair p_m with ordinal w_m, the slack of some of the pairs following p_m in the lexicographic order may increase, but all the pairs preceding p_m remain aligned. This sequence of alignment steps produces two sequences of behaviors (one of behaviors in P_1 latency equivalent to b_1 and one of behaviors in P_2 latency equivalent to b_2), whose distance is decreasing monotonically. Since both b_1 and b_2 contain the same finite number of informative events [10], the set U of pairs of unaligned corresponding events is also finite. The slack of each of these pair is also a finite number. At the m-th step, there are at most h_m sub-steps that need to be performed to align p_m, where h_m is the starting slack for p_m. In the worst case, each behavior $b^\star$ obtained during the sub-steps of the alignment step may have slacks of all the remaining unaligned pairs increased by at most K (see Definition 3.15). Hence, at the end of the m-th step, $|U|$ has been decreased by one, while all the slacks of its remaining elements have been increased by at most $h_m \cdot K$, a finite number. Thus for $|U| \geq l > m$, the new slacks for the remaining unaligned pairs is $h_l' \leq h_l + h_m \cdot K$. Globally, the worst case requires to perform $|U|$ alignment steps and for each of them a finite number of sub-steps. Hence, the two sequences of behaviors are also finite and the last elements of these sequences do not have unaligned pairs, and, therefore, have distance equal to zero. □

Theorem 3.2 *If P_1 and P_2 are patient processes then $(P_1 \cap P_2)$ is a patient process.*

Proof. Let $b = (s_1, \ldots, s_N)$ be a behavior in $P_1 \cap P_2$. Consider behaviors $b_1 = (r_1, \ldots, r_N) \in P_1$ and $b_2 = (q_1, \ldots, q_N) \in P_2$, s.t. $b_1 = b_2 = b$. For all $j \in [1,N]$ and for all $k \in \mathbb{N}$, let $e_k(s_j) \in \mathcal{E}_\iota(s_j)$. Since $b_1 = b_2 = b$, then $e_k(r_j) \in \mathcal{E}_\iota(r_j)$ and $e_k(q_j) \in \mathcal{E}_\iota(q_j)$. Let $b' =$

[10]Recall that the number of informative events for every behavior considered in latency-insensitive designs is finite.

$$\begin{tikzcd}
 & b^* = \iota_1\ \tau\ \iota_2\ \tau\ \tau\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \iota_5\ \tau\ \tau\ \ldots & \\
b_1 = \iota_1\ \tau\ \iota_2\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \iota_5\ \tau\ \tau\ \ldots \arrow[ur] & & b_2 = \iota_1\ \iota_2\ \tau\ \tau\ \iota_3\ \iota_4\ \tau\ \iota_5\ \tau\ \tau\ \ldots \arrow[ul] \\
 & b_{ref} = \iota_1\ \iota_2\ \iota_3\ \iota_4\ \iota_5\ \tau\ \tau\ \ldots \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 3.5: Sketch for the proof on the compositionality of latency equivalence.

$stall\big(e_k(s_j),b\big) \equiv_\tau b$. Similarly, $b_1' = stall\big(e_k(r_j),b_1\big) \equiv_\tau b_1$ and $b_2' = stall\big(e_k(s_j),b_2\big) \equiv_\tau b_2$. Since P_1 is patient there exists a behavior $b_1'' \equiv_\tau b_1$ s.t. $b_1'' \in \mathcal{PE}[b_1'] \cap P_1$ and since P_2 is patient there exists a behavior $b_2'' \equiv_\tau b_2$ s.t. $b_2'' \in \mathcal{PE}[b_2'] \cap P_2$. Notice that $b_1 = b_2$ implies that $b_1'' \equiv_\tau b_2''$, however, it is not necessarily the case that $b_1'' = b_2''$. In fact, procrastination effects may have misaligned pairs of corresponding informative signals which come after $\big(e_k(r_j), e_k(s_j)\big)$ with respect to lexicographic order $\leq_{lo}$. Since $b_1 = b_2$ share the same lexicographic order, by Lemma 3.1, there exists a behavior $b'' \equiv_\tau b_1'' \equiv_\tau b_2''$ s.t. $b'' \in P_1 \cap P_2$. The construction of b'' given in the proof of Lemma 3.1 involves only unaligned pairs of corresponding events between b_1'' and b_2'' and all these unaligned pairs correspond to informative events which come after $e_k(s_j)$ with respect to lexicographic order $\leq_{lo}$. Further, since the number of informative events is finite, the number of unaligned pairs is also finite. Hence, each signal s_i'' of b'' is obtained by inserting a finite number of stalling events not earlier than timestamp t_l, with $e_l(s_i) = nextEvent\big(s_i, e_k(s_j)\big)$. Therefore, by Definition 3.15, $b'' \in \mathcal{PE}\Big[stall\big(e_k(s_j),b\big)\Big]$. Since $b'' \in P_1 \cap P_2$, then $(P_1 \cap P_2)$ is a patient process. □

Figure 3.5 illustrates the above proof for the case when the two behaviors are just 1-tuple signals.

Theorem 3.3 *For all patient processes P_1, P_2, P_1', P_2':*

$$(P_1 \equiv_\tau P_1') \wedge (P_2 \equiv_\tau P_2') \Rightarrow (P_1 \cap P_2) \equiv_\tau (P_1' \cap P_2')$$

Proof. Let $b = (s_1, \ldots, s_N)$ be a behavior in $P_1 \cap P_2$. Latency equivalence implies that there must be behaviors $b_1 = (r_1, \ldots, r_N) \in P_1'$ and $b_2 = (q_1, \ldots, q_N) \in P_2'$ such that $b_1 \equiv_\tau b \equiv_\tau b_2$. Since b_1 and b_2 are latency equivalent and P_1' and P_2' are patient, Lemma 3.1 guarantees that there must be a latency-equivalent behavior $b' \in (P'_1 \cap P'_2)$. The other direction of the proof is symmetric. □

Therefore, one can replace any process in a system of patient processes by a latency-equivalent process, and the resulting system will be latency equivalent. A similar theorem holds for replacing strict processes with patient processes.

Theorem 3.4 *For all strict processes P_1, P_2 and all patient processes P'_1, P'_2:*

$$(P_1 \equiv_\tau P'_1) \wedge (P_2 \equiv_\tau P'_2) \Rightarrow (P_1 \cap P_2) \equiv_\tau (P'_1 \cap P'_2)$$

Proof. The argument that every behavior in $(P_1 \cap P_2)$ has an equivalent in $(P'_1 \cap P'_2)$ is as in Theorem 3.3. For the other direction, let b' be a behavior in $P'_1 \cap P'_2$. Latency equivalence implies that there must be behaviors $b_1 \in P_1$ and $b_2 \in P_2$ such that $b_1 \equiv_\tau b' \equiv_\tau b_2$. Since P_1 and P_2 are strict, b_1 and b_2 are also strict. Being latency equivalent, they must therefore be equal. Thus $b_1 \in (P_1 \cap P_2)$. □

This means that it is possible to replace all processes in a system of strict processes by corresponding patient processes, and the resulting system will be latency equivalent to the original one. This is the core of latency-insensitive design: take a design based on the assumption that computation in each functional module and communication among modules "take no time" (synchronous hypothesis) [11] and replace it with a design where communication does take time (more than one virtual clock cycle) and, as a result, signals are delayed, but without changing the sequence of informative events observed at the system level. In other words, take a design obtained through strictly synchronous composition of functional modules and replace it with a design made of patient processes (a latency-equivalent process for each module) communicating by means of patient channels.

3.3.2 Channels and Buffers

The tagged-signal model provides the notion of channel to formalize the composition of processes [144]. A *channel* is a connection [12] constraining two signals to be identical.

[11]Or the equivalent assumption that communication and computation are completed in exactly one clock cycle.

[12]See Section 3.2.1 for the definition of connection.

Definition 3.17 *A* channel $C(i,j) \subset \mathcal{S}^N, i,j \in [1,N]$ *is a process s.t.*

$$b = (s_1,...,s_N) \in C(i,j) \Leftrightarrow s_i = s_j$$

As the following lemma formally proves, a channel is not a patient process because it lacks the capacity of storing an event and delaying its communication between two processes.

Lemma 3.5 *A channel $C(i,j) \subset \mathcal{S}^N$ is not a patient process.*

Proof: Let $b = (s_1,...,s_N)$ be a behavior of a channel $C(i,j)$ and, without loss of generality, suppose that $s_i \leq_c s_j$. Consider a pair of corresponding informative events in s_i and s_j: $e_k(s_i) = (v_1,t_k)$ and $e_k(s_j) = (v_2,t_k)$. Since $b \in C(i,j)$ then $s_i = s_j$ and, therefore, $v_1 = v_2 \neq \tau$. Moreover, $s_i = s_j$ implies that $ord(e_k(s_i)) = ord(e_k(s_j))$ and, since $s_i \leq_c s_j$, $e_k(s_j) = nextEvent(s_j, e_k(s_i))$. Without loss of generality, suppose that $e_k(s_i)$ and $e_k(s_j)$ are followed by $(l-1) > 0$ stalling events, i.e., formally, that for $l > 1$, $|\mathcal{F}_\iota[\sigma_{[t_k,t_{k+(l-1)}]}(s_i)]| = |\mathcal{F}_\iota[\sigma_{[t_k,t_{k+(l-1)}]}(s_j)]| = 0$. Then, consider informative event $e_{k+l}(s_i) = (v_{k+l}, t_{k+l})$. By Definition 3.13, $e_{k+l}(s_i) = nextEvent(s_i, e_k(s_j))$. Now, let $b' = (s'_1,...,s'_N) = stall(e_k(s_j), b)$ be the behavior obtained by applying a stall move on $e_k(s_j)$. At timestamp t_k, s'_j presents a stalling event, while the event of s'_j corresponding to $e_k(s_j)$ is $e_{k+1}(s'_j) = (v_2, t_{k+1})$, which occurs at timestamp t_{k+1}. Then, consider any behavior $b'' = (s''_1,...,s''_N) \in \mathcal{PE}[b']$. By Definition 3.15, since $e_{k+l}(s_i) = nextEvent(s_i, e_k(s_j))$, then $\sigma_{[t_0,t_k]}(s''_i) = \sigma_{[t_0,t_k]}(s'_i) = \sigma_{[t_0,t_k]}(s_i)$. In particular, $\sigma_{[t_k,t_k]}(s''_i) = \sigma_{[t_k,t_k]}(s_i) \neq \tau$ and therefore, $\sigma_{[t_k,t_k]}(s''_i) \neq \sigma_{[t_k,t_k]}(s''_j)$, which, finally, implies that $s''_i \neq s''_j$. Hence, $\forall b'' \in \mathcal{PE}[b']$ $(b'' \notin C(i,j))$ and, by Definition 3.16, $C(i,j)$ is not patient. □

Hence, to formally model communication delays as well as pipeline stages it is necessary to introduce the notion of buffer. A buffer is a process relating two signals s_i, s_j of a behavior b and is defined by means of 3 parameters: capacity c, minimum forward latency l_f and minimum backward latency l_b. A buffer forces signals s_i, s_j to be latency equivalent and to satisfy the following relationships for all natural numbers k:

1. the difference between the amount of information events seen at s_i from timestamp zero to timestamp $k - l_f$ and the amount of informative events seen at s_j from timestamp zero to timestamp k is greater or equal than zero;

$$\sigma(s_i) = \underbrace{\boxed{k-1 \text{ informative events and } (g-k-1)\ \tau \text{ events}}}_{\cdot \leq_{lo} e_h(s_j)} \ \underbrace{\boxed{e_g}\ \boxed{\tau \text{ events}}\ \boxed{e_{g+1}}\ \cdots}_{e_h(s_j) \leq_{lo} \cdot}$$

$$\sigma(s_j) = \underbrace{\boxed{k-1 \text{ informative events and } (h-k-1)\ \tau \text{ events}, h \geq g}}_{\cdot \leq_{lo} e_g(s_i)} \ \underbrace{\boxed{e_h}\ \boxed{\tau \text{ events}}\ \boxed{e_{h+1}}\ \cdots}_{e_g(s_i) \leq_{lo} \cdot}$$

Figure 3.6: Relationships between the notion of *nextEvent* and corresponding events in pairs of latency-equivalent signals.

2. the difference between the amount of information events seen at s_i from timestamp zero to timestamp k and the amount of informative events seen at s_j from timestamp zero to timestamp $k - l_b$ is at most c.

Definition 3.18 *A* buffer $B^c_{l_f,l_b}(i,j)$ *with capacity* $c \geq 0$, *minimum forward latency* $l_f \geq 0$ *and minimum backward latency* $l_b \geq 0$ *is a process s.t.* $\forall i,j \in [1,N]$: $b = (s_1,...,s_N) \in B^c_{l_f,l_b}(i,j)$ iff $(s_i \equiv_\tau s_j)$ *and* $\forall k \in \mathbb{N}$

$$0 \leq |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-l_f)}]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]| \qquad (3.1)$$

$$c \geq |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-l_b)}]}(s_j)]| \qquad (3.2)$$

By definition, given a pair of indexes $i,j \in [1,N]$, for all $l_b, l_f, c \geq 0$, all buffers $B^c_{l_f,l_b}(i,j)$ are latency equivalent. Observe also that buffer $B^0_{0,0}(i,j)$ coincides with channel $C(i,j)$ and, therefore, is not a patient process. Since buffers having unit latencies are particularly interesting, it is important to establish under which conditions they are patient processes [13].

Lemma 3.6 *If* s_i, s_j *are two signals s.t.* $s_i \equiv_\tau s_j$ *and* $s_i \leq_c s_j$, *then*

1. $\forall g \in \mathbb{N}$ *s.t.* $e_g(s_i) \in \mathcal{E}_\iota(s_i)$, *nextEvent*$(s_j, e_g(s_i))$ *is the corresponding event of* $e_g(s_i)$ *in* s_j.

2. $\forall h \in \mathbb{N}$ *s.t.* $e_h(s_j) \in \mathcal{E}_\iota(s_j)$, *nextEvent*$(s_i, e_h(s_j))$ = *nextEvent*$(s_i, e_g(s_i))$, *where* $e_g(s_i)$ *is the corresponding event of* $e_h(s_j)$ *in* s_i.

[13]Recalling the discussion of Section 2.2.4, a nonpatient buffer $B^0_{0,0}(i,j)$ corresponds to a stateless repeater, while any patient buffer $B^c_{l_f,l_b}(i,j)$ corresponds to a stateful repeater.

Proof.: Let $e_h(s_j)$ be the corresponding event of $e_g(s_i)$. By Definition 3.8, $ord(e_g(s_i)) = ord(e_h(s_j)) = k$. By Definition 3.12, since $s_i \leq_c s_j$, then $e_g(s_i) \leq_{lo} e_h(s_j)$. For all informative events $e'(s_j)$ with $ord(e'(s_j)) < k$ it is clearly the case that $e_g(s_i) \not\leq_{lo} e'(s_j)$. Instead, $e_g(s_i) \leq_{lo} e''(s_j)$ for all informative events $e''(s_j)$ with $ord(e''(s_j)) > k$. However, $e_h(s_j)$ is clearly the minimum ordinal informative event of s_j that follows $e_g(s_i)$ with respect to the lexicographic order $\leq_{lo}$ and, therefore, by Definition 3.13, $nextEvent(s_j, e_g(s_i)) = e_h(s_j)$. The second relation can be easily proven using the previous relation. Consider $nextEvent(s_j, e_g(s_i)) = e_h(s_j)$ where $e_g(s_i)$ and $e_h(s_j)$ are corresponding events. Let $e'_g(s_i)$ be $nextEvent(s_i, e_g(s_i))$. Then, necessarily, $ord(e'_g(s_i)) = ord(e_g(s_i)) + 1 = k+1$. Since $ord(e_h(s_j)) = ord(e_g(s_i)) = k$, then, by Definition 3.12, $e_h(s_j) \leq_{lo} e'_g(s_i)$. Furthermore, $e'_g(s_i)$ is also the minimum ordinal event of s_i which comes after $e_h(s_j)$ according to lexicographic order $\leq_{lo}$ and, therefore, by Definition 3.13, $nextEvent(s_i, e_h(s_j)) = e'_g(s_i) = nextEvent(s_i, e_g(s_i))$ □

Figure 3.6 illustrates the previous lemma. The following theorem guarantees that a buffer with unit latencies is a patient process as long as its capacity is greater than zero.

Theorem 3.7 *Let $l_b = l_f = 1$. For all $c \geq 1$, $B^c_{1,1}(i,j)$ is patient* iff $s_i \leq_c s_j$.

Proof.: First, if $l_b = l_f = 1$ then inequalities (3.1) and (3.2) become:

$$0 \leq |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-1)}]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]| \tag{3.3}$$

$$c \geq |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-1)}]}(s_j)]| \tag{3.4}$$

["only if" part]: Prove by contradiction that if $s_i \not\leq_c s_j$ then $B^c_{1,1}(i,j)$ is not a patient process. Suppose $s_i \not\leq_c s_j$. For all $c \geq 1$, let $b = (s_1, \dots, s_N)$ be a behavior of $B^c_{1,1}(i,j)$ s.t. $\sigma(s_i) = \iota_0\, \iota_1\, \tau\, \tau\, \tau \dots$ and $\sigma(s_j) = \tau\, \iota_0\, \iota_1\, \tau\, \tau\, \tau \dots$. Let $b' = (s'_1, \dots, s'_N) = stall(e_0(s_i), b)$, with $e_0(s_i) = (\iota_0, t_0)$. Clearly, $b' \notin B^c_{1,1}(i,j)$, because inequality (3.3) does not hold for $k = 1$ since $s'_j = s_j$. Further, for all $b'' = (s''_1, \dots, s''_N) \in \mathcal{PE}\left[stall(e_0(s_i), b)\right]$ it can be proven that $b'' \notin B^c_{1,1}(i,j)$. In fact, since $s''_i = s'_i$, $b'' \in B^c_{1,1}(i,j)$ *iff* $\sigma_{[t_1,t_1]}(s''_j) = \tau$. But, consider that $ord(e_0(s_i)) = ord(e_1(s_j))$ and, since $s_i \not\leq_c s_j$, then $e_0(s_i) \not\leq_{lo} e_1(s_j)$. Further, $ord(e_0(s_i)) = ord(e_2(s_j)) - 1$. Therefore, $e_2(s_j) = nextEvent(s_j, e_0(s_i))$. Recall that, by Definition 3.15 of procrastination effect, $\sigma_{[t_0,t_{l-1}]}(s''_j) = \sigma_{[t_0,t_{l-1}]}(s'_j)$, where t_l is the timestamp of event $nextEvent(s_j, e_0(s_i))$. Hence, in this case, $t_l = t_2$ and $\sigma_{[t_0,t_1]}(s''_j) = \sigma_{[t_0,t_1]}(s'_j)$.

Since $s'_j = s_j$, $\sigma_{[t_1,t_1]}(s''_j) = \sigma_{[t_1,t_1]}(s'_j) = \sigma_{[t_1,t_1]}(s_j) = \iota_0 \neq \tau$. This implies that $b'' \notin B^c_{1,1}(i,j)$. Hence, $\mathcal{PE}\left[stall(e_0(s_i),b)\right] \cap B^c_{1,1}(i,j) = \emptyset$ and $B^c_{1,1}(i,j)$ is not patient.

["if" part]: Prove that if $s_i \leq_c s_j$ then $B^c_{1,1}(i,j)$ is patient. For all $c \geq 1$, let $b = (s_1,\ldots,s_N)$ be a behavior of $B^c_{1,1}(i,j)$. It is necessary to analyze three distinct cases in order to stall respectively an informative signal of s_i, s_j and s_n with $n \in ([1,N]/\{i,j\})$.

1. For all $g \in \mathbb{N}$, such that $e_g(s_i) \in \mathcal{E}_\iota(s_i)$, let $b' = (s'_1,\ldots,s'_N) = stall(e_g(s_i),b) \equiv_\tau b$. Since $s'_j = s_j$, $b' \notin B^c_{1,1}(i,j)$ *iff* inequality (3.3) does not hold for some $k \in \mathbb{N}$. In fact, b' satisfies the other two conditions of Definition 3.18, because $b' \equiv_\tau b$ and to insert a stalling event on s_i (while s_j remains the same) cannot induce a violation of inequality (3.4). Now, suppose first that b' satisfies also inequality (3.3) for all $k \in \mathbb{N}$: then, there exists at least a behavior which belongs to $\mathcal{PE}\left[stall(e_g(s_i),b)\right] \cap B^c_{1,1}(i,j)$ and this behavior is b', because, $\forall g \forall i, stall(e_g(s_i),b) \in \mathcal{PE}\left[stall(e_g(s_i),b)\right]$. A more interesting case is when inequality (3.3) does not hold: in this case $b' \notin B^c_{1,1}(i,j)$. Then, consider a behavior $b'' = (s''_1,\ldots,s''_N)$ s.t. $\forall n \in [1,N], n \neq j, (s''_n = s'_n)$, while s''_j is obtained from s'_j by inserting a stalling event at timestamp t_h, where t_h is also the timestamp of event $e_h(s_j) = nextEvent(s_j,e_g(s_i))$. Clearly, this construction guarantees that $b'' \in \mathcal{PE}\left[stall(e_g(s_i),b)\right]$. It remains to be proven that $b'' \in B^c_{1,1}(i,j)$. First, by construction, $b'' \equiv_\tau b$. Then, check whether s''_i, s''_j satisfy inequalities (3.3) and (3.4) for all $k \in \mathbb{N}$. First, since $s_i \equiv_\tau s_j$ and $s_i \leq_c s_j$, by Lemma 3.6, $e_h(s_j) = nextEvent(s_j,e_g(s_i))$ is the corresponding event of $e_g(s_i)$ in s_j. Hence $ord(e_g(s_i)) = ord(e_h(s_j)) = ord(e_g(s''_i)) = ord(e_h(s''_j))$ and, recalling Definition 3.8, $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_i)| = |\mathcal{F}_\iota[\sigma_{[t_0,t_h]}(s_j)|$. Since, by hypothesis, s_i,s_j satisfy inequality (3.3) for all $k \in \mathbb{N}$. then $g < h$. Compare s''_i and s''_j respectively with s_i and s_j: s''_i has been derived by s_i inserting a τ at t_g, while s''_j has been derived by s_j inserting a τ at t_h. Hence, the following 4 equations can be derived. Further, each term in these equations can be bounded using the fact that s_i,s_j satisfy inequalities (3.3) and (3.4) for all $k \in \mathbb{N}$:

$$\forall k \in [0, g-1],\ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i'')| = |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)| \le \ |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-1}]}(s_j)| + c \quad (3.5)$$
$$\forall k \in [g, \infty[,\ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i'')| = |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-1}]}(s_i)| \le \ |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-2}]}(s_j)| + c \quad (3.6)$$
$$\forall k \in [0, h-1],\ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j'')| = |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)| \le \ |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-1}]}(s_i)| \quad (3.7)$$
$$\forall k \in [h, \infty[,\ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j'')| = |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-1}]}(s_j)| \le \ |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-2}]}(s_i)| \quad (3.8)$$

Now, keeping in mind that $g < h$, it is easy to prove that:

- using inequality (3.7) and equation (3.5), s_i'', s_j'' satisfy inequality (3.3), $\forall k \in [0, g-1]$.
- using [14] inequality (3.7) and equation (3.6), s_i'', s_j'' satisfy inequality (3.3), $\forall k \in [g, h-1]$.
- using inequality (3.8) and equation (3.6), s_i'', s_j'' satisfy inequality (3.3), $\forall k \in [h, \infty[$.
- using inequality (3.5) and equation (3.7), s_i'', s_j'' satisfy inequality (3.4), $\forall k \in [0, g-1]$.
- using inequality (3.6) and equation (3.8), s_i'', s_j'' satisfy inequality (3.4), $\forall k \in [g, h-1[$.
- using inequality (3.6) and equation (3.8), s_i'', s_j'' satisfy inequality (3.4), $\forall k \in [h, \infty[$.

Therefore, $b'' \in B^c_{1,1}(i,j)$.

2. Consider now $b' = (s_1', \ldots, s_N') = stall(e_h(s_j), b) \equiv_\tau b$, where for all $h \in \mathbb{N}$, $e_h(s_j) \in \mathcal{E}_\iota(s_j)$. Let $e_q(s_i) = nextEvent(s_i, e_h(s_j))$ and $e_p(s_i)$ be the corresponding event of $e_h(s_j)$ in s_i: then, since $s_i \equiv_\tau s_j$ and $s_i \le_c s_j$, by Lemma 3.6, $e_q(s_i) = nextEvent(s_i, e_p(s_i))$. Now, construct $b'' = (s_1'', \ldots, s_N'')$ in such a way that $\forall n \in [1, N], n \neq i, (s_n'' = s_n')$, while s_i'' is obtained from s_i' by inserting a stalling event at timestamp t_g, where $g = \min_{k \in [h+1, \infty[}\{k \mid e_k(s_i) \in \mathcal{E}_\iota(s_i)\}$. Hence, if $q > h$ then $e_g(s_i) = e_q(s_i)$ else $e_q(s_i) \le_{lo}$

[14]Recall that $\sigma_{[t_k,t_k]}(s_i) = \tau$.

$e_g(s_i)$. In both cases, this construction guarantees that $b'' \in \mathcal{PE}\left[stall(e_h(s_j), b)\right]$. It remains to be proven that $b'' \in B^c_{1,1}(i,j)$. First, by construction, $b'' \equiv_\tau b$. Then, check whether s''_i, s''_j satisfy inequalities (3.3) and (3.4) for all $k \in \mathbb{N}$. Compare s''_i and s''_j respectively with s_i and s_j: s''_i has been derived by s_i inserting a τ at t_g, while s''_j has been derived by s_j inserting a τ at t_h. Hence, previous relations (3.5-3.8) hold also in this case. Now, keeping in mind that here $h < g$, it is easy to prove that:

- using inequality (3.7) and equation (3.5), s''_i, s''_j satisfy inequality (3.3), $\forall k \in [0, h-1]$.
- using inequality (3.8) and equation (3.5), s''_i, s''_j satisfy inequality (3.3), $\forall k \in [h, g-1]$.
- using inequality (3.8) and equation (3.6), s''_i, s''_j satisfy inequality (3.3), $\forall k \in [g, \infty[$.
- using inequality (3.5) and equation (3.7), s''_i, s''_j satisfy inequality (3.4), $\forall k \in [0, h-1]$.
- using [15] inequality (3.5) and equation (3.7), s''_i, s''_j satisfy inequality (3.4), $\forall k \in [h, g-1[$.
- using inequality (3.6) and equation (3.8), s''_i, s''_j satisfy inequality (3.4), $\forall k \in [g, \infty[$.

Therefore, $b'' \in B^c_{1,1}(i,j)$ in this case too.

3. Finally, for all $n \in ([1,N]/\{i,j\})$ let $b' = (s'_1, \ldots, s'_N) = stall(e_h(s_n), b) \equiv_\tau b$, where for all $h \in \mathbb{N}$, $e_h(s_n) \in \mathcal{E}_\iota(s_n)$. Then, trivially, $b' \in \mathcal{PE}\left[stall(e_h(s_n), b)\right] \cap B^c_{1,1}(i,j)$.

In conclusion, combining all three cases gives the following result:

$$\forall b = (s_1, \ldots, s_N) \in B^c_{1,1}(i,j), \forall n \in [1,N], \forall e_k(s_n) \in \mathcal{E}_\iota(s_n), (\mathcal{PE}\left[stall(e_k(s_n), b)\right] \cap B^c_{1,1}(i,j) \neq \emptyset)$$

Hence, $B^c_{1,1}(i,j)$ is patient. □

[15]Recall that $\sigma_{[t_h,t_h]}(s_j) = \tau$.

3.3.3 Pipelining Patient Channels

As anticipated in Section 3.1, one of the goals of the latency-insensitive design methodology is to be able to *pipeline* a communication channel by inserting an arbitrary amount of storage elements. In the framework of the present theory, this operation corresponds to adding patient buffers on the channels between patient processes. Here I sketch how this can be done. Then, in the next section, I introduce a particular class of patient buffers, called *relay stations*, that presents some optimal characteristics.

Consider a strict system $P_{strict} = \bigcap_{m=1}^{M} P_m$ with N strict signals $s_1, \ldots, s_N$. As explained in Section 3.2.1, processes can be defined over different signal sets and to compose them it may be necessary to formally extend the set of signals of each process to contain all the signals of all processes. However, without loss of generality, consider the particular case of composing M processes that are already defined on the same N signals. Hence, any generic behavior $b_m = (s_{m_1}, \ldots, s_{m_N})$ of P_m is also a behavior of P_{strict} *iff* for all $l \in [1,M], l \neq m$ process P_l contains a behavior $b_l = (s_{l_1}, \ldots, s_{l_N})$ s.t. $\forall n \in [1,N]$ $(s_{l_n} = s_{m_n})$. In fact, system P_{strict} may be derived through the connection of the M processes via $(M-1) \cdot N$ channel processes $C(l_n, (l+1)_n)$, where $l \in [1,(M-1)]$ and $n \in [1,N]$. Further, assume to "decompose" any channel process $C(m_n, l_n)$ with an arbitrary number X of channel processes $C(m_n, x_1), C(x_1, x_2), \ldots, C(x_{X-1}, l_n)$, by adding $X-1$ auxiliary signals, each of them forced to be equal to $m_n = l_n$. The theory developed in Section 3.2 guarantees that if each process $P_m \in P_{strict}$ is replaced with a latency-equivalent patient process and each channel $C(i,j)$ with a patient buffer $B^1_{1,1}(i,j)$ of unit latency, then the resulting system $P_{patient}$ is patient and latency equivalent to P_{strict}. In fact, *having a patient buffer in a patient system is equivalent to having a channel in a strict system.* Furthermore, "decomposing" a channel $C(i,j)$ has no observable effect on a strict system and, therefore, it is possible to add an arbitrary number of these patient buffers into the corresponding patient system to replace this channel. Using patient buffers with unit latencies makes it possible to vary arbitrarily the communication latency of a channel. This action represents the theoretical equivalent of wire pipelining, the method discussed in Section 2.2.4 to deal with long wires while building SOCs with nanometer technologies. The essence of wire pipelining is the distribution of several instances of these buffers along the long wires (which implement the inter-module on-chip

$$B^1_{1,1}\begin{cases}\sigma(s_1) &= \iota_1\,\tau\,\iota_2\,\tau\,\iota_3\,\tau\,\iota_4\,\tau\,\tau\,\tau\,\iota_5\,\tau\,\iota_6\,\tau\,\iota_7\,\tau\,\iota_8\,\tau\,\iota_9\,\tau\,\tau\,\tau\,\iota_{10}\,\tau\,\ldots\\ \sigma(s_2) &= \tau\,\iota_1\,\tau\,\iota_2\,\tau\,\iota_3\,\tau\,\iota_4\,\tau\,\tau\,\tau\,\iota_5\,\tau\,\iota_6\,\tau\,\iota_7\,\tau\,\iota_8\,\tau\,\tau\,\tau\,\iota_9\,\tau\,\iota_{10}\,\tau\,\ldots\end{cases}$$

$$B^2_{1,1}\begin{cases}\sigma(s_1) &= \iota_1\,\iota_2\,\iota_3\,\tau\,\tau\,\iota_4\,\iota_5\,\iota_6\,\tau\,\tau\,\tau\,\iota_7\,\tau\,\iota_8\,\iota_9\,\iota_{10}\,\ldots\\ \sigma(s_2) &= \tau\,\iota_1\,\iota_2\,\iota_3\,\tau\,\tau\,\iota_4\,\tau\,\tau\,\tau\,\iota_5\,\iota_6\,\iota_7\,\tau\,\iota_8\,\iota_9\,\iota_{10}\,\ldots\end{cases}$$

Figure 3.7: Comparing the behaviors of finite buffers $B^1_{1,1}(s_1,s_2)$ and $B^2_{1,1}(s_1,s_2)$.

communication channels) in such a way that the wires get decomposed in segments whose physical lengths can be spanned in a single cycle of the system clock (the *real clock*).

3.3.4 Relay Stations

The following Lemma 3.8 proves that no behaviors in $B^1_{1,1}(i,j)$ may contain two informative events of s_i, s_j which are synchronous, i.e. there cannot be any timestamp for which both s_i and s_j present an informative event. This implies that the maximum achievable throughput across such a buffer is 0.5, which may be considered suboptimal. Instead, buffer $B^2_{1,1}(i,j)$ is the minimum capacity buffer which is able to "transfer" one informative unit per timestamp, thus allowing, in the best case, to communicate with maximum throughput equal to 1. Figure 3.7 compares two possible behaviors of these buffers.

Lemma 3.8 *$B^2_{1,1}(i,j)$ is the minimum capacity buffer with $l_f = l_b = 1$ s.t. for all K, closed intervals of* $\mathbb{N}$, *:*

$$\exists b^K = (s^K_1,\ldots,s^K_N) \in B^2_{1,1}(i,j) \wedge \forall k \in K,\ \left(e_k(s^K_i) \in \mathcal{E}_\iota(s^K_i) \wedge e_k(s^K_j) \in \mathcal{E}_\iota(s^K_j)\right) \quad (3.9)$$

Proof.: Relation (3.9) says that $B^2_{1,1}(i,j)$ is the minimum capacity buffer with $l_f = l_b = 1$ containing a behavior b^K where s_i and s_j present $|K|$ consecutive pairs of synchronous informative events (i.e., the two informative events of each a pair have the same timestamp t_k) for all K, closed intervals of $\mathbb{N}$. Notice that the only buffer with $l_f = l_b = 1$ having capacity less than $B^2_{1,1}(i,j)$ is $B^1_{1,1}(i,j)$. First I show that $B^2_{1,1}(i,j)$ contains at least one behavior b^K satisfying relation (3.9) and then I prove that the same is not true for any behavior of $B^1_{1,1}(i,j)$. It is easy to construct an example of such a behavior for any K.

For instance, consider a behavior $b = (s_1, \ldots, s_N)$ s.t. $\sigma(s_i^K) = \iota_1\ \iota_2\ \ldots\ \iota_K\ \tau\ \tau\ \tau \ldots$ and that $\sigma(s_j^K) = \tau\ \iota_1\ \iota_2\ \ldots\ \iota_K\ \tau\ \tau\ \tau \ldots$ Clearly, $s_i^K \equiv_\tau s_j^K$ and inequalities (3.3) and (3.4) (with $c = 2$) are satisfied for any $k \in K$. Hence, $b \in B_{1,1}^2(i,j)$. Moreover, at all timestamps $t_1, t_2, \ldots, t_K$ both s_i^K and s_j^K present an informative event.

Now, consider $B_{1,1}^1(i,j)$. If $c = 1$, the combination of inequalities (3.3) and (3.4) gives that $\forall k \in \mathbb{N}$:

$$\begin{array}{lclcl} |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-1)}]}(s_j)]| + 1 & \geq & |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| & \geq & |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k+1)}]}(s_j)]| \\ |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-1)}]}(s_i)]| & \geq & |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]| & \geq & |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k+1)}]}(s_i)]| - 1 \end{array}$$

Hence, for all behaviors $b = (s_1, \ldots, s_N) \in B_{1,1}^1(i,j)$, signals s_i, s_j are not only latency equivalent but also correlated according to a very regular pattern (see Figure 3.7) which can be summarized in two properties:

there are no two synchronous informative events in s_i, s_j;

for all timestamps, informative events appear alternately on s_i and on s_j (possibly, at not consecutive timestamps). The first property is a negation of relation (3.9). □

Definition 3.19 *The buffer* $B_{1,1}^2$ *is called a* relay station.

Hence, a relay station is a particular instance of the buffer process that has the property to be the minimum capacity patient buffer which is able to sustain a communication throughput of one informative unit per timestamp. Observe that the formal specification of a buffer provided by Definition 3.18 is powerful because it is extremely abstract: it does not even specify whether s_i or s_j are input or output signals. In Chapter 4 I present an RTL implementation of a relay station, based on a simple latency-insensitive protocol, which uses two signals (a stop signal and a void signal) to control the flow of data on a channel. Using model checking it is possible to formally verify that this RTL implementation is a refinement of the above formal specification.

3.4 Latency-Insensitive Design

In this section, I formally present the notion of latency-insensitive design as an application of the concepts previously introduced. To do so, I assume that:

- the pre-designed functional modules are synchronous functional processes, which are called *core processes*, or *pearl processes*
- the core processes are strictly causal;
- the core processes belong to a particular class of processes called *stallable*.

Composing a set of pre-designed, synchronous, functional modules in the most efficient way is fairly straightforward under the assumption that the synchronous hypothesis holds. This composition corresponds to a composition of strict processes since there is a priori no need for inserting stalling events. However, as I argued in Chapter 1, it is very likely that the synchronous hypothesis will not be valid for many implementations due to their distributed nature. If indeed the processes to be composed are patient, then adding an appropriate number of relay stations yields a process that is latency equivalent to the strict composition. In fact, since relay stations are patient processes, their insertion in a patient system guarantees that the system remains patient. Further, since they have minimum latencies equal to one, they can be repetitively inserted on a channel to pipeline it while increasing its latency. Finally, since a relay station forces its two signals to be latency equivalent, the resulting system remains latency-equivalent to the original one. Therefore, if the definition of correct behavior is that the sequences of informative events do not change, then inserting relay stations solves the distributed communication problem without affecting the system functionality.

However, requiring core processes to be patient at the onset is definitely too demanding from a practical point of view. Still, in practice, a patient system can be *derived* from a strict one as follows: first, take each strict process P_m and compose it with a set of auxiliary processes to obtain an equivalent patient process P'_m. To be able to do so, all processes P_m must satisfy the simple condition that the processes be stallable, which is formally specified in the next section. Then, put together all patient processes by connecting them with relay stations. The set of auxiliary processes implements a "queuing mechanism" across the signal of P_m in such a way that informative events are buffered and reordered before being passed to P_m: informative events having the same ordinal are passed to P_m synchronously.

In the next section, I present the notion of stallable processes and prove that every stallable process can be encapsulated into a shell process which acts as an interface towards a latency-insensitive protocol.

3.4.1 Stallable Processes

In the sequel I consider only strictly causal processes and assume that for each of them the well founded order $\leq_c$ of Definition 3.12 subsumes the causality relations among its signals, i.e. formally: $\forall i \in I, \forall j \in O, (s_i \leq_c s_j)$.

The following definition of synchronous stalling captures a particular application of a set of stall moves on a behavior of a causal process.

Definition 3.20 *The application of* synchronous stalling *at timestamp h on a behavior* $b = (s_1, \ldots, s_Q, s_{Q+1}, \ldots, s_N) \in P$ *of a process P with* $I = \{1, \ldots, Q\}$ *and* $O = \{Q+1, \ldots, N\}$ *returns a behavior* $b_\tau = synchStall(h, b)$ *such that*

$$\begin{aligned} b_\tau &= stall\Big(e_{h+1}(s_N), stall\Big(e_{h+1}(s_{N-1}), stall\Big(\ldots \\ \ldots\ & e_{h+1}(s_{Q+1}), stall\Big(e_h(s_Q), stall\Big(e_h(s_{Q-1}), stall\Big(\ldots \\ \ldots\ & e_h(s_3), stall\Big(e_h(s_2), stall\Big(e_h(s_1), b\Big)\Big)\ldots\Big)\Big)\Big)\ldots\Big)\Big)\Big) \end{aligned}$$

Obviously, $b_\tau = synchStall(h, b) \equiv_\tau b$. Behavior b_τ is not necessarily a behavior of P. But this is always the case when P is a stallable process.

Definition 3.21 *A process P with* $I = \{1, \ldots, Q\}$ *and* $O = \{Q+1, \ldots, N\}$ *is* stallable *when for all its behaviors* $b = (s_1, \ldots, s_Q, s_{Q+1}, \ldots, s_N) \in P$ *and for all* $k \in \mathbb{N}$ *:*

$$\forall i \in I \ \left(\sigma_{[t_k, t_k]}(s_i) = \tau\right) \ \Leftrightarrow \ \forall j \in O \ \left(\sigma_{[t_{k+1}, t_{k+1}]}(s_j) = \tau\right)$$

and for all $h \in \mathbb{N}$ *behavior* $b_\tau = synchStall(h, b)$ *is also a behavior of P, i.e.* $b_\tau \in P$.

Hence, while a patient process tolerates arbitrary distributions of stalling events among its signals (as long as causality is preserved), a stallable process demands more regular patterns: τ symbols can only be inserted synchronously (i.e., with the same timestamp) on all input signals and this insertion implies the synchronous insertion of τ symbols on all output signals at the following timestamp. To assume that a functional process is stallable is quite reasonable with respect to a practical implementation: for instance, most hardware systems can be stalled (see Section 4.1). At the same time, the most popular models of computation can easily represent stallability: for instance, consider an extension of finite state machines (FSM) [124, 240] such that each machine M has an extra input, that, if equal to τ, forces M to stay in the current state and to emit τ at the next cycle.

Generally, a stallable process P is not a patient process.

Lemma 3.9 *A generic stallable process P is not patient.*

Proof.: Recalling Definitions 3.16 and 3.12, it is sufficient to assume that P has at least two input signals s_i, s_j with $s_i \leq_c s_j$ and that a stall move is applied to an event $e_k(s_j)$ of a strict behavior b of P. Then, recalling Definition 3.15, no procrastination effect can be applied to the stalled behavior b' in order to have a stall move for $e_l(s_i)$ with $ord(e_l) = ord(e_k)$ that is synchronous with the previous stall move. Hence, $\mathcal{PE}\left[stall(e_k(s_j), b)\right] \cap P = \emptyset$. □

3.4.2 Shell Encapsulation of Stallable Processes

The goal of this section is to define a group of functional processes that can be composed with a stallable process P to derive a patient process which is latency equivalent to P. The first step is to consider a process that aligns all the informative events across a set of channels.

Definition 3.22 *An* equalizer *E is a process, with $I = \{1, \ldots, Q\}$ and $O = \{Q+1, \ldots, 2 \cdot Q\}$, such that for all behaviors $b = (s_1, \ldots, s_Q, s_{Q+1}, \ldots, s_{2 \cdot Q}) \in E$, $\forall i \in I, (s_i \equiv_\tau s_{Q+i})$ and $\forall k \in \mathbb{N}$:*

- $\forall i, j \in O \ \Big((\sigma_{[t_k,t_k]}(s_i) = \tau) \Rightarrow (\sigma_{[t_k,t_k]}(s_j) = \tau) \Big)$
- $\min_{i \in I} \Big\{ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| \Big\} - \max_{j \in O} \Big\{ |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]| \Big\} = 0$

The first relation forces the output signals to have stalling events only synchronously, while the second guarantees that at every timestamp the number of informative events occurred

$$\boxed{\begin{array}{lll} \sigma(s_1) = \ldots\ \iota_2\,\tau\,\iota_1\,\iota_3\,\iota_1\,\tau\,\iota_3\,\tau\,\tau\ \ldots & & \sigma(s_4) = \ldots\ \tau\,\iota_2\,\tau\,\iota_1\,\tau\,\iota_3\,\iota_1\,\tau\,\iota_3\ \ldots \\ \sigma(s_2) = \ldots\ \tau\,\iota_8\,\tau\,\iota_4\,\tau\,\iota_7\,\iota_8\,\tau\,\iota_8\ \ldots & \rightarrow & \sigma(s_5) = \ldots\ \tau\,\iota_8\,\tau\,\iota_4\,\tau\,\iota_7\,\iota_8\,\tau\,\iota_8\ \ldots \\ \sigma(s_3) = \ldots\ \tau\,\iota_6\,\tau\,\iota_5\,\iota_5\,\tau\,\iota_9\,\tau\,\iota_6\ \ldots & & \sigma(s_6) = \ldots\ \tau\,\iota_6\,\tau\,\iota_5\,\tau\,\iota_5\,\iota_9\,\tau\,\iota_6\ \ldots \end{array}}$$

Figure 3.8: Example of a behavior of an equalizer E with $I = \{1,2,3\}$ and $O = \{4,5,6\}$.

at any output is always equal to the least number of informative events seen by any input signal up to that timestamp.

Example Figure 3.8 illustrates a possible behavior of an equalizer. Notice how the presence of a stalling event at a certain input at a given timestamp does not necessarily force the presence of a stalling event on all outputs at the same timestamp. For instance, while the two stalling events on s_2 and s_3 at timestamp t_1 do force stalling events on all output signals at t_1, instead the stalling event present on s_1 at t_2 does not result in any timestamp on the output signals: this is due to the fact that at timestamp t_2 all input signals have seen at least one informative event while no output event have occurred on the output signal up to t_1. ■

Definition 3.23 *An* extended relay station $\mathcal{ERS}$ *is a process with* $I = \{i\}$ *and* $O = \{j,l\}$, $i \neq j \neq l$ *s.t. signals* s_i, s_j *are related by inequalities (3.1) and (3.2) of Definition 3.18 (with* $l_f = l_b = 1$ *and* $c = 2$*) and* $\forall k \in \mathbb{N}$*:*

$$\sigma_{[t_k,t_k]}(s_l) = \begin{cases} 1 & \textit{if}\, |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_{k-1}]}(s_j)]| = 2 \\ 0 & \textit{otherwise} \end{cases}$$

Definition 3.24 *A* stalling signal generator $\mathcal{SSG}$ *is a process with* $I = \{1,\ldots,Q\}$ *and* $O = \{Q+1\}$ *s.t.* $\forall b = (s_1,\ldots,s_{Q+1}), \forall k \in \mathbb{N}, \forall i \in [1,Q], \left(\sigma_{[t_k,t_k]}(s_i) \in [0,1]\right)$ *and*

$$\sigma_{[t_k,t_k]}(s_{Q+1}) = \begin{cases} \tau & \textit{if}\ \exists j \in [1,Q]\ \left(\sigma_{[t_k,t_k]}(s_j) = 1\right) \\ 0 & \textit{otherwise} \end{cases}$$

As illustrated in Figure 3.9, any stallable process P can be composed with an equalizer, a stalling signal generator and some extended relay stations to derive a patient process which is latency equivalent to P.

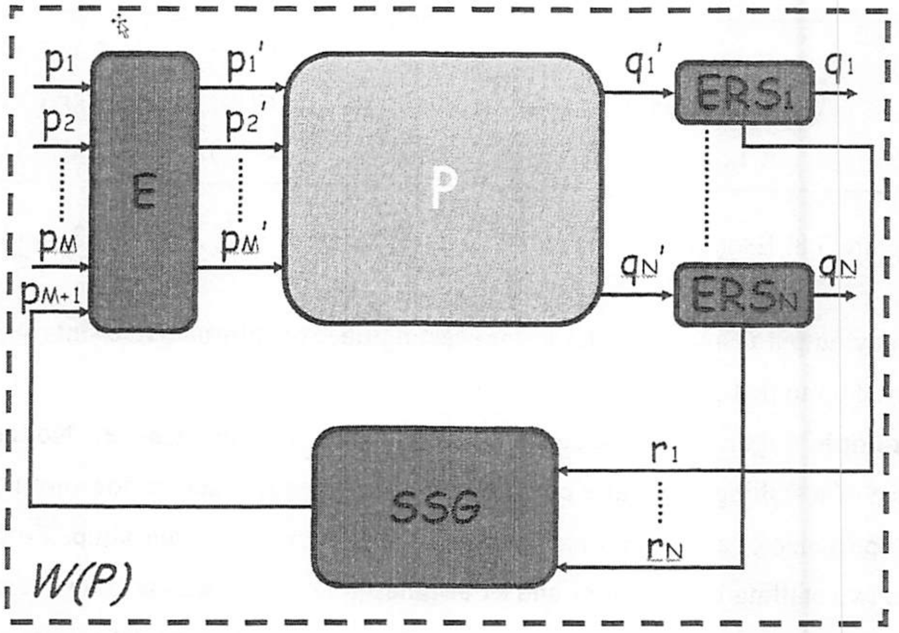

Figure 3.9: Encapsulation of a stallable process P into a shell $W(P)$.

Definition 3.25 *Let P be a stallable process with $I_P = \{p'_1, \ldots, p'_M\}$ and $O_P = \{q'_1, \ldots, q'_N\}$. A* shell process *(or,* wrapper process*) $W(P)$ of P is the process with $I_W = \{p_1, \ldots, p_M\}$ and $O_W = \{q_1, \ldots, q_N\}$ which is obtained by composing P with the following processes:*

- *an equalizer E with $I_E = \{p_1, \ldots, p_M, p_{M+1}\}$ and $O_E = \{p'_1, \ldots, p'_M, p'_{M+1}\}$,*
- *N extended relay stations $\mathcal{ERS}_1, \mathcal{ERS}_2, \ldots, \mathcal{ERS}_N$ s.t. $I_j = \{q'_j\}$ and $O_j = \{q_j, r_j\}$, with $j \in [1, N]$*
- *a* stalling signal generator *$\mathcal{SSG}$ with $I_G = \{r_1, \ldots, r_N\}$ and $O_G = \{p_{M+1}\}$.*

Theorem 3.10 *Let $W(P)$ be the shell process of Definition 3.25. Process*

$$W = proj_{I_W \cup O_W}(W(P))$$

is a patient process that is latency equivalent to P.

Proof: Throughout the proof I follow the index notation of Definition 3.25.

["$W \equiv_\tau P$" part]: Let $b' = (s_{p'_1}, \ldots, s_{p'_M}, s_{q'_1}, \ldots, s_{q'_N})$ be a behavior of P and $b = (s_{p_1}, \ldots, s_{p_{M+1}}, s_{q_1}, \ldots, s_{q_N}, s_{p'_1}, \ldots, s_{p'_{M+1}}, s_{q'_1}, \ldots, s_{q'_N}, s_{r_1}, \ldots, s_{r_N})$ one of $W(P)$. Let $b_W = proj_{I_W \cup O_W}(b) = (s_{p_1}, \ldots, s_{p_M}, s_{q_1}, \ldots, s_{q_N})$ be the corresponding behavior of W. Then, by Definition 3.22 of equalizer, $\forall i \in I_E, (s_{p_i} \equiv_\tau s_{p'_i})$, and, by definition of relay station, $\forall i \in [1, N], (s_{q_i} \equiv_\tau s_{q'_i})$. Therefore, $b_W \equiv_\tau b$.

["W patient" part]: Recalling Definition 3.16, the goal is to prove that

$$\forall b = (s_{p_1}, \ldots, s_{p_M}, s_{q_1}, \ldots, s_{q_N}) \in W,\ \forall j \in I_W \cup O_W,\ \forall e_k(s_j) \in \mathcal{E}_\iota(s_j),$$
$$\left(\mathcal{PE}\left[stall(e_k(s_j), b)\right] \cap W \neq \emptyset \right)$$

Consider first stalling any input signal of W: for all $s_{p_i}, i \in [1, M]$ and all $e_g(s_{p_i}) \in \mathcal{E}_\iota(s_{p_i})$, let $b' = (s'_{p_1}, \ldots, s'_{p_M}, s'_{q_1}, \ldots, s'_{q_N}) = stall(e_g(s_{p_i}), b)$. Two cases may happen:

1. there exists a signal $s_{p_k}, k \in [1, M], k \neq i$, s.t. $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_k})]| < |\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_i})]|$. Consequently, by Definition 3.22, no additional stalling events are added at the output of E nor, ultimately, on the output signals of W. Hence, even though stall move $stall(e_g(s_{p_i}), b)$ affects only signal s_{p_i}, still $b' \in W$ and $\mathcal{PE}\left[stall(e_k(s_{p_i}), b)\right] = b'$ (the stall move is "absorbed" by the equalizer). Therefore, $\mathcal{PE}\left[stall(e_k(s_{p_i}), b)\right] \cap W \neq \emptyset$.

2. s_{p_i} is a signal of b s.t. $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_i})]| = \min_{i \in I}\{|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_i})]|\}$. In this case, the insertion of a stalling event on s_{p_i} at t_g implies that all the output signals of equalizer E have a stalling event at t_g. Then, by analyzing the interrelationships among the components of W, it is easy to verify that $b' \notin W$. In fact, all the output signals of P are forced to have a stalling event at t_{g+1}, and, similarly, all the output signals $s_{q_1}, \ldots, s_{q_N}$ to have it at t_{g+2}. Hence, $\forall j \in N, e_{g+2}(s_{q_j}) \in \mathcal{E}_\iota(s_{q_j})$ must be also stalled. Then, since move $stall(e_g(s_{p_i}), b)$ does not affect any other signal but s_{p_i}, $b' \notin W$. However, since $ord(e_{g+2}(s_{q_j})) = nextEvent((s_{q_j}), e_g(s_{p_i}))$, the insertion of one stalling event on each of the shell outputs at t_{g+2} is compatible with the definition of procrastination effect and, therefore, $\mathcal{PE}\left[stall(e_k(s_{p_i}), b)\right] \cap W \neq \emptyset$.

Next, consider stalling any output signal of W: for all $s_{q_j}, j \in [1, N]$ and all $e_h(s_{q_j}) \in \mathcal{E}_\iota(s_{q_j})$, let $b' = (s'_{p_1}, \ldots, s'_{p_M}, s'_{q_1}, \ldots, s'_{q_N}) = stall(e_h(s_{q_j}), b)$. By definition of stall move,

$\forall m \in [1,M]$, $(s'_{p_m} = s_{p_m})$ and $\forall n \in [1,N], n \neq j$, $(s'_{q_n} = s_{q_n})$. Hence, again, $b' \notin W$. In fact, the insertion of a stalling event on signal s_{q_j} at t_h has an impact on signal $s_{q'_j}$ of $\mathcal{ERS}_j$ that is *constrained* to stall the input event $e_{h+l}(s_{q'_j}) \in \mathcal{E}_\tau(s_{q'_j})$ occurring l timestamps later [16]. As a consequence, all the outputs of the stallable process P must have a stalling event at t_{h+l}. While no other stalling events are forced on s_{q_j} of $\mathcal{ERS}_j$ at t_{h+l+1}, all the remaining relay stations $\mathcal{ERS}_r, r \in [1,N]/\{j\}$ must stall their $e_{h+l+1}(s_{q_r}) \in \mathcal{E}_\tau(s_{q_r})$. Hence, $b' \notin W$ because $stall(e_h(s_{q_j}), b)$ does not affect any other signal but s_{q_j}. However, $\forall r \in ([1,N]/\{j\})$, since $e_{h+l+1}(s_{q_r}) = nextEvent(s_{q_r}, e_g(s_{q'_r}))$, where $e_g(s_{q'_r}) = nextEvent(s_{q'_r}, e_h(s_{q_j}))$, then $e_h(s_{q_j}) \leq_{lo} e_{h+l+1}(s_{q_r})$. Hence, the insertion of one stalling event on each shell output s_{q_r}, $r \in ([1,N]/\{j\})$ at t_{h+l+1} is compatible with the definition of procrastination effect. Therefore, $\mathcal{PE}\Big[stall(e_h(s_{q_i}), b)\Big] \cap W \neq \emptyset$. $\square$

3.4.3 Latency-Insensitive Design Methodology

By putting together the ideas discussed in the previous sections, I can derive the following guidelines for the definition of a generic *latency-insensitive design methodology*:

1. begin with a system of stallable processes and channels;
2. encapsulate each stallable process to yield a corresponding shell process that is patient and latency equivalent to the original one;
3. by inserting the required amount of relay stations on each channel, the latency of the communication among any pair of processes can be arbitrarily varied without affecting the overall system functionality.

This approach clearly *orthogonalizes* computation and communication: in fact, one can build systems by assembling functional core processes (which can be arbitrarily complex as far as they satisfy the stalling assumption) and shell processes (which interface the cores with the channels, by "speaking" the latency-insensitive protocol). While the specific functionality of the system is distributed in the cores, the shell can be automat-

[16]By Definition 3.19, $e_{h+l}(s_{q'_j})$ must be stalled even though $\forall k \in [h+1, h+l-1], (\sigma_{[t_k,t_k]}(s_{q'_j}) = \tau)$.

ically generated around them [17]. Furthermore, the validation of the system now can be efficiently decomposed based on assume-guarantee reasoning and compositional model checking [2, 54, 109, 165, 166]: each shell is verified assuming a given protocol while the protocol is verified separately.

3.4.4 Example: Latency-Insensitive Design Methodology for SOC

With regard to the design of digital integrated circuits, the generic framework of the latency-insensitive design methodology can be used as the formal basis for defining a methodology centered around the following, simple, non-iterative design flow:

1. the designers design and validate the chip as a collection of synchronous modules which can be specified with usual hardware-description languages such as VERILOG [234] or VHDL [5] (RTL specification);
2. each module is automatically encapsulated within a block of control logic (the shell) to make it latency-insensitive;
3. traditional logic synthesis and place&route steps are applied;
4. if the presence of unexpectedly large wire delays makes it necessary, the resulting layout is corrected by simply inserting the right amount of relay stations to meet the clock cycle constraints everywhere.

Notice that the design, layout and routing of individual modules would not need to be changed to reflect any necessary changes in wire latencies during the chip-level layout and wiring process. This clearly represents a significant advantage for future SOC designs, where the designers completing the chip-level integration most likely will not work at the same company as the designers of the individual modules. Furthermore, since it is based on the synchronous hypothesis, the approach facilitates the adoption of state-of-the art formal verification techniques within a new design flow that, for the rest, can be built using traditional and widely-available layout and synthesis CAD tools. The application of the latency-insensitive design methodology to SOC design is discussed in detail in Chapter 4.

[17]This is the reason of the term *shells*: a shell just "protect" the intellectual property (*the pearl*) it contains from the "troubles" of the external communication architecture.

3.5 Related Work

This section briefly discusses the relationships between the theory of latency-insensitive protocols and the work of other researchers in the fields of asynchronous design, high-level synthesis, and synchronous reactive programming languages. The relationships between latency-insensitive design and retiming are discussed in Chapter 7.

3.5.1 Latency Insensitive versus Asynchronous Design

The theory of latency-insensitive protocols is reminiscent of several ideas that have been proposed in the asynchronous design community during the past three decades [73, 106].

In particular, the idea of a design methodology which is inherently modular is already present in the work on *Macromodular Computer Systems* by Clark and Molnar [52, 53]. To separate the design of these modules from the design of the system and make the entire process amenable to automation, the modules must be implemented as *delay-insensitive* circuits [171, 200]. A delay-insensitive circuit is designed to operate correctly regardless of the delays on its gates and wires (unbounded delay model) [237]. However, it has been proven that almost no useful delay-insensitive circuit can be built if one is restricted to a class of simple logic gates [27, 159]. To be able to build complex systems one must use more complex components, which are "externally" delay insensitive, while "internally" are designed by carefully verifying their timing and avoiding or tolerating meta-stability [79, 121, 200]. By slightly relaxing the unbounded delay model and allowing "isochronic forks" [18], practical *quasi-delay-insensitive* circuits can be built using simple logic gates [32]. A further relaxation leads to *speed-independent* circuits, which operate correctly regardless of gate delays, while wire delays are assumed to be negligible [8, 77, 134].

Sutherland's 1989 Turing Award lecture on *micropipelines* [222] contains several ideas that have influenced the work on latency-insensitive protocols. Micropipelines are asynchronous elastic pipelines based on the concepts of transition-signalling and two-phase bundled data interface. They have been used in several projects, including the design of an

[18] A bounded skew is allowed between the different branches of a net.

asynchronous microprocessor [221]. The implementation of latency-insensitive protocols discussed in Section 4.2 uses two control signals for each data channel and shares similarity with the bundled data interface. At the methodological level, common is the focus on composability, i.e. in in defining a design framework to build complex systems by means of simple building blocks. Naturally, the main difference lies in the judgement on the benefits of making the synchronous hypothesis during the early stage of the design process for deriving the specification of such complex systems.

Back in 1985, Van de Sneupschet observed that the decreasing feature size of VLSI devices would have lead to "a decrease of the propagation speed of electrical signals relative to the switching speed", and proposed the use of suitable communication protocols to obtain chip designs whose correct operation is independent of the propagation speed [239]. Van de Sneupschet's *theory of trace structures and composition functions* has more than one contact point with the present work, but differs on the basic fact that leads to the choice of speed-independent circuits over synchronous circuits. Dill has also proposed a *trace theory* for modeling and specifying speed-independent circuits that is the basis for a hierarchical verification approach [77].

Both quasi-delay-insensitive and speed-independent circuits assume that the designers are able to control wire delays, and, therefore, do not appear as interesting alternatives as we move towards nanometer design. Instead, a methodology based on assembling complex modules which are "externally" delay-insensitive seems the right solution, on condition that the synthesis of such modules is not too cumbersome. However, it should be noted that asynchronous approaches do not address the fundamental problem of latency, because an asynchronous design simply slows down to accommodate the slowest component, e.g. the wires.

As we head towards the design of integrated circuits to be fabricated with nanometer technologies, the delays of long intermodule wires are becoming dominant with respect to both the delays of the intra-module wires and those of the logic gates. More importantly, intermodule delays are difficult to predict or to control during the different phases of the design of a chip, leading to an exacerbation of the timing closure problem. Delay-insensitive approaches as well as the latency-insensitive methodology allow the designer to specify and implement the system while assuming that intermodule wire delays may vary arbitrarily. However, while a delay-insensitive system is based on the assumption that the delay between two subsequent events on a communication channel is completely arbitrary, in a latency-insensitive system this arbitrary delay is forced to be a *multiple of the clock period.* The key point is that this kind of *discretization* enables the leveraging of well-accepted design methodologies for the design and validation of synchronous systems. In fact, the important distinction between any of the previous asynchronous design methodologies and the latency-insensitive approach is, essentially, that a latency-insensitive system is specified as a synchronous system. Notice that I write "specified" because from an implementation viewpoint a latency-insensitive communication protocol can also be designed using *handshaking signaling* techniques (such as request/acknowledge protocols), which are typically asynchronous [19]. However, from a specification viewpoint, each module (as well as the overall system) is viewed as a synchronous system relying on the synchronous hypothesis. Now, to specify a complex system as a collection of modules whose state is updated collectively in one instantaneous step is naturally simpler than specifying the same system as the interaction of many components whose state is updated following an intricate set of interdependency relations. Furthermore, the synchronous specification makes it possible to slightly modify the traditional semi-custom design flow by simply inserting a step to encapsulate each synchronous module within a so-called *shell process*. The impact is very different also from a validation point of view because simulation is naturally a less complex task for a synchronous system than for an equivalent asynchronous one. Furthermore, while the performance of a synchronous design can be immediately determined from its clock frequency, to evaluate exactly the performance of an asynchronous one can be challenging because it depends on various factors [247]. As a consequence, asynchronous designers are typically forced to use *ad hoc* approaches at different levels of the design hierarchy [248].

In conclusion, the theory of latency-insensitive design leads to a methodology that can be implemented on top of a commonly adopted design flow, while asynchronous approaches typically forces the designers to use new tools and, more importantly, *to think of the digital system in a completely different way.*

[19]Here the communication bandwidth would be limited by the inverse of the longest of the round trip times between pairs of communicating relay stations.

3.5.2 Latency Insensitivity and Slack Elasticity

The *slack* of a communication channel in an asynchronous system refers to the amount of buffering present in the channel and it is implemented with buffers that are pipeline stages [160]. The process of adjusting the slack on channels to optimize the system throughput is called *slack matching* [66]. Slack matching is similar to the insertion of relay stations in a latency-insensitive system [20]. An important difference is that increasing the channel slack can modify the synchronization among the components of the asynchronous system and lead to incorrect behaviors. This problem, which was studied by Manohar [158], is due to the presence of non-determinism in the system. A *decision point* is a place where the system makes a non-deterministic choice. This corresponds to the case of processes that probe a channel to determine whether a communication on the channel can complete, and then perform different computations depending on the result of the probe [160]. Decision points require the introduction of arbiters. The main result by Manohar [158] is that: *increasing the slack of a channel by one does not affect the correctness of the computation if (and only if) it does not introduce additional decision points in the system.* A channel is *slack elastic* if its slack can be incremented without affecting the correctness of the system. A system is slack elastic when every channel in the system is slack elastic. The highly modular design of the MiniMIPS asynchronous microprocessor was obtained exploiting its slack elasticity [160]. Based on the above definition, latency-insensitive systems can always be considered slack elastic because they are deterministic systems.

3.5.3 Latency-Insensitive Protocols and High-Level Synthesis

Parallels with latency-insensitive design can also be found in some of the ideas that have been proposed in the field of high-level synthesis to schedule the sequential execution of interacting processes under unbounded timing constraints [85, 136]. However, the hardware model used in these works, a polar hierarchical acyclic graph, does not handle efficiently circuits with feedback loops. Furthermore, the authors admit that the application of their technique is restricted to non-pipelined synchronous design. Instead, the theory of latency-insensitive design permits the specification of systems that can be im-

[20]How to optimize the system throughput through relay station insertion is the subject of Chapter 6.

plemented using both synchronous and asynchronous circuits and is aimed to optimize latency/throughput trade-offs by exploiting those pipelining techniques typically used in the design of high-end microprocessors.

3.5.4 The Composition Principle

In [1] Abadi and Lamport discuss the "fundamental problem of composing specifications", i.e. proving that a composite system satisfies its specification if all its components satisfy their specifications. They state the *Composition Principle* informally as follows: *let a system S be the composition of systems $S_1, \ldots, S_n$ and let the following conditions hold:*

1. *S guarantees a property P if each component S_i guarantees a property P_i;*

2. *the environment of each component S_i satisfies an assumption E_i if the environment of S satisfies an assumption E and every S_j satisfies P_j;*

3. *every component S_i guarantees P_i under the environment assumption E_i;*

then S guarantees P under environment assumption E. They also prove a theorem that provides a formal statement of this composition principle and show that if the environment assumptions are safety properties, the properties guaranteed by the system and its components need not be only safety properties, but can include liveness.

In [139] Lamport provides an interesting perspective on the role of compositionality in specifying and validating complex systems. Lamport's main point is that compositional reasoning is just one particular, highly constrained and not particularly "natural" way of decomposing a mathematical proof. And since often it requires extra work, it can be advantageous only when the latter is performed by a computer, for instance with model checking [55]. Lamport does point out, however, that compositional reasoning cannot be avoided when it is necessary to reason about a component that may be utilized in several different systems. In this case an *open-system* specification—one that specifies the component itself, not the complete system—is necessary. Informally, such specification is of the type $E \Rightarrow S$, i.e. the component satisfies S if the environment in which it operates satisfies E. Therefore, it admits that the component misbehaves if the environment does so. A formal study of such open-system specifications is given in [2]. In [139], Lamport recognizes that *"composition of open-system specifications is an attractive problem, having obvious application to reusable software and other trendy concerns*, while he is still skeptical of its practical applications. He writes: *"but in 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It is naive to expect them to go to the extra effort of proving properties of open-system component specifications because they might re-use those components in other systems."*

3.5.5 Latency-Insensitive Protocols and Theory of Desynchronization

Benveniste *et. al.* developed the *theory of desynchronization* [21] in the context of embedded system applications [12, 13]. The main motivation behind this theory is to address the issue of compositionality of synchronous programming languages [16, 17, 99, 100] and thereby to enable modular code generation for distributed architectures. Benveniste *et. al.* advocate a methodology centered on the use of the synchronous paradigm for system specification and validation, which are then followed by a provably correct desynchronization step to derive a distributed implementation. The practical goal of the theory of desynchronization is the development of formal techniques for the synthesis of the protocols necessary to preserve the program semantics when a distributed implementation is performed on asynchronous distributed real-time architectures [22]. In particular, programs written with synchronous languages can be deployed on those *globally-asynchronous locally-synchronous (GALS) architectures* [48] that satisfy the following assumption: *"the architecture obeys the model of a network of synchronous modules interconnected by point-to-point wires, one per each communicated signal; each individual wire is lossless and preserves the ordering of messages, but the different wires are not mutually synchronized."* [15]. Clearly this assumption, whose importance for the theory of desynchronization is explained in detail in [228], is also at the core of the theory of latency-insensitive protocols. This is not the only commonality between the two theories,

[21]More recently, a *desynchronization approach* for IC design has been proposed [63, 64]. I discuss it briefly in Section 8.2.

[22]A general analysis of the distributed implementation problem can be found in two recent papers [11, 14], which formalize the notion of heterogeneous parallel composition and provide a comprehensive study of correct-by-construction deployment on heterogenous distributed systems.

which have been developed completely independently. In October 2001, Benveniste presented the paper *"Some synchronization issues when designing embedded systems from components"* [9] at the First International Workshop on Embedded Software (EMSOFT). The paper presents both theories in the context of a general reflection on the role of synchronization and compositionality in embedded system design.

The central result of the theory of desynchronization is that the properties of *endochrony* and *isochrony* are sufficient to derive robust mechanisms for the synthesis of the protocols necessary for distributed code generation. Benveniste *et al.* distinguish between *endochronous (or proactive)* and *exochronous (or reactive)* processes [23]. Informally, a synchronous process is endochronous when the presence or absence of each signal at each reaction can be inferred incrementally from its current internal state and the presence of other input signals. The immersion of an endochronous process in an asynchronous environment does not change its semantic properties. In other works, endochrony enables desynchronization, i.e. it allows the relaxation of the synchronization barrier of the process by making it part of an asynchronous system. However endochrony is not a compositional property and, therefore, by itself it not sufficient in the effort to reach the goal of the theory of desynchronization, the preservation of the semantics of synchronous composition when the components interact in an asynchronous environment. The property of *isochrony* must be considered too. Informally, two processes are isochronous when, at each reaction, if there is a pair of shared signals that are present and agree on the event value, then for each other pair of shared signals, either they are present and agree on their value or they are absent. Both endochrony and isochrony can be model-checked and synthesized [13]. The main result of the theory of desynchronization is that for any pair of endo-isochronous processes, P and Q, the *desynchronization of the composition of synchronous processes is equal to the desynchronized composition of the corresponding desynchronized processes*, or formally [13]:

$$(P \,||\, Q)^a = (P^a \,||^a\, Q^a)$$

In practice, the theory is the foundation of a correct-by-construction design methodology based on the following steps:

[23]The concepts of endochrony, exochrony and isochrony are formally defined in [13] using the synchronous transition systems (STS) formalism.

1. specify the system as a composition of synchronous programs;
2. synthesize a communication protocol that makes each program *endochronous* and pairs of interacting programs *isochronous*;
3. derive a distributed implementation of the system on an *asynchronous* communication architecture.

In 2003, however, Potop-Butucaru, Caillaud and Benveniste discovered a flaw in the original paper [190]. Although it does not affect the main result of the theory, the flaw prevents *efficient* handling of GALS architectures with more than two components. In their effort to correct this problem, they have extended classical trace theory [76] to a synchronous setting and used it to formalize the concepts of *weak endochrony* and *weak isochrony* [191]. Weak endochrony is a compositional property while weak isochrony is able to exploit concurrency in order to provide lighter communication schemes. Together they form a correct desynchronization criterion that is decidable on finite synchronous systems and make it possible to handle the desynchronization problem for arbitrary GALS architectures. Transforming a general synchronous system to satisfy these properties is easy, *"although making it in an efficient way is a difficult, yet unsolved problem"* [191].

In summary, the following similarities exist between the theory of latency-insensitive design and the theory of desynchronization:

- both follow the principle of correct-by-construction design;
- the role of the τ event in latency-insensitive protocols (denoting lack of an informative signal and the need for stalling) is similar to the role played by the *absence* symbol $\perp$ in the theory of desynchronization; also, the importance of identifying absence of events was already recognized during the early development of the synchronous programming language SIGNAL [16, 94];
- the notion of stallable process in latency-insensitive design is similar to the notion of stuttering-invariant process, an underlying assumption of the theory of desynchronization [9, 13]; *stuttering invariance*, which was first introduced by Lamport as a key attribute of a specification logic [138], means that in each state a process can

always remain "silent" by not changing the state and not emitting any event (i.e. emitting the *absence* symbol $\perp$ on each output signal); latency equivalence is similar to the notion of *stuttering equivalence* on Kripke structures introduced by Browne *et. al.* [26].

- the stalling mechanism and the desynchronization mechanism are similar in that both involve delaying the next reaction until all necessary data become available [9];

Alternatively, the main differences follow:

- the assumption on the global specification of the system is different: the theory of latency-insensitive protocols assumes a strictly synchronous specification with a single common clock: each process reads each input signal and writes every output signal at each tick of this clock [24]; this assumption, which is generally stronger, applies well to hardware design; the theory of desynchronization assumes a synchronous reactive specification where multiple clocks are present; this weaker assumption makes it possible to model multi-rate computation and, potentially, to *"support lighter communication protocols that minimize communication and power consumption"* [191].

- the assumption on the local specification of each system component is also different: the theory of latency-insensitive protocols simply requires stallability (knowledge about the inner logic structure of each core module is not necessary and all cores are treated uniformly as *black-box* modules); the theory of desynchronization requires the analysis of the inner logic structure of each component, even if it is clever in exploiting the information resulting from this analysis [41].

- patience and *weak* endochrony are compositional properties, but endochrony is not;

- latency-insensitive protocols can easily handle causality and enable the development of simple synthesis algorithms; instead, *"neither endochrony, nor weak endochrony take into account causality in the computation of reactions, and efficient synthesis algorithms have yet to be defined for both of them"* [191].

[24] In other words, in the strict system S signals are always present; absence is introduced in an encoded form via stalling events as relay stations are inserted in the patient system S' which is latency-equivalent to S.

I conclude this comparison reporting Benveniste's closing remark: "*'think synchronously - act asynchronously' emerges as a common paradigm for the design of embedded systems from components*" [9].

3.6 Concluding Remarks

Latency-insensitive designs are synchronous distributed systems composed by functional modules that exchange data on communication channels according to a latency-insensitive protocol. The protocol guarantees that latency-insensitive systems, composed of functionally correct modules, behave correctly independently of the channel latencies. This allows for the increase in the robustness of a design implementation since any delay variation of a channel can be "recovered" by changing its latency while the overall system functionality remains unaffected. The protocol works on the weak assumption that the functional modules are stallable. An important application of the proposed theory is represented by the latency-insensitive methodology to design large digital integrated circuits with nanometer technologies. This methodology is the subject of Chapter 4.

Chapter 4

Correct-by-Construction SOC Design Methodology

In which a solid stone goes a long way chasing two elusive birds.

ROBUST techniques to design systems-on-chip with nanometer technologies are the subject of this chapter. Building on the theory of latency-insensitive protocols presented in Chapter 3, I define a correct-by-construction methodology that makes SOC design functionally insensitive to the latency of long wires. Thanks to the compositionality of the notion of latency equivalence, this methodology makes it possible to orthogonalize communication and computation, while the timing requirements imposed by the clock are met by construction. This facilitates the solution of the communication and synchronization issues that naturally arise while assembling pre-designed components, thereby opening the way to IP core reuse in SOC design. Hence, the latency-insensitive design methodology addresses at once the two hard challenges in integrated circuit design discussed in Chapter 2: design productivity and timing closure. Furthermore, it does so without requiring designers to undertake a revolution in their practices: since it is based on the synchronous assumption, it represents a theoretically sound framework from which to develop a new class of design flows for nanometer design through the use of traditional and popular CAD tools.

4.1 Latency-Insensitive Design Methodology for SOC

The theory of latency-insensitive protocols provides the theoretical foundation for a methodology that maintains the inherent simplicity of synchronous design and yet does not suffer from the "interconnect-delay problem". The proposed methodology is centered around the automatic synthesis of a *communication architecture* that implements a latency-insensitive communication protocol.

The starting point is already familiar to the designer of digital integrated circuits (IC): a synchronous specification of the design, based on the assumption that the operation of the final chip will be controlled by a single clock signal. Hence, the system can be thought of as completely synchronous, a collection of functional modules that communicate by means of point-to-point channels having "zero-delay", i.e. a delay negligible with respect to the period of the common clock signal (*synchronous hypothesis*) [1]. I refer to this clock as the *virtual clock* and I call *strict* a system whose specification starts from this assumption. Once the final implementation of the system is derived, its operation is controlled by a *physical clock* that has a precise frequency value. Unfortunately, due to the impact of interconnect delay (discussed in Section 2.2), some of the wires implementing these channels on the final layout may require a delay longer than one real clock cycle to transmit the appropriate signals. Nevertheless, the theory of latency-insensitive protocols guarantees that it is not necessary to complete costly re-design iterations or to slow down the real clock. The basic idea is borrowed from pipelining [108, 133]: partition the long wires into segments by inserting logic blocks called *relay stations*, which have a function similar to the one of registers on a pipelined data-path. The resulting shorter wire segments satisfy the timing requirements imposed by the real clock. While the timing constraints are now met *by construction*, the latency of these channels becomes two or more clock cycles. Still, since the functionality of the design is based on the sequencing of the signals on each channel and not on their exact timing, this modification does not change its functional correctness provided that all its components are *patient processes*.

As discussed in Chapter 3, a module is a patient process if its behavior does not de-

[1]See Section 3.5.1 for a discussion on how the synchronous assumption distinguishes the present design methodology from similar approaches proposed in the realm of asynchronous design during the past three decades.

pend on the latency of the communication channels because it is compliant with a latency-insensitive communication protocol. Locally, the protocol allows a channel to run a number of clock cycles ahead of or behind other channels. Globally, it guarantees that a system, if composed of functionally correct modules, behaves correctly independently from the delays of the channels connecting the modules. As a result, it is possible to synthesize automatically a hardware implementation of the system such that its functional behavior is robust with respect to large variations in communication latency.

The central idea behind this methodology is *to relax time constraints during the early phases of the design when correct measures of the communication delay among modules are not yet available* [2]. Instead, the specification of a complex system is significantly simplified if performed under the synchronous assumption. Once the corresponding physical implementation is completed, if there are mismatches between the time constraints and the interconnect delays among the system modules, they can be easily corrected by inserting the necessary number of relay stations.

4.1.1 Latency-Insensitive Design Flow

A latency-insensitive design flow for integrated circuits consists of a sequence of five basic steps:

1. *Synchronous specification.* The designer starts with a completely synchronous specification of the system as a collection of interacting *functional modules*. These can be either acquired as intellectual property (IP) cores from an (internal or external) third-party or can be specified as "synthesizable" code at behavioral or RTL level using a hardware description language such as VERILOG or VHDL [3]. At this stage, designers do not make any attempt to model accurately the latency of the wires connecting the modules. Instead, they rely on the synchronous hypothesis and assume

[2]As discussed in Sections 2.2.3 and 2.2.4, it is very difficult to estimate accurately the on-chip interconnect latency at the early stages of the design process, and poor estimations exacerbate the timing-closure problem.

[3]Naturally, if system-level design languages such as SYSTEMC [182] and SYSTEMVERILOG [116] are broadly adopted in the future, nothing should prevent this methodology from being extended to embrace them. In fact, latency-insensitive protocols are a promising approach for communication-based design at the *system* level.

that the communication takes one clock cycle on every wire (see Section 1.1 and the comment at the end of Section 3.3.1).

2. *Shell encapsulation.* Each module is encapsulated within a shell logic block. A shell is simply a collection of buffering queues (one for each input port) plus the control logic that serves as an interface between the module and the communication architecture. Hence, the pair module/shell becomes a patient process as defined in Section 3.2.4. This encapsulation step, which is performed automatically, depends only on the input/output (I/O) interface of each module and is independent from the module's specific internal logic structure. The only requirement, discussed below in Section 4.1.2, is that the module be stallable.

3. *Channeling.* The wires connecting the modules are grouped in point-to-point[4] *channels*. Thanks to the interface role played by the shell, each channel operates according to a latency-insensitive communication protocol and is made up of wires and, possibly, relay stations. The wires in a channel are laid out together and share physical characteristics. The hardware implementation of a relay station is obtained by composing *storage elements* like latches or flip-flops[5] together with the control logic necessary to implement the functionality related to the latency-insensitive protocol. In the context of the discussion of Section 2.2.4, relay stations are stateful repeaters, similar to regular pipeline latches, and, therefore, inherently different from stateless repeaters like traditional combinational buffers.

4. *Synthesis and layout generation.* A standard cell netlist and its corresponding final layout are obtained automatically by following the traditional steps of logic synthesis and *place & route*.

5. *Channel pipelining.* A post-layout optimization is performed to fix those design exceptions resulting from long (and slow) wires and to ensure that the timing con-

[4]Working with point-to-point channels is both "natural", given the synchronous specification, and practical. However, the final implementation of the channels does not necessarily have to follow the point-to-point structure.

[5]For a distinction between the various kinds of sequential circuits offering storage capabilities see [70, 132, 193].

straints imposed by the chip clock are met everywhere. Each channel whose delay is greater than the nominal clock period ψ is *critical*, and, therefore, is segmented by distributing the necessary number of relay stations (*channel pipelining*). Some iterations may be required, but they are limited to each channel separately, while the logic and layout of every module remain untouched.

Figure 4.1 illustrates a typical latency-insensitive system with core modules, shells, channels, and relay stations.

This design methodology is a direct application of the principle of orthogonalization of concerns, discussed in Section 2.3, since it separates computation and communication, i.e. the design of the functional modules from the design of the communication architecture and protocol. As a result:

- module design is simplified because it is performed assuming that inter-module communication occurs according to the synchronous hypothesis;
- trade-offs in deriving the communication architecture can be explored up to late stages in the design process because the protocol guarantees that arbitrary latency variations can be easily absorbed by the interface logic;
- since the communication mechanism is automatically synthesized (as described later in this chapter both relay stations and shells can be built with no intervention of the designer, based solely on the theory of latency-insensitive protocols), the designer can focus on the choice of the modules that make up the functionality of the implementation without worrying about synchronization and latency of the overall design.

In fact, the latency-insensitive design methodology facilitates the reuse and assembly of pre-designed and pre-validated IP cores because the control logic necessary for their synchronization and communication is implicitly provided by the shells implementing the latency-insensitive protocol.

At the same time, this design methodology is a direct application of the principle of correct-by-construction design, discussed also in Section 2.3, since it guarantees the robustness of the system functionality regardless of arbitrary latency variations. This property

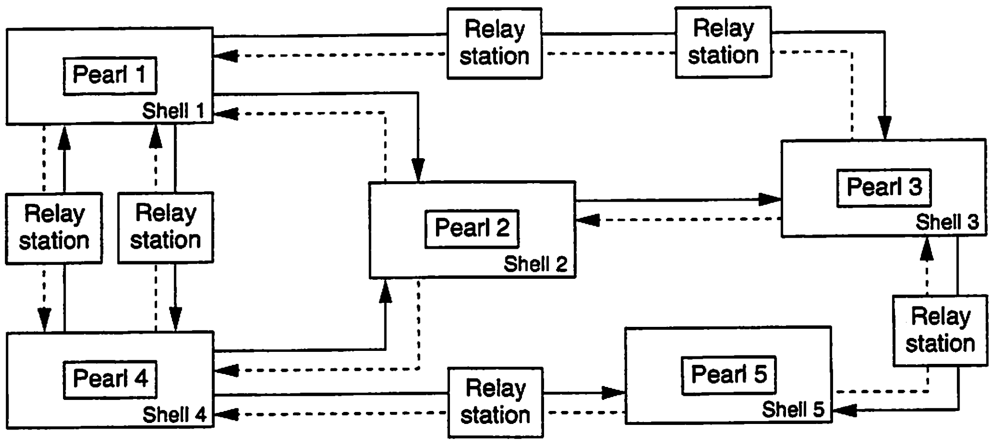

Figure 4.1: Shell encapsulation, relay station insertion and channel back-pressure.

of the system in fact is formally ensured throughout the design process, from the original synchronous specification to the final latency-insensitive implementation. Regardless of the number of relay stations inserted on any channel, the resulting system is guaranteed *by construction* to be functionally equivalent to the original one (without the need of any modification in the logic structure of the individual modules). Naturally, the focus on the by-construction preservation of this particular timing property is motivated by the desire to avoid those multiple costly iterations within the design flow that otherwise result from the inaccurate estimation of wire delays (see Section 2.2.3). It is precisely the lack of this information that leads us away from searching through the "solution space" to find an *optimum* design (an impracticable task) toward a correct-by-construction approach which trades off optimality for *robustness*.

Finally, so far I focused on "the robustness of the system functionality", and haven't discussed about its performance. Naturally, a latency-insensitive design is satisfactory only to the extent that a sufficient throughput can be maintained after increasing channel latencies. How to analyze and optimize system performance is an important issue that is broadly discussed in Section 4.3 and then, in detail, in Chapters 5 and 6.

4.1.2 The Stallability Requirement

Communication design does not have any impact on the design and implementation of the modules provided that they share a fundamental property: *patience* (see Section 3.3). To require that a module be patient at the onset is very demanding. For this reason each module is encapsulated within an appropriate shell which has the task of making it "look patient". Such shells can be automatically generated for all modules as long as each module is a sequential *stallable* functional process. As discussed in Section 3.4.1, stallability is a characteristic whereby a module can stall for any number of clock cycles without losing its internal state (and the overall state of the system). This translates into the assumption that the functionality of the overall system only depends on the relative ordering of the signals and not on their exact timing.

Stallability is a requirement much easier to satisfy than patience. Consequently, it is a property that is both easier to specify and implement in digital hardware design. For instance, at the specification level digital designers commonly use the *finite state machine (FSM)* model of computation to design the control blocks. As explained in Section 3.4.1, it is quite simple to make any FSM specification stallable. Also, working at the implementation level, most hardware systems can be easily made stallable: for instance, consider the addition of a *clock gating* mechanism [180] to any sequential logic block, a practice that has already become standard for most applications because it helps achieving a low power design [44, 45] [6].

4.2 Latency-Insensitive Communication Architecture

In this section I show how to develop a latency-insensitive communication architecture for SOC based on the proposed methodology. The basic building blocks of this architecture are channels, relay stations, and shells. I provide here an operational description of these blocks, whose abstract definitions were given in a denotational fashion in Chapter 3. Taken together, these descriptions represent a reference hardware implementation of a latency-

[6]In fact, latency-insensitive protocols could also be used to develop design automation techniques that make it possible to implement clock-gating for low power design at a fine level of granularity.

insensitive protocol that uses the concept of *back-pressure*. This is by no means the only possible hardware implementation. Still, it is relatively simple, very modular and, as proven in Chapter 5, guarantees optimal performance.

As discussed in Chapter 1, the goal of latency-insensitive design is to preserve the benefits of the synchronous paradigm at the specification level, while introducing at the implementation level those *elements of asynchrony* that are necessary for the effectiveness of distributed design. This does not mean that the implementation must use asynchronous circuitry. In fact, the building blocks that I describe in the next sections refer to a single-clock synchronous implementation [7]. It means instead that at any clock cycle some modules in the chip are free to stall (because they are waiting for some incoming data) while others are active processing data. Meanwhile, the latency-insensitive protocol ensures that the system progresses without errors or deadlocks.

4.2.1 Channels and Back-Pressure

Channels are point-to-point unidirectional links between a *source* module and a *sink* module. Each module is a computational component of the system and, using the terminology of Section 3.4, corresponds to a core functional process and its associated shell process, which acts as an interface towards the latency-insensitive protocol.

Data are transmitted on a channel by means of *packets*: a packet consists of a variable number of fields. In this implementation there are only two basic fields: *payload*, which contains the transmitted data, and *void flag*, which is a one-bit signal that, if set to 1, denotes that no data are present in the packet (*void packet*). If a packet does contain "meaningful" payload data (i.e., void is set to 0) it is called a *true packet*. Clearly each true (void) packet corresponds to an informative (stalling) event as described in Section 3.2.2. A channel is made of wires and relay stations. The number of relay stations in a channel is finite and limits the buffering capability of the channel.

At each clock cycle, the source module may either put a new true packet on the channel

[7]In practice, a latency-insensitive system can be implemented using different circuit design styles: synchronous, asynchronous, globally-asynchronous locally-synchronous (GALS) [48]. Furthermore, in the case of synchronous circuit design, the final chip can have a single-clock domain as well as multiple clock domains.

or, in case no output data are available to be sent, put a void packet on it; meanwhile, at the other side of the channel the sink module retrieves the incoming packet and, based on the value of the void flag, it decides whether to discard it or to store it on its input channel queue for later use. Just as a source module may not be ready to send a true packet, so may a sink module not be ready to receive it. A possible reason could be that the corresponding input queue in the shell of the sink module is full (as explained in the next section). However, as discussed in Chapter 3, the latency-insensitive protocol demands a fully reliable communication among the modules. In other words, no lossy communication link is allowed and all packets must be properly delivered. Consequently, the sink module must be able to interact with the channel (and ultimately with the corresponding source module) to slow down the communication flow and, therefore, avoid the loss of any packet. For this reason, I slightly relax the definition of a channel as a unidirectional communication medium and allow one bit of information, called a *stop flag*, to move in the opposite direction. This signal, which is similar to the *nack* signal in request/acknowledge protocols of asynchronous design, is represented by a dashed line in Figure 4.1. Conceptually, the stop flag can be interpreted as a stalling event moving in the reverse direction. By setting the stop flag equal to one during a certain clock cycle, the sink module informs the channel that the next packet cannot be received and that it must be held until the stop flag is reset. Similar to the sink module, the channel has a limited amount of buffering resources: a channel dealing with a sink module that requires a long stall period may eventually fill up all its relay stations and be forced to send a stop flag to the source module so that the latter will put its packet production on stall. This mechanism to control the flow of information on a channel while guaranteeing that no packet is lost is called *back-pressure*.

4.2.2 Shell Encapsulation

Given a particular core module M, an instance of a shell module can be automatically synthesized as a wrapper to encapsulate M and interface it with the channels so that their combination becomes a patient system. Section 3.4 explains that the only necessary precondition is that M be stallable. At each clock cycle the internal computation of the core must be *fired* only if all inputs have arrived. In other words, the computation of M can start

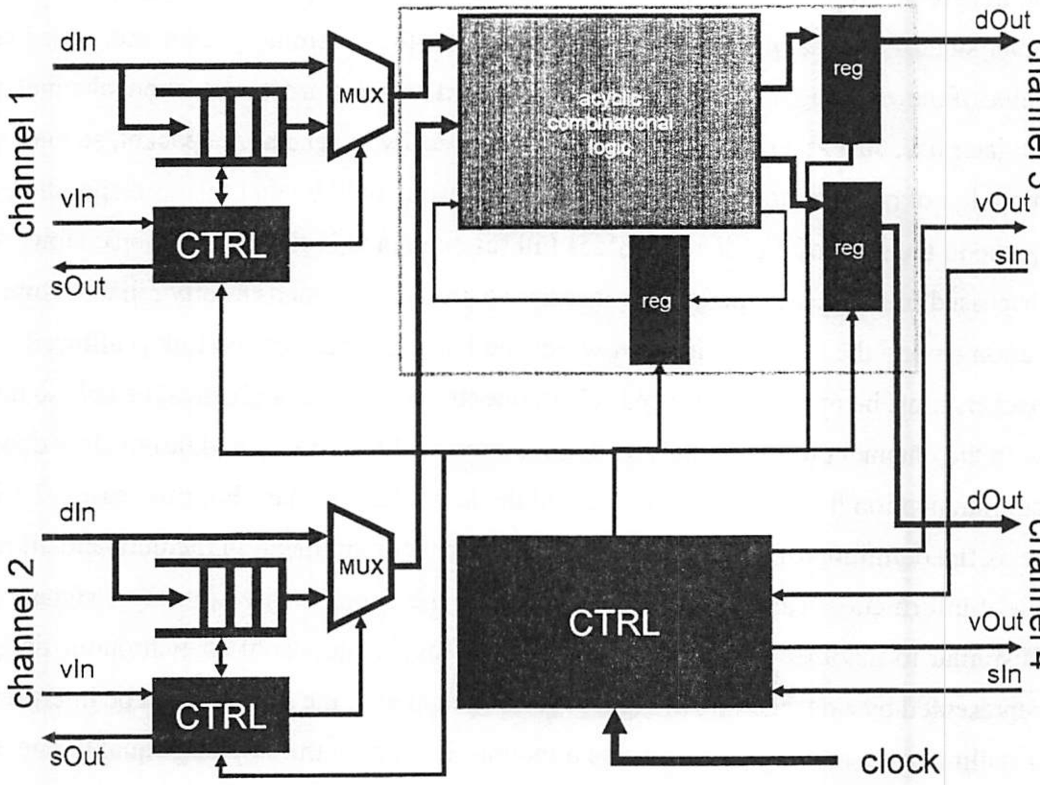

Figure 4.2: Shell encapsulation: making an IP core patient.

for the virtual clock cycle t only when each input channel has produced the corresponding true packet associated to timestamp t. The absence of a true packet is explicitly expressed by the arrival of a void packet, i.e. a packet with the void flag set to 1. Guaranteeing this *input synchronization* is the first task of the shell of a core module. The second task, *input buffering*, is performed whenever an input channel is late with respect to the other channels in producing the next true packet awaited by the core. To avoid losing those true packets that have regularly arrived (and that cannot yet be processed by the core), the shell must store them in a dedicated queue. The third and final task of the shell is called *output propagation*: at each clock cycle, if module M has produced new output data and no output channel has raised a stop flag, then these output data can be transmitted as a new true packet; instead, if any of these two conditions is not verified, then the packet transmitted in the *previous* cycle is re-transmitted, however as a void packet. In summary, a shell for module *M* performs the following functions cyclically:

1. it gets the incoming packets from the input channels, filters away the void packets, extracts the input values for *M* from the payload fields of the true packets, and stores them in its queues;

2. when the input values from all incoming channels are available for the next computation (and, if no output channel has raised a stop flag during the current clock cycle) it passes them to *M* and fires the computation; also, whenever this is actually fired, the shell forwards directly to *M* those true packets that are arriving on channels whose corresponding queues are empty (i.e. the queues are by-passed to avoid wasting a clock cycle);

3. it gets the results of the computation from *M* and, if no output channel has raised a stop flag during the current clock cycle , it routes the results into the output channels.

As it implements the protocol outlined above, the combination shell-core operates according to an AND-causality semantics [96], like a transition of an ordinary marked graph or an actor of a homogeneous synchronous data flow. These models of computation present several useful properties, including the possibility of computing efficiently and exactly the performance of the overall system. These properties are discussed in Chapter 5.

Example Figure 4.2 shows the conceptual diagram of a RTL implementation of a shell encapsulating a core module with two input channels and two output channels. Notice the presence of the input channel queues and the control logic implementing the tasks described above. With respect to the abstract definition of shell encapsulation given in Section 3.4.2, the functionality of the equalizer is performed by the input channel queues, while no extended relay stations are necessary since the incoming stop flags are directly used for the controlling of the queue and the stalling of the core. The stalling/firing mechanism is simply obtained by *gating the clock signal*, which controls the register of the core (as discussed in Section 4.1.2). This implementation guarantees that if a new set of incoming true packets is available and no output channel has raised a stop flag then a new set of outgoing true packets is produced in one clock cycle. The core module of Figure 4.2 is a Moore finite state machine, but it could be replaced by any stallable sequential module as long as it does not have any direct combinational path from its inputs to its outputs. Hence, also any pipelined module can be encapsulated within a shell as illustrated in Figure 6.11. In Section 6.2.1, I will discuss the trade-offs in choosing where to draw the shell boundaries in a pipelined data-path.

The bar charts of Figure 4.3 illustrate the experimental results obtained performing shell encapsulation of a third-party designed IP core (a module computing the *discrete cosine transform* [181]). In this experiment the shell was designed at the RTL level using the VERILOG HDL and was synthesized using commercial logic synthesis tools and standard-cell technology libraries. The queues of the shell were designed having length equal to two [8]. Logic synthesis was performed twice following the same procedure with two distinct technology process: 130*nm* and 90*nm*. The experimental results, however, show that the migration of technology does not have any relevant impact for this particular design. The shell does not add any delay penalty (i.e. the critical path of the core has delay larger than the critical path of the shell). The static power overhead is also minimal. The area occupation of the shell-core pair is about 5% larger than the area occupation of the stand-alone core. ■

The design overhead due to shell encapsulation generally depends on the size of the core. Naturally, the larger is the core, the smaller is the impact of shell encapsulation. Further, to develop a library of optimized shells is a feasible task because they can generally be reused across many different core modules. In fact, the control logic of the shell remains the same regardless of the internal complexity of the core and the number of shell queues only depends on the core I/O interface. This important advantage of the proposed latency-insensitive protocol not only simplifies the design of the shell but it also guarantees its broad applicability: as long as the core module is a stallable sequential circuit it is not necessary to know its internal structure or behavior. In other words, latency-insensitive design can take advantage of any stallable *"black-box"* IP core.

[8]The problem of sizing the shell queues is discussed in Chapter 5, where the benefits of having queues of length two for implementations based on back-pressure are described in detail.

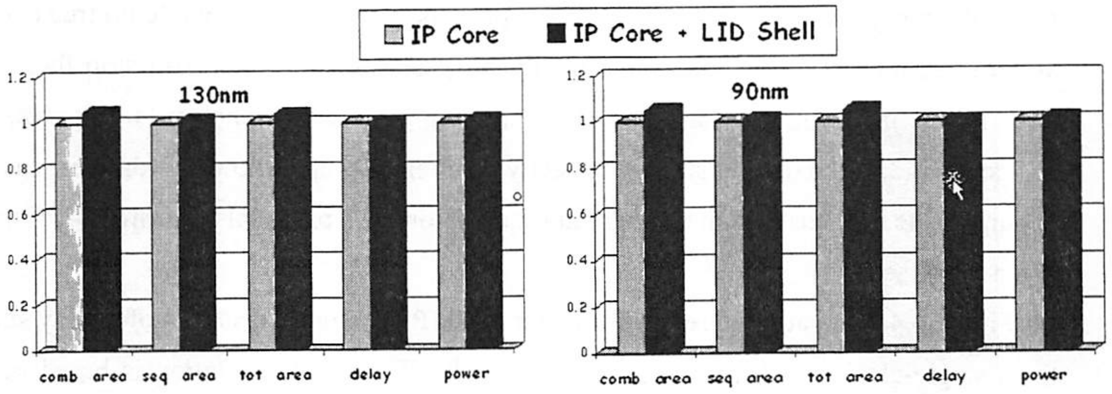

Figure 4.3: Shell encapsulation: experimental results with a third-party DCT core.

4.2.3 Relay Stations

In the context of hardware design, a relay station is a patient process interacting with two channels c_i and c_o such that if s_i and s_o are the signals associated to the channels and $I(l,k,s_i), l \leq k$ denotes the sequence of true packets of s_i between the l-th and the k-th clock cycle, then s_i and s_o are latency equivalent and for all k:

$$I(1,(k-1),s_i) - I(1,k,s_o) \geq 0 \tag{4.1}$$

$$I(1,k,s_i) - I(1,(k-1),s_o) \leq 2 \tag{4.2}$$

The following is an example of relay station behavior, where τ may denote either a void flag or a stop flag and ι_i denotes a generic true packet:

$$s_i = \iota_1\,\iota_2\,\iota_3\,\tau\,\tau\,\iota_4\,\iota_5\,\iota_6\,\tau\,\tau\,\tau\,\iota_7\,\tau\,\iota_8\,\iota_9\,\iota_{10}\,\ldots$$
$$s_o = \tau\,\iota_1\,\iota_2\,\iota_3\,\tau\,\tau\,\iota_4\,\tau\,\tau\,\tau\,\iota_5\,\iota_6\,\iota_7\,\tau\,\iota_8\,\iota_9\,\iota_{10}\,\ldots$$

No further specification is given on the signals s_i and s_o, (for instance to say that s_i is the input and s_o is the output). The definition of relay station involves simply a set of relations, i.e. a protocol, between s_i and s_o without any implementation detail. Still, it is clear that each true packet received on channel c_i is later emitted on c_o, while the presence of a void packet on c_o may translate into the emission of a void packet on c_i in a later cycle. In fact, any true packet takes at least one clock cycle to pass through a relay station (minimum forward latency = 1), no more than two true packets can arrive on c_i while no true packets are emitted on c_o (internal storage capacity = 2), and, finally, one extra stop flag on c_o will "move" into c_i in at least one cycle (minimum backward latency = 1). As explained in Section 3.3.4, the double storage capacity of a relay station allows, in the best case, to communicate with maximum throughput (equal to one). A practical confirmation of this is given in the sequel.

Figure 4.4 illustrates the register-transfer level (RTL) implementation of a relay station for a single-clock synchronous integrated circuit. This implementation is based on the following specification, which refines the abstract definition given above.

At each clock cycle t a relay station takes a packet $packetIn^t$ and a stop flag $stopIn^t$ as inputs and it emits a packet $packetOut^{t+1}$ and a stop flag $stopOut^{t+1}$ as outputs. Packet $packetIn^t$ contains the payload $dataIn^t$ and the void flag $voidIn^t$. Packet $packetOut^{t+1}$ has the same structure. The value of $stopOut^{t+1}$ is always equal to the value of $stopIn^t$, unless the relay station is in the *Processing* state and $voidIn^t = 1$: in this case $stopOut^{t+1}$ is set equal to 0. The value of $packetOut^{t+1}$ depends on $packetIn^t$ as well as the current internal state of the relay station, which, at each clock cycle, can be in one of four possible states. A finite state machine (FSM) capturing the control logic of the relay station of Figure 4.4 is described in Figure 4.5. Basically, the relay station switches between two main states: *Processing*, when it is traversed by a flow of true packets (and $packetOut^{t+1}$ is continuously set equal to $packetIn^t$) and *Stalling*, when the communication is interrupted and both registers in the relay station are occupied by true packets (while the up-link module keeps the value of $packetIn^t$ constant). The transition between these two states is governed by switching through states *WriteAux* and *ReadAux*, which make sure that the auxiliary register is properly controlled in order to avoid losing a packet while the stop flag takes a cycle to propagate backwards through the relay station. Specifically, if $stopIn^t = 1$ and $voidIn^t = 0$ the relay station enters state *writeAux*, where it only remains for one cycle. From this state it goes back to *Processing* without actually writing the auxiliary register if $stopIn^t = 0$ (i.e. it has stayed high for only a cycle). Instead, if $stopIn^t = 1$ then it does write the value of $packetIn^t$ into the auxiliary register and it goes into the *Stalling* state. The relay station leaves the *Stalling* state to go into the *ReadAux* only when $stopIn^t = 0$. This means that the down-link module is reading the data stored in the main register at this clock cycle and

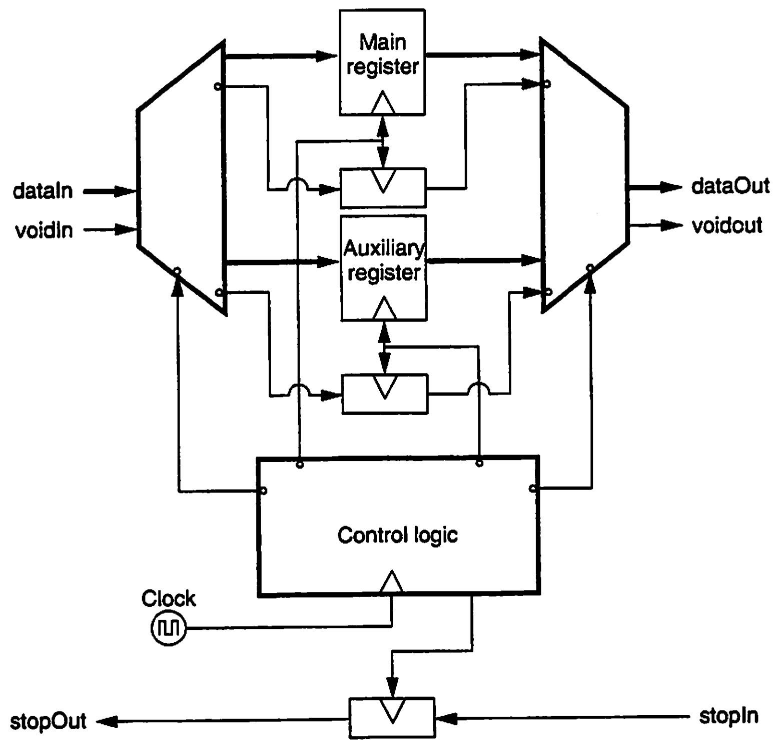

Figure 4.4: Hardware implementation of a relay station (block diagram).

will read (through $packetOut^{t+1}$) the data stored in the auxiliary register at the next cycle. Hence, as for state *writeAux*, the relay station remains only one cycle in state *ReadAux*. From this state it goes into processing if $stopIn^t = 0$. Instead, if $stopIn^t = 1$ it goes back into *writeAux*.

Example Figure 4.6 illustrates two modules, *Fetch Unit* and *Instruction Cache*, which communicate using two channels *Address Channel* and *Data Channel*. Both channels have been partitioned into four segments by the insertion of three relay stations. As a consequence, the lower bound on the latency of each channel has become four clock cycles. Figure 4.7 shows a snapshot of the waveforms obtained by simulating a VERILOG RTL description of the *Address Channel*: here, the source module is the *Fetch Unit* producing a sequence of addresses for a *Memory Block* which represents the sink module. The addresses are reported as hexadecimal numbers. The nominal clock of the system has period

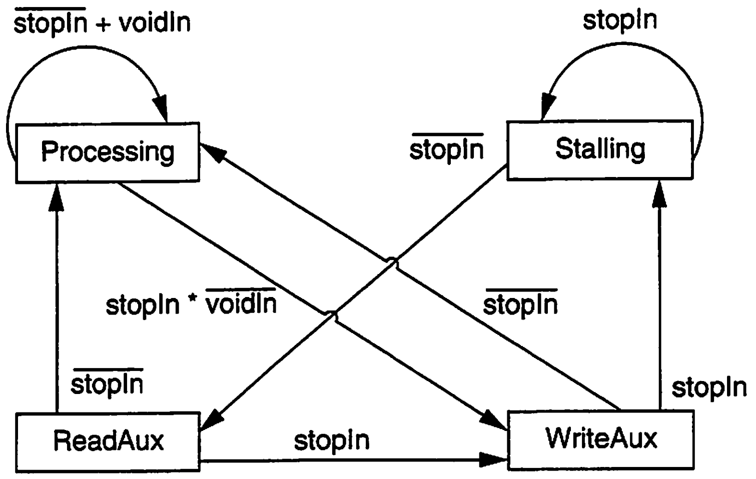
$\overline{\text{stopIn}}$ + voidIn
stopIn
Processing
Stalling
$\overline{\text{stopIn}}$
stopIn * $\overline{\text{voidIn}}$
$\overline{\text{stopIn}}$
$\overline{\text{stopIn}}$
stopIn
ReadAux
stopIn
WriteAux

Figure 4.5: Hardware implementation of a relay station (control logic FSM).

ψ equal to $10ns$. Going from top to bottom, the eight waveforms correspond respectively to the following signals of Figure 4.6: *R*2.*packetOut*, *R*2.*stopIn*, *R*1.*packetOut*, *R*1.*stopIn*, *R*0.*packetOut*, *R*0.*stopIn*, *FU*.*packetOut*, *FU*.*stopIn*.

At time $t = 75ns$ the sink module sets *R*2.*stopIn* equal to one and keeps it equal to one for three clock cycles. As a consequence, *R*2 stalls two cycles as it maintains *R*2.*packetOut* = $h'44$ for the next three cycles while storing *R*1.*packetOut* = $h'45$ on a auxiliary set of registers. In the meantime, the stop signal is propagated to *R*1.*stopIn*. When, after three clock cycles, at time $t = 105ns$, the sink module can finally receive *R*2.*packetOut* = $h'44$, it resets *R*2.*stopIn* such that at the following clock cycle *R*2 may set *R*2.*packetOut* = $h'45$. In the meantime, the three consecutive high values of the stop signal propagate back through the channel, provoking a stall of two cycles for each station while guaranteeing that no packets are lost. An important characteristic of this implementation of the protocol is that when a *stopIn* signal is kept high for only one cycle, the relay station does not really stall: Figure 4.7 illustrates this fact for the sequence of clock cycles starting at $t = 165ns$. This is simply a positive by-product of the fact that the storing capacity of a relay station is double. Recall that the primary reason for this double capacity is the need to avoid losing data during the necessary one cycle it takes to propagate the stop signal. ■

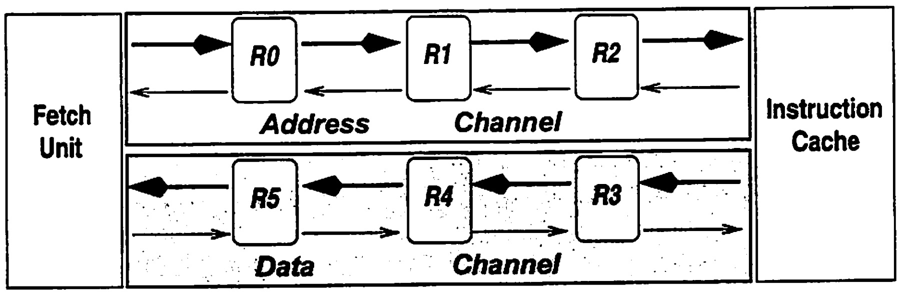

Figure 4.6: Channels between *Fetch Unit* and *Instruction Cache*

Besides providing a systematic method to perform wire pipelining, the insertion of relay stations on a channel creates a sort of *distributed queue*. Further, the control logic of the queue is also distributed as it is implemented by the back-pressure mechanism, which is inherently modular. Due to the increasingly distributed nature of SOCs (as discussed in Section 1.1.3), distributed queues seems to represent a design solution more promising than having long, centralized communication queues located next to each IP core module. This observation is also supported by the fact, discussed in Chapter 5, that sizing the shell input-queues with a length equal exactly to two provides an optimal implementation of a latency-insensitive protocols with back-pressure.

4.3 Impact on System Performance

In this section I discuss how the insertion of relay stations may affect the system performance and discuss some techniques that can minimize the decrease in communication throughput in DSM design.

4.3.1 Nominal versus Effective Clock Frequency

Latency-insensitive design makes the system functionality robust with respect to arbitrary variations of the interconnect latency among the core modules. But, to which extent is the system performance also made robust? Meeting the time constraints imposed by the target nominal clock ψ is certainly a most important performance metric, but it would be

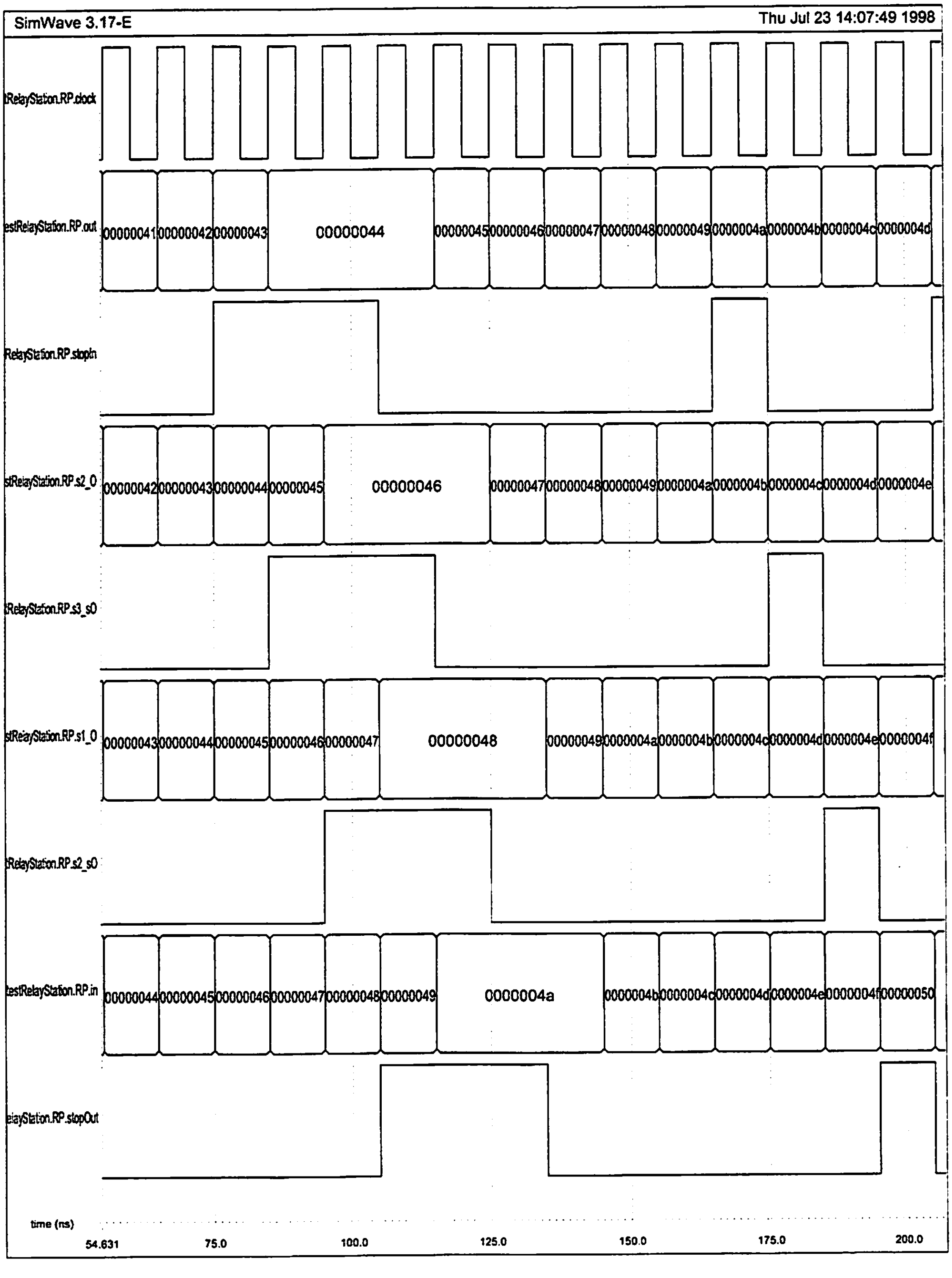

Figure 4.7: Waveforms on *Address Channel*.

fallacious to claim that the final SOC runs with frequency $\phi = \frac{1}{\psi}$ if, for instance, at every other cycle it produces a void packet. It would then be more accurate to say that the system runs with an *effective* frequency that is only half of the *nominal* frequency. Hence, the final design implementation would be correct but it would run relatively quite slowly. This thought raises the question that instead of inserting relay stations it would suffice to slow down the nominal clock in the first place.

In order to correctly evaluate the performance of a latency-insensitive system it is necessary to check how frequently it produces void packages at its output ports. Accordingly, I define the *throughput* $\vartheta(S)$ of a latency-insensitive system S as the amount of true packets produced by S in a given time interval. This of course corresponds to the ratio of true packets over the sum of true packets plus void packets (as observed at the system outputs during such interval) and it is a number between zero and one. It follows that, given a nominal clock period ψ, the *effective frequency* of a latency-insensitive system S is

$$\phi_{eff}(S) = \phi \cdot \vartheta(S)$$

The next question is where do void packets come from. A latency-insensitive system S may receive void packets at its primary inputs from the environment as well as generate them itself. In the first case, obviously, the environment reduces the throughput of S. The second case is more interesting from a design perspective because it sets a limit on the maximum throughput that S can sustain regardless of the environment in which it operates.

A properly designed shell emits void packets (stalling events) on its output channels only as a result of having previously received one or more void packets on its input channels. In fact, when the system starts-up, each core, being a sequential process, has is output registers initialized with a true packet (an informative event) [9]. Then, according to the AND-causality semantics that controls the shell operations, incoming void packets force the shell to stall the core and produce, at the next clock cycle, outgoing void packets. Therefore, the shell does not really generate new void packets, but it only re-transmits those that it receives. Instead, the generation of void packets inside system S is due to the

[9]This follows directly from the fact that, as discussed in Chapter 3, a latency-insensitive system is derived from a correct strict system where every process is a strict process and no stalling events are present.

presence of relay stations. More precisely, each relay station introduces one void packet in the system. The void packet corresponds to the initialization value for the storage element of the relay station. Since a relay station is a "design correction" that is extraneous to the original system specification, its initialization value must remain transparent to the cores (while visible to other relay stations and shells) in order to make sure that it does not corrupt their internal state. In other words, the simplest way to insert additional stateful repeaters into a sequential system without changing the internal logic of its components nor jeopardizing the correctness of its overall functional behavior is to make sure that such repeaters are initialized with values that will not get processed by the components that receive them. With a latency-insensitive protocol this can be done systematically as relay stations introduce stalling events and shells make sure that the cores do not see them [10].

Example Figure 4.8 illustrates a simple latency-insensitive system with three shell/core pairs P_L, Q_L, R_L and one relay station between Q_L and R_L. Table 4.1 reports the behavior of this system. When the system starts up, a relay station outputs its initialization value, a stalling (τ) event, while each core outputs its initialization value, an informative event. As the behavior evolves, new stalling events are generated whenever a core is stalled by its shell. In fact, since the system is cyclic, stalling events occur periodically on each signal at the rate of $1/4$. ■

Techniques to analyze and optimize the performance of a latency-insensitive system are presented in Chapter 5 and Chapter 6, respectively.

4.3.2 Preserving Communication Throughput in DSM Design

The effectiveness of the latency-insensitive design methodology is strongly related to the ability of maintaining a sufficient communication throughput in the presence of increased channel latencies. However, in the case of integrated circuit design this is just one instance of a more general problem that has to be faced while working with nanometer technologies. In fact, since on-chip communication was not an issue with previous process technologies, the vast majority of chips produced over the past two decades are based on ar-

[10]The systematic nature of this transformation is very important for the reasons discussed in Section 2.2.4.

	timestamp	:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
	w_l	:	1	0	τ	1	0	1	τ	0	...
P_l :	y_l	:	0	2	2	τ	6	6	10	τ	...
	z_l	:	0	τ	0	4	0	τ	8	4	...
	w_l	:	1	0	τ	1	0	1	τ	0	...
Q_l :	x_{l_1}	:	1	3	5	τ	7	9	11	τ	...
	y_l	:	0	2	2	τ	6	6	10	τ	...
	w_l	:	1	0	τ	1	0	1	τ	0	...
R_l :	x_{l_2}	:	τ	1	3	5	τ	7	9	11	...
	z_l	:	0	τ	0	4	0	τ	8	4	...

Table 4.1: The periodic behavior of the latency-insensitive system of Figure 4.8.

chitectural models relying on low-latency communication to shared global resources. The advantage of such models is that they provide the most uniform computational framework and the best utilization of the functional units. However, their focus on function rather than on communication is now seen as the fundamental conceptual roadblock to be overcome in nanometer design [112]. Due to the inherent separation of communication and computation, the latency-insensitive design methodology for SOC represents a promising approach in this new environment.

Inserting extra latency stages on a cyclic pipeline does not necessary translate into a performance degradation. For instance, the case of the Alpha 21264 microprocessor—where the integer unit is partitioned into two modules communicating with a latency of an additional clock cycle [89] and the pipeline presents an additional stage just to drive date from the outputs of the primary data cache back to the processor core [129]—shows how it is possible to pipeline long wires, thereby increasing their latency, while still offering high computational bandwidth. Similarly, the presence of the so-called *drive stages* in the INTEL® hyper-pipelined NETBURST® micro-architecture [91, 111, 201] indicates that for high-end designs, such as the PENTIUM®4 microprocessor, the insertion of extra stages dedicated exclusively to handle wire delays may be the result of a precise engineering

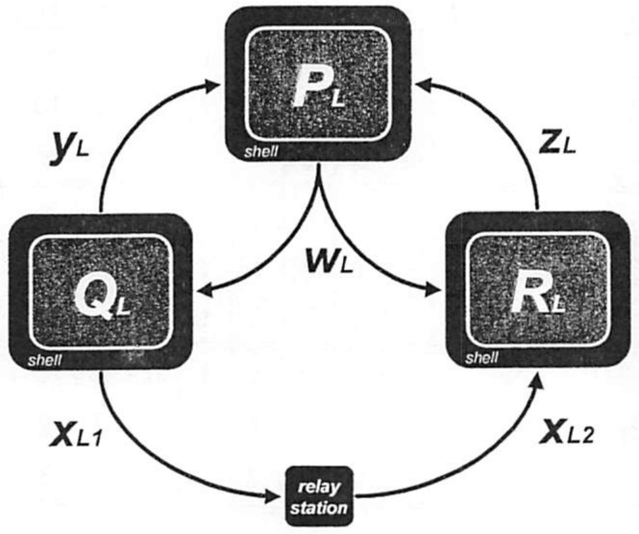

Figure 4.8: A simple cyclic latency-insensitive system.

decision [11]. Accordingly, the proposed approach represents a formal framework for the definition of design methodologies that allow an efficient analysis and exploitation of the latency/throughput trade-offs at any stage of the design flow.

Furthermore, the latency-insensitive approach can be extended to incorporate other techniques also aimed at minimizing the impact of large variations in channel latency on the performance of the system. For instance, a simple technique consists of ensuring that the design specification contains some *slack* in the form of uncalled-for pipeline delays. If there is sufficient slack latency around every cyclic path in the design, after the final layout is derived the latency-insensitive protocol allows us to re-distribute this slack in a completely transparent manner in order to cover an increase in wire latency without any changes to the design or the layout of the modules [12]. The overall design becomes more robust in the sense that it is less likely that some modules would have to be modified in

[11] A drive stage is used purely to move signal across the chip without performing any computation and, therefore, can be seen as a physical implementation of a relay station.

[12] This idea is discussed in detail in Chapter 6 in the context of a more general discussion on how to optimize latency-insensitive systems by balancing communication and computation.

order to recover the performance lost due to unexpectedly large wire delays.

There will sometimes be cases where cyclic paths of low latency affecting the overall system throughput are inevitable. In such cases, techniques typically used for microprocessor design such as *speculative* [153] and *out-of-order execution* [108] may be embedded within a latency-insensitive protocol. For instance, when a particular data item on the critical path, is not yet computed, it is sometimes possible to predict its likely value and allow the computation to proceed. The predicted value can be later "retracted" if it proves to be incorrect. Such techniques can dramatically increase the overall system performance. They are currently adopted only in high-end microprocessor design because they are error-prone and quite difficult to implement, but their use is destined to grow as wire latencies continue to increase. They could be rigorously built into a latency-insensitive protocol that would allow speculation to break the tight dependency cycles, provided the designer can identify an appropriate prediction strategy. Once they become part of the protocol, these techniques would be isolated from the functional specification and would enter the design picture in a correct-by-construction manner that would forbid the introduction of a design error and prevent previous simulation work from being invalidated.

4.4 Case Study: the PDLX Microprocessor

To experiment with the proposed methodology, I performed a *latency-insensitive design* of PDLX, a microprocessor with out-of-order and speculative execution. In the present section, I first summarize the architectural specification of PDLX, and, then, discuss the latency-insensitive design as well as the experimental results.

4.4.1 PDLX Architecture and Instruction Flow

The instruction set of PDLX is the same as the one of the DLX microprocessor, described in [108]. The PDLX architecture is based on an extended version of the *Tomasulo's Algorithm* [235], which combines traditional dynamic scheduling with hardware-based speculative execution [108]. As a consequence, the data-path of PDLX is similar to the one of some of the most advanced microprocessors available on the market today [89, 129].

The PDLX architecture is conceptually illustrated by the block diagram of Figure 4.9. At the center of the PDLX pipeline lies a set of execution units (gray shaded) which operate in parallel. Schematically the PDLX behavior can be summarized as follows: the *branch processing unit* sends the next value of the *program counter (PC)* to the *fetch unit*, which fetches the corresponding instruction from the *instruction cache* and passes it to the *decode unit*. Once instruction decoding is completed, the result arrives to the *dispatch unit* which interacts with the *reservation stations (RS)* of the different processing units as well as with the *completion unit* and the *system register unit*. Instructions are fetched, decoded and dispatched sequentially following the order of the program which is executed. Once a new decoded instruction I arrives, the dispatch unit starts by assigning it to one of the functional categories (integer arithmetic/logic, floating-point, load, store, branch, ...) and, then checks whether the following two conditions are verified:

1. there is one entry available in the reorder buffer within the completion unit;
2. there is an available reservation station at the head of a processing unit matching the functional category to which the instruction belongs;

If one of these conditions is not satisfied the dispatch unit stalls, otherwise instruction I is dispatched. This means that the instruction is labeled with a *tag* t_I identifying the reorder buffer entry which has been assigned to it and which at the end of the execution will contain the result. Then, operands and *opcode* of instruction I are loaded into the selected reservation station to start the execution. The execution starts immediately only if the values of the operands that have been read from the *System Register Unit* are "currently correct", in the sense that no other instruction I' (previously dispatched and still not completed) is destined to change the value of one of these operands. If this is indeed the case, then for each operand whose value is not yet available the dispatch unit writes on the corresponding entry in the reservation station the tag $t_{I'}$. Tag $t_{I'}$ identifies the reorder buffer entry previously assigned to instruction I' on which the operand depends. The execution of instruction I is procrastinated until all correct values of its operands arrive at the reservation station. Different instructions may take a different number of clock cycles to be executed not only due to this procrastination, but also because their executions may present different latencies. When the execution of an instruction is completed, the corresponding processing unit

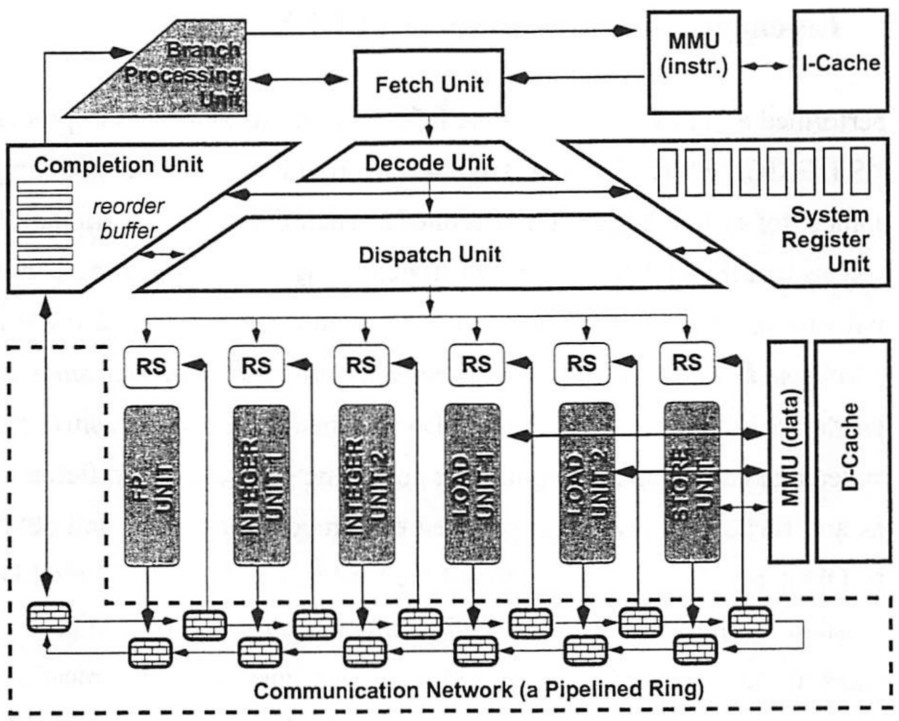

Figure 4.9: PDLX microprocessor block diagram (conceptual view).

broadcasts the result to the reorder buffer entry and to any reservation station which has been waiting for it.

The completion unit tracks each instruction from dispatch through execution, retires it by removing the result from the bottom entry of the reorder buffer, and commits it to update the system registers. In-order completion guarantees that the system is in the correct state should it be necessary to recover from a mispredicted branch or any exception. In the presence of a conditional branch whose condition cannot be resolved immediately, the branch processing unit predicts the "branch target address" and instruction fetching, dispatching, and execution continue from the predicted path [108]. However, these instructions cannot commit and write back results into the system register until the prediction has been resolved, i.e. determined to be correct. In the case when a prediction is proved wrong, the instruction from the wrong path are flushed from the data-path and the execution resumes from the correct path.

4.4.2 Latency-Insensitive Design of PDLX

I performed a high-level cycle-accurate design of the PDLX microprocessor by using BONES DESIGNER [208], a CAD tool which provides a powerful modeling and analysis environment for system design. I first defined a synchronous specification of the PDLX and designed each of the PDLX modules illustrated in Figure 4.9, assuming only the following informal rule in order to make the process stallable: *at each clock cycle the execution process of a module can always be frozen without affecting its internal state*. Independently from the design of the PDLX modules, I also specified a latency-insensitive protocol library of parameterized components to guide the automatic generation of different kinds of relay stations and shells. Finally, I encapsulated each module in a shell and obtained the final system. Obviously, this decomposition of the hardware implementing the PDLX is not the only possible, let alone the best one. Still, while reasonably simple, it presents interesting challenges to the design of the proposed latency-insensitive communication architecture. In particular, modules such as the fetch unit and the dispatch unit merge channels coming from separate sources, and which are likely to have different latencies. Further, each time the predicted value of a conditional branch is confirmed a *feedback path* is activated from the system register through the completion unit to the branch processing unit.

Most PDLX modules were specified based on the assumption that they communicate by means of point-to-point channels with arbitrary latencies. However, due to the peculiar structure of a microprocessor such as PDLX, and, in particular, to the parallel organization of its execution units, I adopted a different type of communication structure to connect the several relay stations, the execution units, and the reorder buffer: a *pipelined ring*. A ring, like a bus, inherently supports broadcast-based communication: the sender places a packet on the ring and the other modules inspect (*snoop*) it as it goes by and decide if it is relevant to them [65]. In the case of PDLX, the snooping mechanism is obviously based on identifying the tags associated to the entries of the reorder buffer, as described in Section 4.4.1. In general, it is more complex to keep sequential consistency on a ring than on a bus since multiple packets may traverse it simultaneously. However, in PDLX it constitutes a lesser problem since the completion unit guarantees in-order instruction commitment together with the correct serial progress of the state of the system. And while the linear point-to-

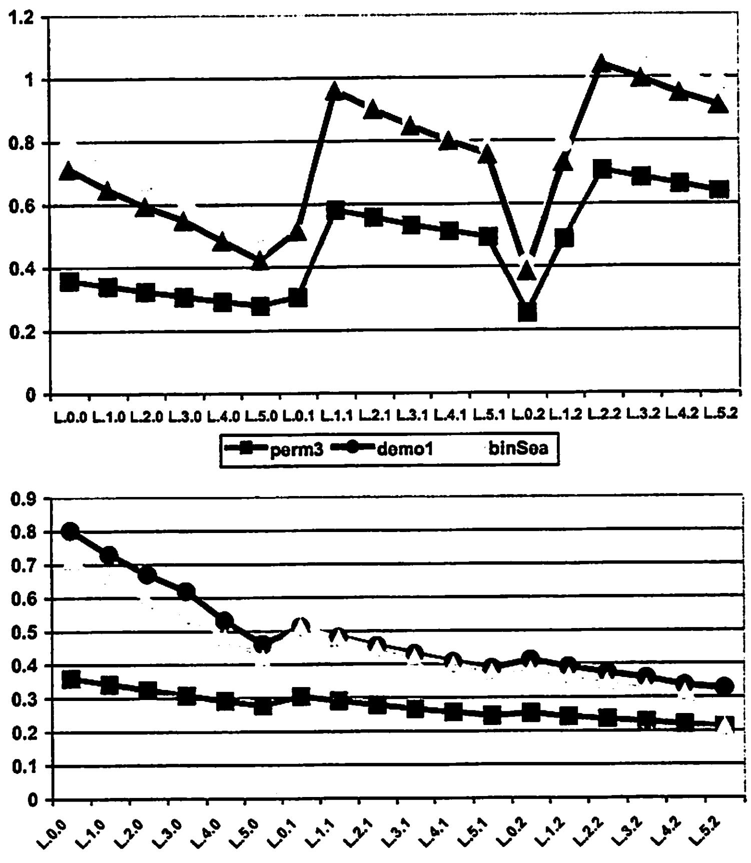

Figure 4.10: PDLX performance: throughput (bottom) and effective throughput (above)

point nature of a ring allows aggressive pipelining and potentially very high clock rates, it has the disadvantage of high communication latency which grows linearly with the number of nodes on the ring. Overall a ring represents a design solution in high agreement with the latency-insensitive design methodology: its modules are by definition latency-insensitive and pipelining can be extensively applied.

4.4.3 Experimental Results with PDLX

To test this design, I used some simple numerical *C* programs (permutations, binary search, ...) and generated the corresponding DLX assembler code by running DLXCC, a publicly available DLX compiler [218]. The assembler was loaded into the PDLX's instruction cache and executed while logging every read/write access to the data cache. Then, I compared the "log file" with the one obtained executing the same assembler code on the DLX simulator DLXSIM to verify that the functional behavior was indeed the same.

For each program execution, I computed the total number of clock cycles N necessary to complete the execution of the assembler code: this number is equal to $I+S+P$, where I is the number of instruction which have been issued, S is the number of cycles lost due to a stall within the execution unit, and P is the number of cycles lost due to pipeline latency. Since PDLX is a single-issue multiprocessor, the instruction throughput $T = I/N$ is a quantity less than or equal to one. This quantity can be multiplied by the system clock frequency to obtain the *effective instruction throughput* $ET = (I/N) * \phi$, which makes it possible to compare the execution of the same assembler code on different PDLX implementations running at different speeds. Figure 4.10 illustrates the results obtained running three different assembler programs: the y-axis represent the instruction throughput on the bottom chart and the effective instruction throughput on the top chart. In both charts each discrete point on the x-axis corresponds to a different PDLX implementation with its own fixed amount of latency specified communication channels.

For this experiment, I focused on two specific channels on Figure 4.9: channel C_a between the *Instruction Memory Management Unit* and the *Instruction Cache (I-Cache)* and channel C_b between the *Data Memory Management Unit* and the *Data Cache (D-Cache)*. The latencies of the two channels were changed as follows: going from left to right on the x-axis, each of the 18 data-points represents an implementation case and is labeled as L_a_b, where a and b denote the amounts by which the latencies of channels C_a and C_b have been increased. In particular, a varies from 0 to 5 and b from 0 to 2. As expected, the bottom chart confirms that the larger is the latency between the two caches and the rest of the system, the higher is the throughput degradation. It is also clear that for this PDLX implementation the impact of increasing the D-Cache latency by 1 unit while leaving untouched the I-Cache latency (data-point L_0_1) is more or less equivalent to increasing the I-Cache latency by 4 units while leaving untouched the D-Cache latency (data-point L_4_0).

The data illustrated in the chart of Figure 4.10 have been obtained based on the assumption that the wires grouped in channels C_a and C_b represent the critical path of the overall PDLX design and that, after segmenting them (by inserting relay stations), it is possible to raise the clock frequency appropriately. Specifically, for each implementation, the system has a nominal clock with period $\psi = \min\{a, b\} + 1$. One could argue that the assumption is too coarse, since, for instance, it is unlikely that all the other modules in the design are able to work correctly after doubling the clock. However, the main point of the experiment is that, within the present methodology, one may perform an early exploration of the latency/throughput trade-offs to guide architectural choices based on a rough estimation of the channel latencies and proceed to refine these choices throughout the different stages of the design flow in order to accommodate various implementation constraints, while relying always on the property of the latency-insensitive communication protocol [13] It is important to emphasize that the above implementations are all functionally equivalent *by construction*, being obtained simply by changing the number of relay stations on the channels without any need of re-designing the PDLX modules. Also, the insertion of relay stations to "fix" those channels whose latencies are higher than the desired clock cycle can be made at late stages in the design process, after detailed information has been extracted from the physical layout. While this operation is performed it is easy to keep track of the throughput variations.

4.5 Related Work

In this section I discuss related work along three main lines. I summarize first some of the approaches that have been proposed to address the increasing impact of interconnect delays in DSM design. Secondly, I discuss recent papers that advocate the combination of wire pipelining and retiming as the solution to the global interconnect problem, while highlighting the key difference between these approaches and latency-insensitive design. Thirdly, I briefly comment on some circuit-level works that are closely related to latency-insensitive design.

[13] In Chapter 6, I discuss in detail how latency-insensitive design supports system-level design exploration.

4.5.1 DSM Design Methodologies

As discussed in Section 2.2.1, the arrival of nanometer technologies has generated an intense debate on the magnitude of the "wire delay problem". Although there has been a fair amount of disagreement between the various studies proposing models to predict the interconnect delay [74, 112, 113, 163, 167, 223, 224, 225, 226, 206, 207], researchers tend to agree with each other on the dominant role played by global wires and on the likely aggravation of the timing-closure problem. In Section 2.2.3 I explained how this problem leads designers to many costly iterations during the design process before they can arrive at the final chip layout. The main cause is that logic synthesis and physical design are performed separately and the former uses statistical delay models that badly estimate the post-layout wire load capacitance [58, 68, 126, 188]. Specifically, in the current standard-cell design methodology, logic synthesis is performed using delay estimates for library modules that are parameterized to account for loading factors and transition (or slew) rates. As the delays of long wires become larger relative to gate delays, these estimates become increasingly sensitive to layout. Furthermore, process variations, cross-talk and power-supply drop variations, which were treated as second-order "physical effects" while designing with previous technologies, can no more be underestimated in the DSM realm. In fact, their combined action may substantially affect the interconnect delays in a way that is difficult to predict and that can vary across chip regions and periods of chip operation. Advances on interconnect optimization techniques (such as interconnect topology optimization, optimal buffer insertion and sizing, and optimal wire-sizing) can help to reduce interconnect delays significantly [59], but they are not able to reverse the trend of the growing gap between device and interconnect performance [58]. In summary, it is not surprising that most of the approaches that have been proposed in the literature to address the timing-closure problem call for a tighter interaction between synthesis and physical design.

Otten and Brayton proposed to account for layout effects by performing floor-planning and wire-planning on register-transfer level (RTL) descriptions [192]. Such an approach requires extreme precaution in deriving constraints for the synthesis tools because any wire whose delay approaches a single clock cycle may cause a failure to meet the timing constraints. Furthermore, as explained in Section 2.2.3, logic synthesis is inherently unstable, i.e. small variations in the HDL RTL specification may lead to major variations in the produced standard-cell netlist and, consequently, in the final layout.

Gosti *et al.* have proposed a synthesis-driven methodology that optimizes for interconnect delay rather than gate delay during logic synthesis [92, 93]. Unfortunately, their approach produces a large amount of logic duplication with consequent expensive area overheads.

Floorplanning, technology mapping and gate placement are combined in [204], where, after placement has been completed, the critical paths are reduced one at a time in order to meet the timing requirements. Because, fixing one critical path often results in generating new ones, this approach is unable to solve *by construction* the timing-closure problem.

A series of layout-driven approaches have offered to fix the layout by extracting accurate physical information which is then used to guide different types of logic optimization, such as gate-resizing [114], fanout optimization [125], buffer insertion [205] and logic re-synthesis [154].

These approaches, however, constitute remedies to the effects of poor estimations made during logic synthesis but do not seem able to scale well with the continuous shrinking of process geometries.

4.5.2 Wire Buffering and Wire Pipelining

The combination of wire buffering and wire pipelining, recently advocated by several researchers, is an important issue and deserves a separate discussion.

First, it should be said that wire buffering is a well-known technique that designers have used for many years to optimize on-chip wire transmission [7, 70, 193]. Also from a CAD perspective, methods for combining routing with the insertion of buffers have been proposed for many years before the advent of nanometer technology processes, e.g. see the classic paper by Van Ginneken [90]. Given a metal line of a certain length, *optimal wire buffering* corresponds to the distribution of an optimal number of optimally sized transistor buffers on the line to minimize the transmission delay. As illustrated in Figure 4.11, with optimal wire buffering the delay of a metal line grows linearly with its length instead of

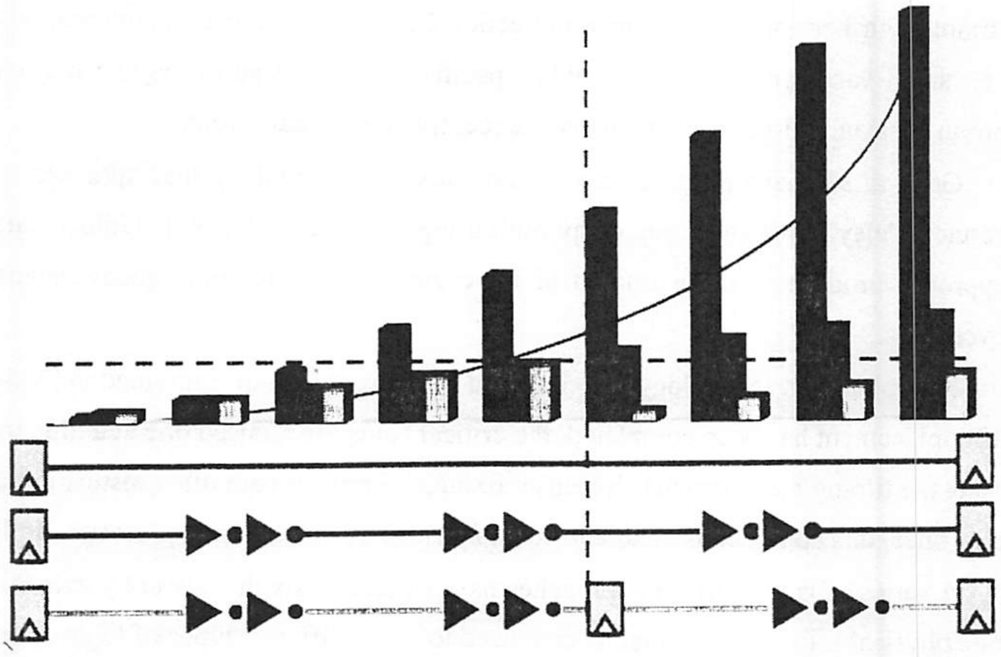

Figure 4.11: Wire buffering versus wire pipelining.

quadratically. Naturally this is a formidable gain, which, according to a recent study by Ho *et. al* [113], will continue to have a major impact on IC design: a careful use of simple buffering circuits (a single inverter or two back-to-back inverters) can reduce the degradation of global wire delay with respect to gate delay from a factor of $2000x$ to a factor of $40x$ (over nine generations) while for local wire delay the reduction is from a factor of $10x$ to a factor of $2-3x$.

Still, optimal wire buffering carries precise limitations that are also illustrated in Figure 4.11: if the the delay of the optimally buffered metal line is still larger than the nominal clock period, it is necessary to go beyond wire buffering and break the long wire into smaller segments by inserting memory elements like latches or flip-flops (*wire pipelining*). This approach trades-off fixing a wire exception with increasing its latency by one or more clock cycles and allows driving long wires with the same clock signal used to control short wires and logic gates. An alternative to wire pipelining is to drive long wires with slower clocks, thus effectively implementing the chip as a *multi-clock* system where many isochronic regions exchange data at a speed slower than the one at which they operate. It is also possible to avoid using clock mechanisms to drive long wires altogether and instead to adopt *asynchronous* communication protocols to build GALS architectures [14].

The common point among all these options is that the chip is regarded as a *distributed system*, invalidating the main assumption of the synchronous design methodology [42]. This important fact is implicitly addressed in the case of multi-clock and GALS implementations, which do not rely on the synchronous hypothesis. Different is the case of those approaches that have been recently proposed in the literature combining wire buffering and wire pipelining in the context of a traditional design methodology. Cocchini has proposed algorithms for the concurrent insertion of buffer and flip-flops on latency-constrained interconnects [56]. However, to assume that the designers predefine latency-constraints for each wire at the micro-architectural level is a strong prerequisite that may limit considerably the applicability of the approach. Lu *et al.* have proposed methods to reserve spaces for buffers and flip-flops during floorplanning [155], but they seem to neglect the key difference between inserting combinational buffers and inserting latches or flip-flops. In fact, while the former are *stateless repeaters*, the latter are *stateful repeaters*, i.e. they are sequential elements that must be initialized and whose insertion modifies the latency (expressed in number of clock cycles) between the chip modules lying at the two extremes of the pipelined wire. In general, to initialize these elements and to interface them with the surrounding control logic, which has been derived assuming a different communication latency, a certain amount of careful redesign must be performed with negative consequences on design productivity [209].

Finally, it should be noticed that techniques combining global placement and retiming may help to avoid the need of wire pipelining only to the extent that the original design specification contains a sufficient number of latches that can be distributed along the interconnect paths [61]. As explained in Section 7.1.3, however, retiming carries as an intrinsic limitation the fact that the number of latches on any feedback loop in the design must remain constant.

[14]While originally conceived to facilitate wire pipelining and IP assembly in single-clock chips, latency-insensitive design can be used also as a formal framework to develop tools for the automatic deployment of synchronous designs on distributed architectures, like GALS [41].

4.5.3 Related Work in Integrated Circuits Design

Chelcea and Nowick have developed a library of robust circuit interfaces for mixed-timing integrated circuits, i.e. circuits working at different clock speeds and different timing models (synchronous, asynchronous) [49]. The library contains several low-latency, high-throughput, first-in first-out (FIFO) queues as well as two new *mixed-timing relay stations*. These circuits have been designed using a modular approach: a set of basic interfaces, both synchronous and asynchronous, has been defined; these interfaces then can be assembled to obtain a FIFO meeting the desired timing assumptions on both the senders' and receivers' end. Thus, the design of a mixed-timing FIFO is reduced to reusing and assembling a few pre-designed components. One of the important contributions of this work is precisely the novel design of relay stations for mixed asynchronous/synchronous interfaces. This is the first published work that addresses simultaneously the following challenges: handling mixed asynchronous/synchronous interfaces and coping with slow global interconnect.

At ASYNCH'02, Jacobson *et al.* presented an interesting paper on *synchronous interlocked pipelines* [118] which contains several commonalities with the latency-insensitive design methodology. The authors share some of the motivations (e.g., the dominance of interconnect delay in DSM design) although they do put a strong emphasis on the need to develop power-aware techniques that perform computation only on demand. More importantly, however, the underlying philosophy—the goal of finding *"a middle ground in techniques that can provide the benefits of asynchronous properties in a synchronous context"*—parallels the main thesis advocated in the present dissertation and that was first proposed at ICCAD'99 [34]. Jacobson *et al.*'s strategy towards this goal embodies some of the ideas that were presented in the context of latency-insensitive design [34, 37], including:

- applying stalling at a fine-grained level;
- using "clock-gating" to implement this fine grained stalling;
- performing pipelining stalling in the backward direction (i.e., *pipelined back-pressure*);
- using a valid/void signal to encode the presence/absence of informative events.

The main difference between the two approaches lies at the implementation level, particularly in the circuitry used to make the stages of the synchronous pipeline. A relay station requires the insertion of an "auxiliary register" in parallel to each "stage register". The role of this auxiliary register is to avoid losing data during the single cycle necessary for the back-pressure stalling signal to cross the stage. In [118], instead, there is no insertion of parallel extra registers and the loss of data is avoided by using every other register in series to achieve the same result (this is done on opposite phases of the clock in the 2-phase clocked pipeline and relying on the master/slave structure in the 1-phase clocked pipeline). As a consequence, Jacobson *et al.* report that the maximum occupancy of the pipeline, when no stalling has occurred, is of $N/2$ data items, where N is the number of (serial) stage registers. This is equivalent to the occupancy of an analogous pipeline with N relay stations, where, however, the flow throughput is 1 instead of 0.5.

Finally, in [21], Borgatti *et al.* have proposed a reconfigurable on-chip communication network that consists of a multi-context, programmable crossbar implemented using a matrix of modified Flash-EEPROM devices. On-chip interconnect networks that are modular, wide-bandwidth, and reconfigurable are of major interest for the design of large multiprocessors systems. However, programmable interconnects have strong intrinsic limitations in terms of signal propagation speed. To improve their speed, Borgatti *et al.* have combined the concept of *elastic interconnect*, recently proposed by Mizuno *et al.* [169], together with a newly-designed communication protocol. The protocol operates quite similarly to a latency-insensitive protocol as it guarantees lossless communication through the use of a *congestion signal*, which propagates in opposite direction with respect to the flow of data and control. In completing the custom design of the crossbar, they paid particular attention to the timing synchronization of the switches, thus avoiding the area overhead of a typical standard-cell implementation of a relay-station.

4.6 Concluding Remarks

I presented a *correct-by-construction* methodology for designing complex SOCs by assembling IP functional modules. The modules interact exchanging data on a communication architecture that is built according to a latency-insensitive protocol. The protocol guarantees a correct system behavior independently from the latencies of the communication channels. As a consequence, a robust implementation is achieved in a shorter time frame simplifying the assembly of pre-designed modules and reducing the number of iterations between logical and physical design. Specifically, the application of latency-insensitive design to integrated circuits provides two main advantages: (1) it facilitates the assembly of pre-designed components, that, as long as they are stallable, can be interfaced with the communication protocol without changing their internal structure, and (2) it enables the *a-posteriori* automatic pipelining of long wires through the insertion of relay stations. I gave an operational description for the main building blocks of a latency-insensitive communication architecture (channels, relay stations, and shells) and provided a reference hardware implementation based on the concept of back-pressure. As a case study, I reported on the latency-insensitive design of PDLX, an a microprocessor with out-of-order and speculative execution. Finally, I discussed qualitatively the possible impact of the proposed methodology on the system performance. A quantitative analysis is the subject of Chapter 5.

Chapter 5

Performance Analysis

In which tokens move around and everything is transient, but in tune.

ONE of the main advantages of latency-insensitive design is perhaps the simplicity of doing performance analysis. As shown in the sequel, a latency-insensitive system can be modeled using a particular kind of Petri nets called *marked graphs*. Albeit their simplicity, marked graphs allow us to capture the concurrent behavior of a distributed deterministic system, to verify whether it satisfies important properties like liveness and boundedness, and to compute statically its performance. In this chapter, after providing essential background material, I present a method to use marked graphs for building models of latency-insensitive system implementations. Specifically, I distinguish two cases: first I assume that shells have unlimited memory space to store interface signals (infinite queues) and then I analyze the case of finite queues with back-pressure. For both models I rely on previous results about the cycle time of a marked graph necessary to compute the maximum throughput at which the system implementation can consume/produce informative events. I conclude presenting a theoretical result valid under the assumption that the environment is willing to adapt itself to the system's throughput: a *physical* implementation based on back-pressure and finite queues with length equal to two is able to perform as well as a *virtual* implementation with infinite queues.

5.1 Petri Nets and Marked Graphs

This section provides some background material on Petri nets and marked graphs. A more complete presentation is offered in the journal paper by Murata [176] and the book by Peterson [186], which also contains a detail, albeit outdated, annotated bibliography.

5.1.1 Petri Nets

Petri nets are a model of computation introduced by C.A. Petri in 1962 [187]. They can be used to model various kinds of systems and are particularly effective to capture concurrency, parallelism, distributiveness, non-determinism, as well as various synchronization policies.

A Petri net consists of a particular type of directed weighted bipartite graph [62] together with an initial state called the initial marking. Being a bipartite graph, a Petri net has its vertices partitioned into two subsets, called respectively *places* and *transitions*, that are connected through directed arcs. Graphically places are drawn as circles and transitions as bars. A marking assigns to each place a nonnegative integer k corresponding to the number of *tokens* contained by the place.

Definition 5.1 *A* Petri net *is a 5-tuple,* $\mathcal{PN} = (P,T,F,W,M_0)$ *where* P *is a finite set of places,* T *is a finite set of transitions,* $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),* $W : F \rightarrow \mathbf{Z}^+$ *is a weight function,* $M_0 : P \rightarrow \mathbf{Z}^*$ *is the initial marking, and such that* $P \cap T = \emptyset \wedge P \cup T = \emptyset$.

Definition 5.2 *Given a Petri net* $\mathcal{PN} = (P,T,F,W,M_0)$ *a place* $p \in P$ *is* initially empty *when* $M_0(p) = 0$. *The set of initially empty places is denoted as* $P_0 = \{p \in P | M_0(p) = 0\}$.

Pre-sets and post-sets of a transition, or a place, of a Petri net are denoted as follows.

Definition 5.3 *Given a Petri net* $\mathcal{PN} = (P,T,F,W,M_0)$, *for all transitions* $t \in T$ *and all*

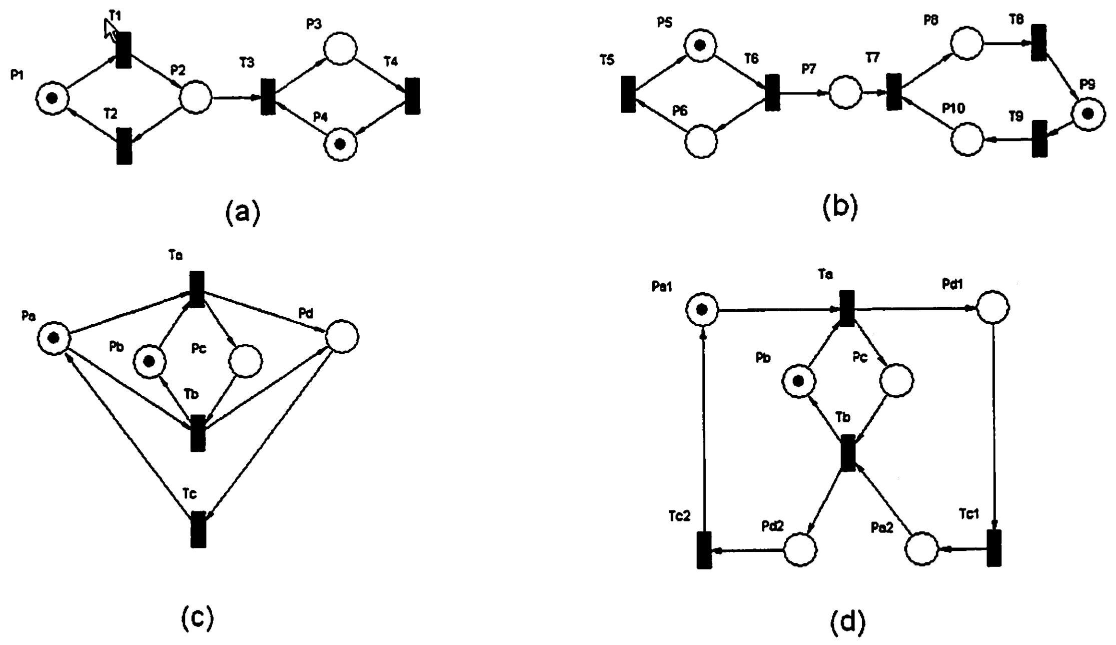

Figure 5.1: Petri net examples taken from [176, 194].

places $p \in P$,

$$\begin{aligned}
\bullet t &= \{p|(p,t) \in F\} && \textit{is the set of input places (pre-set) of } t \\
t\bullet &= \{p|(t,p) \in F\} && \textit{is the set of output places (post-set) of } t \\
\bullet p &= \{t|(t,p) \in F\} && \textit{is the set of input transitions (pre-set) of } p \\
p\bullet &= \{t|(p,t) \in F\} && \textit{is the set of output transitions (post-set) of } p
\end{aligned}$$

Example Figure 5.1 illustrates four examples of Petri nets taken from [176, 194] and drawn using VISUAL OBJECT NET++, an engineering toolset for regular and hybrid Petri nets developed by Drath [78]. In particular, Figure 5.1(a) represents a Petri net $\mathcal{PN} = (P, T, F, W, M_0)$ with a set of places $P = \{p_1, p_2, p_3, p_4\}$, a set of transitions $T = \{t_1, t_2, t_3, t_4\}$, a set of arcs $F = \{p_1 \rightarrow t_1, t_1 \rightarrow p_2, p_2 \rightarrow t_2, t_2 \rightarrow p_1, p_2 \rightarrow t_3, t_3 \rightarrow p_3, p_3 \rightarrow t_4, t_4 \rightarrow p_4, p_4 \rightarrow t_3\}$ and an initial marking $\{M_0(p_1) = 1, M_0(p_2) = 0, M_0(p_3) = 0, M_0(p_4) = 1\}$. Hence, $P_0 = \{p_2, p_3\}$. Also, for instance, $\bullet t_3 = \{p_2, p_4\}$ and $p_2\bullet = \{t_2, t_3\}$. ∎

In order to simulate the dynamic behavior of a system, a marking in a Petri net is changed according to the following *firing rules*:

1. a transition $t \in T$ is said to be *enabled* if each input place p of t is marked with at least $w(p,t)$ tokens, where $w(p,t)$ is the weight of the arc from p to t;

2. an enabled transition may or may not fire (depending on whether or not the corresponding event does occur);

3. a firing of an enabled transition t removes $w(p_i,t)$ tokens from each input place p_i of t, and adds $w(t,p_o)$ tokens to each output place p_o of t.

A transition without any input place is called a *source transition* and one without any output place is called a *sink transition.* A source transition is unconditionally enabled. The firing of a sink transition consumes tokens, but does not produce any.

A marking M is said to be *reachable* M_0 if there is a sequence of firings that transforms M_0 into M. The set of all possible markings reachable from M_0 is denoted as $R(M_0)$.

A Petri net is *ordinary* when all of its arc weights are equal to one, i.e. it has a weight function $W : F \rightarrow 1$. In the sequel I only consider ordinary Petri nets and, therefore, omit the adjective [1]. A Petri net is *K-bounded*, or simply *bounded* when for each place p there is an upper bound K on the number of tokens that can be contained by p for any marking reachable from M_0. A Petri net is *safe* when it is 1-bounded.

A Petri net is *reversible* when for each marking $M \in R(M_0)$, M_0 is reachable from M, i.e. it is always possible to get back to the initial marking. In many applications it is not necessary to get back to the initial marking M_0 as long as it is always possible to get back to a certain marking M_{hs}. This leads to a notion of *relaxed reversibility* that is based on the definition of a *home state* M_{hs} as any marking that is reachable from each marking $M \in R(M_{hs})$. This is the notion of reversibility that is used in this chapter.

A Petri net is *live* for the initial marking M_0 if, no matter what marking has been reached from M_0, it is possible to ultimately fire *any* transition of the net by progressing through some further firing sequence. A Petri net that is not live is *deadlocked.*

[1]Both ordinary and non-ordinary Petri nets have the same modeling power.

A Petri net is *persistent* when, for any two enabled transitions, the firing of one transition will not disable the other. In other words, for all reachable markings in a persistent Petri net, a transition t is disabled only by firing t.

Example The Petri net of Figure 5.1(a) is bounded, not live, and not reversible. The Petri net of Figure 5.1(b) is live, not reversible, and not bounded (tokens keep accumulating at place p_7). The Petri net of Figure 5.1(c) presents an interesting substructure referred to as *choice* (or *conflict* or *decision*) which allows the modeling of non-determinism. A choice occurs whenever a place has more than one outgoing arc. This is the case of place p_a that has both t_a and t_b as output transitions. However, an analysis of the global structure of this Petri net reveals that a choice actually never occurs, because when t_a is enabled t_b is not (and vice versa): in fact, this Petri net is persistent. ■

Definition 5.4 *A* directed path $\{p_0, t_1, p_1, t_2, \ldots, t_n, p_n\}$ *of a Petri net* $\mathcal{PN} = (P, T, F, W, M_0)$ *is a sequence of places and transitions such that for* $i \in [1, n]$, t_i *is an output transition of* p_{i-1} *and input transition for* p_i. *A path is* elementary *when no vertex is traversed more than once. A path is* simple *when no arc is traversed more than once. If* p_0 *and* p_n *coincide then the directed path is a* directed cycle. *A Petri net is* strongly connected *if every pair of places is contained in a directed cycle.*

Example The Petri nets of Figures 5.1(a,b) are not strongly connected. Instead they contain two and three strongly connected components [62] respectively. The Petri nets of Figures 5.1(c,d) are strongly connected. ■

While the concept of time is not explicitly given in the original definition of Petri nets, various *extended Petri net* models have been proposed to study the performance and the schedulability of dynamic systems and distributed systems [195, 242]. In these models delays are associated to transitions and/or places. For instance, an execution time $d(t)$ can be associated to each transition t to denote the number of time units necessary to complete the execution of t.

5.1.2 Marked Graphs

Marked graphs, also known as *decision-free* Petri nets, are a simple model for decision-free (or deterministic) concurrent systems [57]. Their simplicity makes them quite amenable to analysis. A Petri net is a marked graph when for each place in the net there is only one incoming arc and one outgoing arc. In other words, the tokens that transit through this place are generated only by one input transition and consumed only by one output transition. Hence, a marked-graph representation of a system with decisions is possible only when each decision can be embedded in a subsystem having only a single input and a single output. In summary, marked graphs allow researchers to model concurrency (multiple enabled transitions can fire simultaneously) and synchronization (a transition with multiple incoming arcs), but not choice (non-determinism).

Definition 5.5 *A* marked graph $\mathcal{MG} = (P,T,F,W,M_0)$ *is an ordinary Petri net in which each place $p \in P$ has exactly one input transition and exactly one output transition, i.e.* $\forall p \in P(|\bullet p| = |p \bullet| = 1)$.

Being an ordinary Petri net, a marked graph has constant weight function $W : F \rightarrow 1$.

A marked graph $\mathcal{MG}$ is *consistent* if there is a marking M_0 and a firing sequence ς from M_0 back to M_0 such that every transition occurs at least once in ς. In the sequel I only consider consistent marked graphs.

Example The Petri nets of Figures 5.1(a,b,d) are marked graphs, while the Petri net of Figure 5.1(c) is not (due both to the presence of the choice structure at place p_a and to the presence of two incoming arcs at place p_d). ■

Definitions and properties of Petri nets naturally extend to marked graphs. Further, marked graphs enjoy special properties on liveness and safeness as described below.

Theorem 5.1 *A marked graph $\mathcal{MG}$ is live when the initial marking M_0 assigns at least one token on each directed cycle. A marked graph $\mathcal{MG}$ is safe when every arc belongs to a directed cycle c such that the total number of tokens on c is equal to one. A live marked graph is reversible.*

Proof. See [57, 176]. □

An important property of any marked graph $\mathcal{MG}$ regards the number of tokens $M_0(c)$ that are present on a cycle c of $\mathcal{MG}$. This number remains constant for any marking that can be reached from the initial marking $\mathcal{M}_0$.

Theorem 5.2 *The number of tokens in every cycle of a marked graph remains invariant under any firing sequence.*

Proof. Proven by many researchers, e.g see [6, 57, 175, 194]. It follows from the following facts: (1) tokens in any cycle can only be produced or consumed by transitions belonging to the cycle and (2) when a transition in the cycle consumes a token, it produces a new token into the cycle. □

Strongly connected marked graphs are not necessarily safe, but they are always K-bounded. This fact is a consequence of the following theorem, which suggests also that the value K can be determined by analyzing the number of tokens $M_0(c)$ of each cycle c of $\mathcal{MG}$.

Theorem 5.3 *Given an initial marking M_0, the maximum number of tokens that a place p can have in a marked graph $\mathcal{MG}$ is equal to the minimum number of tokens $M_0(c)$ over all cycles c containing p.*

Proof. See [176]. □

Marked graphs are a natural model for homogeneous cyclic concurrent systems that have a periodic behavior where for each period the same number of events occur at each component.

Theorem 5.4 *For a strongly connected marked graph a firing sequence leads it back to the initial marking M_0 when it fires every transition an equal number of times.*

Proof. See [6, 176]. □

A marked graph $\mathcal{MG}$ is *timed* if there exists a delay $d(t)$ associated with each transition in $t \in \mathcal{MG}$. In the sequel I assume to deal only with timed marked graphs and, unless specified otherwise, that $\forall t \in \mathcal{MG}\,(d(t) = 1)$. Furthermore, I assume that a timed marked graph $\mathcal{MG}$ operates according to a *step semantics* [197, 198].

Assumption 5.1 *(Step semantics). Marked graph $\mathcal{MG}$ moves from a marking M_i to a marking M_{i+1} in a single step during which all enabled transitions fire concurrently.*

Given this assumption, the firing activity of a timed marked graph can be cast into the synchronous paradigm discussed in Section 1.1: it evolves through an infinite sequence of atomic reactions where each reaction corresponds to a step between two markings; each reaction can be indexed with a natural number capturing the progression of time (a *timestamp*); if each place is associated to a variable v, then the presence/absence of tokens in a place can be interpreted as the presence/absence of a value of v at a given timestamp.

An important metric for a timed marked graph $\mathcal{MG}$ is the average time separation between two consecutive firings of a transition [179].

Definition 5.6 *Given a transition t in a timed marked graph $\mathcal{MG}$, the* average occurrence distance $\delta(t,n)$ *of t after n executions is*

$$\delta(t,n) = \frac{b_t(n)}{n}$$

where $b_t(n)$ is the time at which transition t starts its n-th execution.

The cycle time of a transition is defined as the asymptotic value of $\delta(t,n)$.

Definition 5.7 *The* cycle time $\pi(t)$ *of a transition t of a timed marked graph $\mathcal{MG}$ is defined as*

$$\pi(t) = \lim_{n \to \infty} \delta(t,n)$$

The reciprocal of the cycle time gives the average firing rate of the transition.

Thanks to the following result, for strongly connected marked graphs it is possible to define the *cycle time* $\pi(\mathcal{MG})$ of the marked graph $\mathcal{MG}$ as a whole.

Theorem 5.5 *In a strongly connected timed marked graph $\mathcal{MG}$ all transitions have the same cycle time.*

Proof. See [179]. □

As discussed in Section 5.2.2, this quantity represents a natural performance metric for the system modeled by $\mathcal{MG}$ because its reciprocal gives the rate of consumption/production of tokens by the system, i.e. the system's throughput. Although all the transitions of a strongly connected marked graph have the same cycle time, they do not necessarily have the same average occurrence distance. Before expanding further on this point, I discuss how the cycle time of a marked graph is calculated. In this regard it is useful to define first the notion of cycle metric.

Definition 5.8 *The* cycle metric $\mu(c)$ *of a cycle c of a marked graph* $\mathcal{MG}$ *is equal to the sum of the transition delays along a cycle divided by the number of tokens in the cycle, i.e:*

$$\mu(c) = \frac{\sum_{t \in c} d(t)}{M_0(c)}$$

With the assumption that all the transition have unit delays, i.e $\forall t \in \mathcal{MG}\,(d(t) = 1)$, the numerator of the cycle metric coincides with the number of transitions in the cycle, i.e. the cardinality of the cycle.

Definition 5.9 *The* cardinality $|c|$ *of a cycle c of a marked graph* $\mathcal{MG}$ *is equal to the number of the transitions on c.*

Theorem 5.6 *The cycle time* $\pi(\mathcal{MG})$ *of a strongly connected timed marked graph* $\mathcal{MG}$ *is given by the largest cycle metric across all its cycles, i.e:*

$$\pi(\mathcal{MG}) = \max_{\forall c \in \mathcal{MG}} \{\mu(c)\}$$

Proof. See [194]. Similar results can be found in [32, 71, 175, 179, 199]. □

Definition 5.10 *In a timed marked graph* $\mathcal{MG}$, *a cycle* c^* *whose cycle metric coincides with* $\pi(\mathcal{MG})$ *is called* critical.

Hence, a naive method to compute the cycle time $\pi(\mathcal{MG})$ is simply based on the enumeration of all cycles of $\mathcal{MG}$ in order to identify a critical cycle. All the cycles of a strongly connected graph can be detected in $\bigcirc((|P| + |T| + |F|) \cdot (|C| + 1))$ operations, where C is the set of elementary cycles in the graph [120, 196].

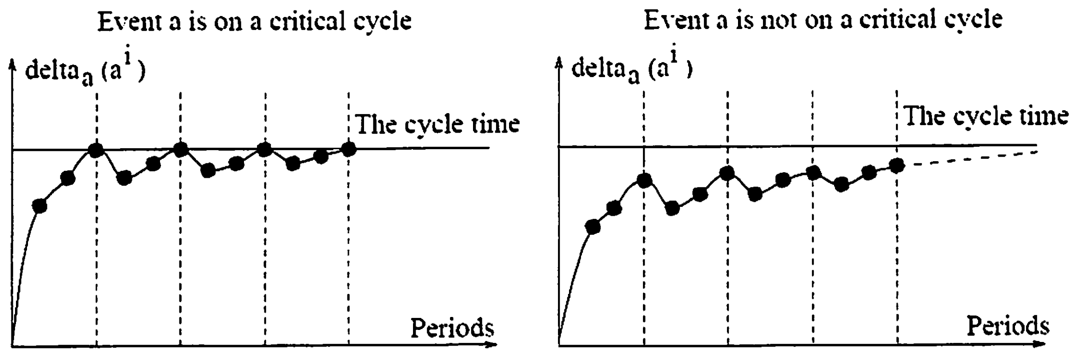

Figure 5.2: Asymptotic behavior of average occurrence distances of transitions [179].

Example The marked graph $\mathcal{MG}_d$ of Figure 5.1(d) is strongly connected and has four cycles:

$$\begin{aligned}
c_1 &= \{t_a, p_c, t_b, p_b\},\\
c_2 &= \{t_a, p_c, t_b, p_{d2}, t_{c2}, p_{a1}\},\\
c_3 &= \{t_a, p_{d1}, t_{c1}, p_{a2}, t_b, p_b\},\\
c_4 &= \{t_a, p_{d1}, t_{c1}, p_{a2}, t_b, p_{d2}, t_{c2}, p_{a1}\}
\end{aligned}$$

With the assumption that $\forall i\big(d(t_i) = 1\big)$, the cycle metrics are $\mu(c_1) = \frac{2}{1}$, $\mu(c_2) = \frac{3}{1}$, $\mu(c_3) = \frac{3}{1}$, $\mu(c_4) = \frac{4}{1}$. Therefore c_4 is the only critical cycle and the cycle time of $\mathcal{MG}_d$ is $\pi(\mathcal{MG}_d) = \mu(c_4) = 4$. ■

Although it is reasonable to expect that marked graphs modeling real systems have a limited number of cycles, in general the number of cycles in a directed graph can be exponential in the number of its arcs. Ramamoorthy and Ho point out that if the required performance of the system is given, then it can be verified efficiently in $O(|P|^3)$ without enumerating all cycles by using Floyd's shortest path algorithm [86]. In his thesis [32] Burns discusses the problem of computing the cycle time [2] and explains the connection between this problem and linear programming (LP). This connection was originally observed

[2]Strictly speaking, Burns uses the term *minimum cycle period* instead of cycle time and his underlying formalism is based on *event-rule systems*, which are equivalent to timed marked graphs.

by Magott who formulated a LP problem with $|T|+1$ variables and $|P|$ constraints, which is solved with a general-purpose polynomial algorithm [156]. Alternative formulations of this LP problem have been proposed and their relationships are discussed in [250]. Burns provides a specialized algorithm that exploits the particular structure of this LP problem to solve it in a low-order polynomial time. In Section 5.3.1 I discuss the relationships between the cycle time of a marked graph and the maximum cycle mean of a generic weighted directed cyclic graph. Specifically, I show under which hypothesis the former can be computed by applying one of the many algorithms that have been proposed, dating back to Karp's algorithm [127], to find the value of the latter.

The definition of critical cycle is instrumental to understanding the difference between cycle time $\pi(t)$ and average occurrence distance $\delta(t,n)$ of a transition t in a strongly connected marked graph $\mathcal{MG}$. As explained by Nielsen and Kishinevsky [179], it is the location of t on $\mathcal{MG}$ that makes the difference: if transition t is located on a critical cycle of $\mathcal{MG}$ then its average occurrence distance $\delta(t,n)$ becomes equal to the cycle time $\pi(\mathcal{MG})$; instead, if t is not located on a critical cycle then it does still present an asymptotic behavior (as expected from Theorem 5.5), but it actually never reaches the asymptote $\pi(\mathcal{MG})$. The different asymptotic behaviors are reported in Figure 5.2, which is taken from [179]. In summary, cycle time $\pi(\mathcal{MG})$ is an upper-bound on the asymptotic performance of the system modeled by $\mathcal{MG}$.

5.2 Performance Analysis of Latency-Insensitive Systems

In this section I present *constructive modeling*, an approach based on marked graphs to capture the structure of a latency-insensitive system at the level of inter-shell communication (protocol level) and analyze its performance. Latency-insensitive systems can be conveniently modeled with marked graphs because at the protocol level they operate as deterministic systems. Given the specification of a latency-insensitive system S, I derive a *marked-graph model* $\mathcal{MG}_S$ representing a particular implementation of S.

Marked-graph models make it possible to abstract away the details of the internal logic of each component of S as well as the particular format of the data that the shells exchange on the channels of S. Instead, the focus is on the presence/absence of the data on these channels, represented by the presence/absence of tokens in the places of $\mathcal{MG}_S$ [3]. In other words, the focus is on the analysis of the performance of the implementation of S, after abstracting away the details of its functional behavior. This corresponds to decoupling functionality from performance, which is an application of the principle of orthogonalization of concerns (see Section 2.3).

In particular, a marked-graph model $\mathcal{MG}_S$ can be used to compute exactly the maximum throughput $\vartheta(S)$ at which the implementation of S can work. This performance metric is fundamental to comprehend the interaction of the implementation of S with the environment in which it operates.

I present two marked-graph models. Both models work on the assumption that the system operates in an environment that "understands" the basics of latency-insensitive protocols. In other words, the environment is able to distinguish between informative events and stalling events. However, the two models differ in the level of this understanding. In both cases the environment is able to detect whether an output event produced by the system is informative or not. But only in the second model the environment can stall the system as well as be forced by the system—via back-pressure—to postpone the transmission of a new informative event.

The first marked-graph model, which is described in Section 5.2.3, relies on the additional assumption that the shells have input queues with infinite length. Naturally *infinite queues* cannot be physically manufactured. Besides being simpler, however, this model is practically useful to analyze system performance in the special case when the environment never asks the system to stall, i.e. stalling events are only produced inside the system. The model can be used to calculate the maximum throughput that the system can sustain as well as the number of stalling events that cycle around during its operations. This information can then be used to verify whether the system's throughput is high enough to satisfy the performance requirement imposed by the environment and, if so, to statically determine the minimum finite size of each queue in order to guarantee correct operations.

[3]With respect to the encoding of the latency-insensitive protocol discussed in Section 4.2.1, the presence of data corresponds to the presence of a true packet (informative event), while the absence of data corresponds to the presence of a void packet (stalling event).

The second marked-graph model, described in Section 5.2.4, assumes that the shells have finite-length input queues and communicate by means of channels implementing a latency-insensitive protocol with back-pressure (like the one discussed in Section 4.2). This translates into the assumption that the environment must be willing to stall if the system requires it to do so (by "sending-back" a stalling event). There is also, however, an important advantage: this model can likewise handle the case when it is the environment that may randomly require the system to stall by sending stalling events to its input ports. Furthermore, at the price of being slightly more complex, this model offers the key to understanding the impact of queue lengths on the behavior of a latency-insensitive system. In particular, it makes it possible to answer the question of whether there is a "single special length" for the queues of all shells in order to support an *optimal* latency-insensitive protocol, i.e. a protocol that, in the best case, can sustain communication with the maximum possible throughput (equal to one).

The two marked-graph models are described in Section 5.2.3 and Section 5.2.4 respectively, while the next two sections contain concepts that are common to both.

5.2.1 Constructive Modeling of Latency-Insensitive Systems

Given a latency-insensitive system S, a marked-graph model $\mathcal{MG}_S$ is constructed in two main steps:

1. define a synthetic library $\mathcal{L}$ of simple marked graphs, which act as *primitives*; specifically, the library must contain one primitive marked graph to model a relay station and a "meta-primitive" for modeling the shell-core pairs; the meta-primitive is a generic template from which a unique primitive marked graph can be derived for a given shell-core pair based only on the cardinality of its input and output sets (the shell I/O interface);

2. execute the *constructiveModeling* procedure of Figure 5.3 to obtain the composite marked graph $\mathcal{MG}_S$ by assembling primitives that are instanced from $\mathcal{L}$.

The *constructiveModeling* procedure is independent from the particular library $\mathcal{L}$ of primitives. In Sections 5.2.3 and 5.2.4 I present primitives from two different libraries which

```
procedure constructiveModeling
    1. take a latency-insensitive system $S$ and a synthetic primitive library $\mathcal{L}$;
    2. for each relay station $i$ of $S$, instance the corresponding primitive marked
       graph $\mathcal{MG}_i$ from $\mathcal{L}$;
    3. for each shell-core pair $j$ of $S$, instance the corresponding primitive marked
       graph $\mathcal{MG}_j$ after generating it from the meta-primitive template in $\mathcal{L}$ based
       on the I/O interface of $j$;
    4. for all relay station $i$ and shell-core pair $j$ of $S$ connect the instanced primitives
       $\mathcal{MG}_i, \mathcal{MG}_j$ based on the interconnect pattern of $S$;
    5. return the composite marked graph $\mathcal{MG}_S$;
```

Figure 5.3: Procedure to build a marked-graph model for a latency-insensitive system.

can be used to derive distinct marked-graph models associated to different implementations of the same latency-insensitive system S. However, both models are derived executing the *constructiveModeling* procedure of Figure 5.3.

The *constructiveModeling* procedure does not necessarily have to be performed directly on the whole system specification S. It is also possible to act on a decomposition of S as follows: derive first a marked-graph model for each sub-system of the decomposition and then build incrementally $\mathcal{MG}_S$ by connecting them. Furthermore, possible modifications or expansions of S can be tracked immediately on $\mathcal{MG}_S$ to assess their impact on the overall system behavior. For instance, modeling the insertion of a relay station on a channel is straightforward (simply insert the corresponding primitive marked graph). These operations are simple because the constructive modeling approach relies on the property of compositionality of patient processes discussed in Section 3.3.

Example Figure 5.4 represents a simple latency-insensitive system S with four shells S_a, S_b, S_f, S_e and three relay stations RS_c, RS_d, RS_g. System S is modeled by marked graph

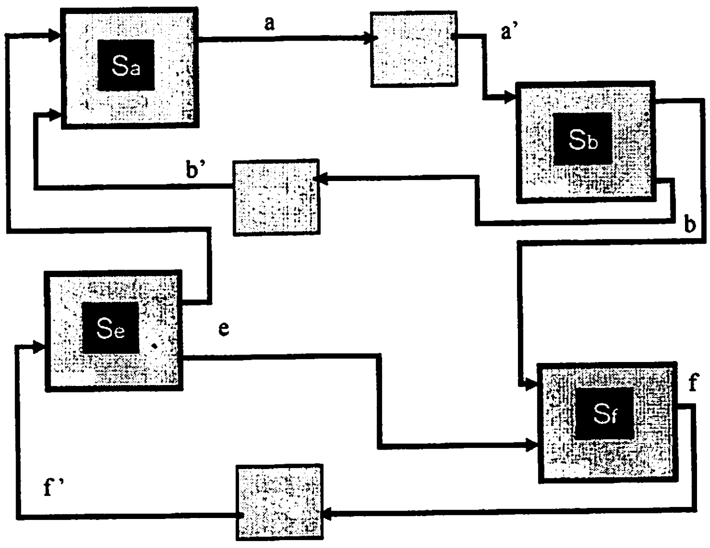

Figure 5.4: Example of latency-insensitive system with no back-pressure.

$\mathcal{MG}_3$ on the right-hand side of Figure 5.5. The derivation of this marked-graph model will be clearer after reading Section 5.2.3 where I present the details of the particular library that was used. On the left-hand side of Figure 5.5 two marked graphs $\mathcal{MG}_1$ and $\mathcal{MG}_2$ represent a possible decomposition of $\mathcal{MG}_3$. Conversely, $\mathcal{MG}_3$ can be seen as the marked graph that is obtained by connecting $\mathcal{MG}_1$ and $\mathcal{MG}_2$, which communicate by means of places P_x and P_y. A general rule of the proposed constructive modeling approach is that connecting two or more marked graphs involves only the addition of new places and arcs between them. Consequently, each single component marked graph preserves its structure while becoming part of a larger composite marked graph. ■

The fundamental performance metric of a latency-insensitive system S is the rate of production of informative events, i.e. its throughput $\vartheta(S)$. The throughput of S depends on two factors: the internal structure of S and the interaction of S with the environment where it operates. The structure of S determines the maximum throughput that S can sustain. Latency-insensitive system S effectively runs at this throughput unless the environment forces it to slow down by either not providing enough informative events to process or

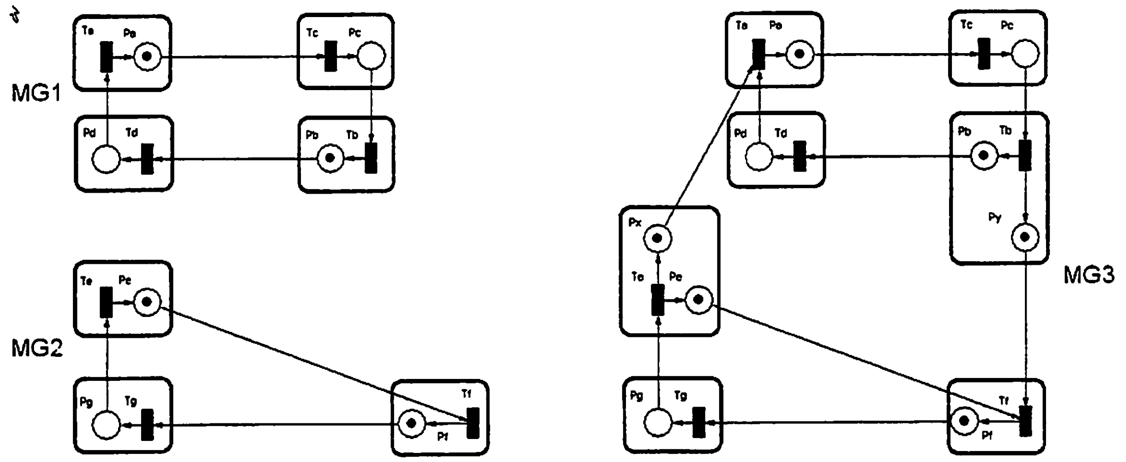

Figure 5.5: Example of constructive modeling with marked graphs.

requiring it to wait via back-pressure.

In Section 4.3 I discussed qualitatively how the insertion of relay stations, by changing the internal structure of S, may have a negative impact on $\vartheta(S)$. To quantify such impact, in the next section I define the notion of maximum sustainable throughput of a marked graph. Then, I explain how to use the properties reported in Section 5.1.2 to compute the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$ of the marked graph $\mathcal{MG}_S$ modeling S. Since, *by construction*, there is a one-to-one correspondence between the tokens of $\mathcal{MG}_S$ and the informative events of S, it follows naturally that $\vartheta(S) \leq \vartheta(\mathcal{MG}_S)$. Obviously, this inequality becomes an equation when the environment is is able to adapt itself to the system's throughput.

5.2.2 Maximum Sustainable Throughput

Definition 5.7 implies that the reciprocal of the cycle time of a transition corresponds to the average firing rate of the transition. Theorem 5.5 guarantees that all transitions in a strongly connected timed marked graph have the same cycle time. These facts together with Assumption 5.1 on step semantics lead to the definition of the maximum sustainable throughput of a strongly connected marked graph simply as the lesser between the reciprocal of its cycle time and one [4].

Definition 5.11 *The* maximum sustainable throughput *of a strongly connected marked graph $\mathcal{MG}$ is defined as*

$$\vartheta(\mathcal{MG}) = \min\left\{1,\ \frac{1}{\pi(\mathcal{MG})}\right\}$$

Hence, the maximum sustainable throughput is a rational number in the interval $]0,1]$. The modifier "maximum sustainable" comes from the following fact: *when a marked graph is connected to another marked graph its throughput can only decrease.*

In the context of constructive modeling, a marked graph represents a patient process of a latency-insensitive system. This can either be a primitive marked graph (a shell/core pair or a relay station) or a composite marked graph, which is obtained by connecting primitive marked graphs. When two generic strongly connected marked graphs $\mathcal{MG}_A$ and $\mathcal{MG}_B$ are connected, two scenarios are possible for the resulting marked graph $\mathcal{MG}_C$: it is either strongly connected or not.

When $\mathcal{MG}_C$ is strongly connected, the cycle time of each component marked graph (taken as a stand-alone marked graph) represents a lower bound on the cycle time of the composite marked graph, i.e.

$$\pi(\mathcal{MG}_A) \leq \pi(\mathcal{MG}_C) \geq \pi(\mathcal{MG}_B)$$

In fact, comparing the observation of the firing of a transition t of $\mathcal{MG}_A$ as a component within $\mathcal{MG}_C$ with the observation of the firing of t when $\mathcal{MG}_A$ is stand-alone gives:

$$\pi_{\mathcal{MG}_C}(t) \geq \pi_{\mathcal{MG}_A}(t),$$

i.e. as a result of the composition the cycle time of t can only increase. This is a consequence of Theorems 5.5 and 5.6: since all cycles of $\mathcal{MG}_A$ are also cycles of $\mathcal{MG}_C$, then $\pi(\mathcal{MG}_A)$ is a lower bound of the cycle time of $\mathcal{MG}_C$. Conversely, $\vartheta(\mathcal{MG}_A)$ is an upper bound of $\vartheta(\mathcal{MG}_C)$, justifying the modifier "maximum sustainable" that is attached to the term "throughput".

[4] Each transition in a marked graph when it fires consumes a token from each input place and produces a token on each output place because a marked graph is an ordinary Petri net. Therefore, given Assumption 5.1, at each step all enabled transitions fire concurrently and only *once*.

Example All marked graphs of Figure 5.5 are strongly connected. Marked graph $\mathcal{MG}_1$ contains only cycle $c_1 = \{t_a, p_a, t_c, p_c, t_b, p_b, t_d, p_d\}$ and marked graph $\mathcal{MG}_2$ contains only cycle $c_2 = \{t_e, p_e, t_f, p_f, t_g, p_g\}$. The cycle times of $\mathcal{MG}_1$ and $\mathcal{MG}_2$ coincide with the cycle metrics of their corresponding single cycles, i.e: $\pi(\mathcal{MG}_1) = \mu(c_1) = \frac{4}{2}$ and $\pi(\mathcal{MG}_2) = \mu(c_2) = \frac{3}{2}$. Besides cycles c_1 and c_2, marked graph $\mathcal{MG}_3$ presents the additional cycle $c_3 = \{t_a, p_a, t_c, p_c, t_b, p_y, t_f, p_f, t_g, p_g, t_e, p_x\}$ with cycle metric $\mu(c_3) = \frac{6}{4}$. Hence, $\mathcal{MG}_3$ has one critical cycle, c_1, and cycle mean $\pi(\mathcal{MG}_3) = \pi(\mathcal{MG}_1) = \mu(c_1) = \frac{4}{2} = 2$. Notice that $\pi(\mathcal{MG}_3)$ remains equal to 2 regardless of the number of tokens that the initial marking presents on P_x and P_y. Seen from the "component perspective" of $\mathcal{MG}_1$ and $\mathcal{MG}_2$, any possible composition may only increase their cycle time, or, dually, make them operate at a throughput which can be at most equal to their maximal sustainable throughput. Instead, from the "system perspective" of $\mathcal{MG}_3$, its throughput is dictated by the slowest among its components, which in this case is $\mathcal{MG}_1$. ■

When $\mathcal{MG}_C$ is not strongly connected the analysis is not as simple. The next example illustrates the main issue with this scenario and suggests a possible way to handle it.

Example The marked graph $\mathcal{MG}_b$ of Figure 5.1(b) is not strongly connected. Instead, it presents two strongly connected components that coincide respectively with cycle $c_1 = \{t_5, p_5, t_6, p_6\}$ and cycle $c_2 = \{t_7, p_8, t_8, p_9, t_9, p_{10}\}$ that are connected via place p_7. Their cycle metrics are respectively $\mu(c_1) = \frac{2}{1}$ and $\mu(c_2) = \frac{3}{1}$. Since it is the strongly connected component with the smallest cycle time that outputs tokens to the other, there is unbounded token accumulation at place p_7. Therefore, $\mathcal{MG}_b$ is not bounded. Further, the transitions of $\mathcal{MG}_b$ do not have the same cycle time: t_5 and t_6 have cycle time equal to 2, while t_7, t_8 and t_9 have cycle time equal to 3. Indeed, the cycle time of the entire marked graph is only defined if it is strongly connected, since Theorem 5.5 does not hold otherwise.

On the other hand, reversing the direction of the linking between the two strongly connected components results in a new marked graph that is now bounded, albeit still not strongly connected. Strictly speaking, it is not correct to refer to the cycle time of the entire marked graph in this case either. Nevertheless transitions t_5 and t_6 are now forced to fire with a cycle time that is also equal to 3, i.e: mirroring the cycle metric of c_3. All transitions therefore have now the same cycle time. ■

Without touching the classical concept of cycle time, I extend the definition of maximum sustainable throughput to marked graphs that are not strongly connected as well as to marked graphs that are acyclic. Recall that a marked graph that is not strongly connected is still a directed graph and, therefore, can be partitioned in strongly connected components (SCCs) using one of the various algorithms that have been proposed [62], including Tarjan's linear-time algorithm [232] or, in case of very large graphs, algorithms based on implicit techniques [246]. In particular, it is simple to build the *component graph* G^{SCC} where each vertex represents an SCC of $\mathcal{MG}$ and there is one arc between two vertices of G^{SCC} whenever there is at least one arc between the two corresponding SCCs of $\mathcal{MG}$ [62]. G^{SCC} is a directed acyclic graph (DAG). The component graph and the maximum sustainable throughput of each SCC are the tools to analyze the interaction between the various SCCs during the system's operation.

Without lack of generality, consider the scenario where marked graph $\mathcal{MG}_C$ has only two SCCs, $\mathcal{MG}_A$ and $\mathcal{MG}_B$, with $\mathcal{MG}_A$ passing tokens to $\mathcal{MG}_B$. Considering the two SCCs as stand-alone marked graphs, three cases are possible:

- $\vartheta(\mathcal{MG}_A) < \vartheta(\mathcal{MG}_B)$, i.e. the rate of token production of $\mathcal{MG}_A$ is smaller than the rate of token consumption of $\mathcal{MG}_B$. Therefore, $\mathcal{MG}_B$ is "slowed down" by $\mathcal{MG}_A$. The composite marked graph $\mathcal{MG}_C$ consumes and produces tokens with the same rate as $\vartheta(\mathcal{MG}_A)$. This suggests that the definition of maximum sustainable throughput can be relaxed by writing $\vartheta(\mathcal{MG}_C) = \vartheta(\mathcal{MG}_A)$ even though $\mathcal{MG}_C$ is not strongly connected.

- $\vartheta(\mathcal{MG}_A) = \vartheta(\mathcal{MG}_B)$, i.e. $\mathcal{MG}_B$ consumes token at the same rate that they are produced by $\mathcal{MG}_A$. Therefore, it is natural to write again: $\vartheta(\mathcal{MG}_C) = \vartheta(\mathcal{MG}_A)$.

- $\vartheta(\mathcal{MG}_A) > \vartheta(\mathcal{MG}_B)$, i.e. the rate of token production of $\mathcal{MG}_A$ is larger than the rate of token consumption of $\mathcal{MG}_B$. Therefore, $\mathcal{MG}_B$ cannot "keep up" with $\mathcal{MG}_A$. In fact, $\mathcal{MG}_C$ is not a bounded marked graph because at the boundary between the two SCCs there is a place p that experiences unbounded token accumulation. In this case, to define a notion of throughput of $\mathcal{MG}_C$ is less immediate because the latency-insensitive system modeled by $\mathcal{MG}_C$ can produce informative events with rate $\vartheta(\mathcal{M}\mathcal{G}_B)$ but cannot really consume them with rate $\vartheta(\mathcal{M}\mathcal{G}_A)$. In fact, it cannot even store all of them! The reason is that practically it is not possible to implement an infinite queue in place of p. Instead, a feasible implementation of a latency-insensitive system can be obtained using only storage elements of finite capacity. Therefore, one of the goals of constructive modeling with marked graphs is to verify the boundedness of $\mathcal{M}\mathcal{G}_C$, as this captures a necessary physical limit for the implementation of the latency-insensitive system modeled by $\mathcal{M}\mathcal{G}_C$.

These observations further motivate the modifier "maximum sustainable" in Definition 5.11. Setting $\vartheta(\mathcal{M}\mathcal{G}_C) = \vartheta(\mathcal{M}\mathcal{G}_B)$ is a concise way to capture a fundamental practical limitation, namely the maximum rate at which the environment can *safely* send informative events (tokens) to the system. Conversely, it is also the rate at which it must send them to keep the system operating at the top of its capabilities.

There is one last scenario of marked graph composition to consider: marked graphs $\mathcal{M}\mathcal{G}_A$ and $\mathcal{M}\mathcal{G}_B$ are acyclic graphs and their connection gives a marked graph $\mathcal{M}\mathcal{G}_C$ that is also acyclic. Since an acyclic marked graph can sustain any rate of production/consumption, it is reasonable to set its maximum sustainable throughput equal to one by definition. This confirms the intuition, because a marked graph without cycles represents a pipelined system without feedback paths. Hence, for any possible initial marking there exists a finite number K of steps after which, all token vacancies (stalling events) originally in the system have been ejected through its output ports and, as long as each input port continues to receive an input token at each step, each place in the system contains a token.

Based on the above discussion, Definition 5.11 can be revised as follows:

Definition 5.12 *The* maximum sustainable throughput *of a marked graph $\mathcal{M}\mathcal{G}$ is defined as*

$$\vartheta(\mathcal{M}\mathcal{G}) = \begin{cases} 1 & \text{if } \mathcal{M}\mathcal{G} \text{ is acyclic;} \\ \min\left\{1, \frac{1}{\pi(\mathcal{M}\mathcal{G})}\right\} & \text{if } \mathcal{M}\mathcal{G} \text{ is cyclic and strongly connected;} \\ \min_{\forall \mathcal{M}\mathcal{G}_{SCC} \in \mathcal{M}\mathcal{G}}\left\{\vartheta(\mathcal{M}\mathcal{G}_{SCC})\right\} & \text{otherwise.} \end{cases}$$

In the sequel, when a marked graph $\mathcal{M}\mathcal{G}$ is firing at the top of its capability, I may refer to $\vartheta(\mathcal{M}\mathcal{G})$ simply as the throughput of $\mathcal{M}\mathcal{G}$.

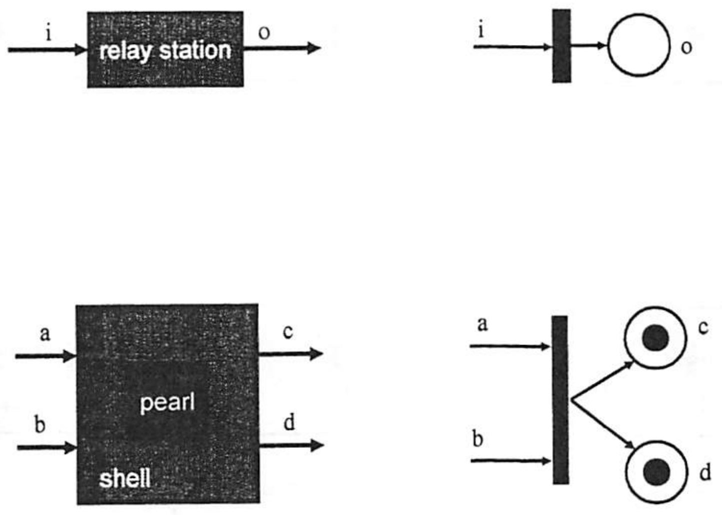

Figure 5.6: Primitive marked graphs from the infinite-queue model library $\mathcal{L}_{IQ}$.

5.2.3 Performance Analysis with the Infinite-Queue Model

The first constructive model is based on the assumption that the shells have unlimited storage space in their input queues, i.e. queues of infinite length. This model is called the *infinite-queue model*. Figure 5.6 illustrates the library $\mathcal{L}_{IQ}$ for this model. The diagrams on the left-hand side represent respectively a relay station and a shell-core pair with two input channels and two output channels. On the right-hand side there are the corresponding primitive marked graphs. Primitive marked graphs for shells having different I/O interfaces have similar structures.

In the infinite-queue model a relay station is simply represented as a transition with an incoming arc, an outgoing arc, and an output place that does not contain a token in the initial marking. This *initially empty place* fulfills two roles:

- it represent the generic storage element that makes a relay station be a stateful repeater which can be used to pipeline a channel between two shells;

- it captures the fact that the initialization value for the storage element of a relay

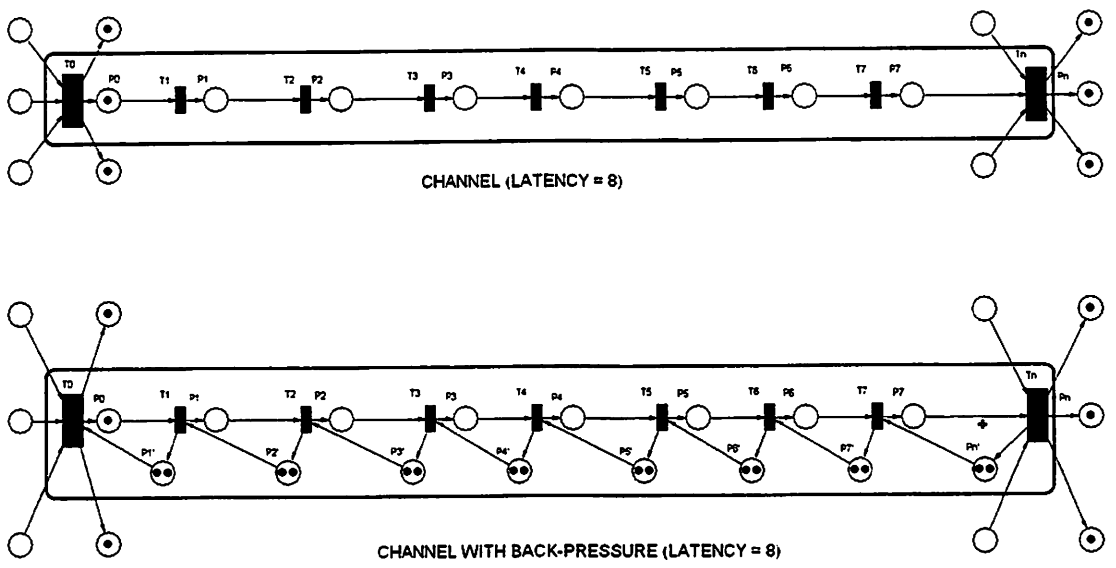

Figure 5.7: Channels in marked graph models.

station is a stalling event as discussed in Section 4.3.1.

In the infinite-queue model, the marked-graph modeling a shell-core pair with m input channels and n output channels is simply a transition with m incoming arcs, n outgoing arcs and n distinct output places. Also, in the initial marking, each of these places contains exactly one token representing the informative event that is associated to the corresponding initialization value stored in the output register of the sequential core, i.e. the first informative event of the corresponding signal according to the strict system specification. The logic controlling the operations of a shell-core pair is implicitly captured by the firing rule that governs a transition in a ordinary marked graph: *a transition fires when each of its input place presents at least a token.* This rule corresponds to an AND-causality semantics [96] and for a latency-insensitive system can be reinterpreted as follows: if a new informative event is available on each input channel then the core processes all of them, thereby updating its internal state and producing a new informative event on each output channel; instead, it is sufficient that one informative event is missing (i.e. a stalling event is present in its place) for the shell to freeze the computation of the core. Based on Assumption 5.1, a marked graph evolves through a sequence of firing steps indexed by timestamps and at each step all enabled transitions fire concurrently. Therefore, the freezing of a core of the latency-insensitive system at a given clock cycle is simply captured in the marked-graph model by the fact that the corresponding transition does not fire at the corresponding timestamp.

The notion of channel of a marked graph is instrumental to define formally an infinite-queue marked-graph model.

Definition 5.13 *A* channel q *of a* marked graph $\mathcal{MG} = (P, T, F, W, M_0)$ *is a directed elementary path* $\{t_0, p_0, t_1, p_1, \ldots, t_{n-1}, p_{n-1}, t_n\}$ *such that* $\forall n \in \mathbb{N}$ *and* $\forall i \in [1, n-1]$

$$\begin{aligned} M(p_0) &= 1 \\ \forall p \in t_n\bullet & \left(M(p) = 1\right) \\ M(p_i) &= 0 \\ |t_i \bullet| &= 1 \\ |\bullet t_i| &= 1 \end{aligned}$$

The latency *of channel* q *is* $\lambda(q) = n$. *Transition* t_0 *is the* tail transition $tail(q)$ *of* q. *Transition* t_n *is the* head transition $head(q)$ *of* q. *Transitions* $\{t_1, \ldots, t_{n-1}\}$ *are the elements of the set* $int(q)$ *of* internal transitions *of* q. *The set of channels of a marked graph* $\mathcal{MG}$ *is denoted as* $\mathcal{Q}(\mathcal{MG})$.

A channel in a marked-graph model corresponds exactly to a channel of a latency-insensitive system, i.e. a point-to-point connection between two shell/pairs cores. A channel may contain any number of consecutive relay stations. The latency $\lambda(q)$ of a channel q is equal to the number of its internal transitions (relay stations) plus one. A channel has minimum latency, equal to one, when it does not contain any transition.

Example The top diagram of Figure 5.7 illustrates a marked-graph channel q with latency $\lambda(q) = 8$. ■

An infinite-queue marked-graph model is a marked graph where no place has initial marking greater than one and the input transition of each initially empty place belongs to the set of internal transition of exactly one channel.

Definition 5.14 *An* infinite-queue marked-graph model *of a latency-insensitive system implementation S, is a* marked graph $\mathcal{MG} = (P,T,F,W,M_0)$ *such that* $\forall p \in P\Big(M_0(p) \leq 1\Big)$ *and* $\forall p \in P_0\, \exists! q \in Q(\mathcal{MG})\, \Big(\bullet p \subseteq int(q)\Big)$.

Hence, with respect to library $\mathcal{L}_{IQ}$, each internal transition of a channel together with its output place corresponds to a primitive marked graph for a relay station, while each of the remaining transitions corresponds to a primitive marked graph for a shell.

Example Marked graph $\mathcal{MG}_3$ of Figure 5.5 is an infinite-queue model of the latency-insensitive system S of Figure 5.4. Transitions t_a, t_b, t_f, t_e together with the corresponding output places correspond to shells S_a, S_b, S_f, S_e respectively. Notice that S_a and S_f have two input channels and one output channel, while S_b and S_e have two output channels and one input channel. Overall, $\mathcal{MG}_3$ has six channels:

$$\begin{aligned}
q_{\{t_b,t_f\}} &= \{t_b, p_y, t_f\} \quad \text{with} \quad \lambda(q_{\{t_c,t_f\}}) = 1\\
q_{\{t_e,t_f\}} &= \{t_e, p_e, t_f\} \quad \text{with} \quad \lambda(q_{\{t_e,t_f\}}) = 1\\
q_{\{t_e,t_a\}} &= \{t_e, p_x, t_a\} \quad \text{with} \quad \lambda(q_{\{t_e,t_c\}}) = 1\\
q_{\{t_a,t_b\}} &= \{t_a, p_a, t_c, p_c, t_b\} \quad \text{with} \quad \lambda(q_{\{t_a,t_b\}}) = 2\\
q_{\{t_b,t_a\}} &= \{t_b, p_b, t_d, p_d, t_a\} \quad \text{with} \quad \lambda(q_{\{t_b,t_a\}}) = 2\\
q_{\{t_f,t_e\}} &= \{t_f, p_f, t_g, p_g, t_e\} \quad \text{with} \quad \lambda(q_{\{t_f,t_e\}}) = 2
\end{aligned}$$

Transitions t_c, t_d, t_g together with their single output place correspond to relay stations RS_c, RS_d, RS_g respectively Figure 5.5 shows the initial marking of $\mathcal{MG}_3$, which captures the initialization values of the storage elements within shell/core pairs and relay stations in accordance with the discussion of Section 4.3.1. Tokens correspond to informative events and "token vacancies" correspond to stalling events (i.e. absences of informative events). Observe the presence of one token on each place of a shell/core pair and the lack of tokens in the place of each relay station. Figure 5.8 shows the same marked graphs after 45 timestamps (steps) have passed by. To look at the extra "count" places that have been added to track the firing activity in each marked graph—recall that these marked graphs are all strongly connected—gives a visual confirmation of the values for their throughput:

$\vartheta(\mathcal{MG}_1) = \frac{1}{2}$, $\vartheta(\mathcal{MG}_2) = \frac{2}{3}$, and $\vartheta(\mathcal{MG}_3) = \frac{1}{2}$. Naturally, these values can be computed statically based on the initial marking of $\mathcal{MG}_3$ by using the result of Theorem 5.6 or one of the methods discussed in Section 5.3.1. ■

As discussed in Section 4.2.2, if a given shell S_i with multiple input channels cannot fire for one or more consecutive timestamps due to the repeated absence of an informative event on an input channel, then it has to buffer one or more informative events that, in the meantime, may have arrived on the remaining channels. In this model, the input-buffering activity of a shell is captured by the accumulation of tokens in the input places of the shell transition [5]. Strictly speaking, based on Figure 5.6, these places represent either the output register of a core in an up-link shell/core pair or the storage element in an up-link relay station. However, latency-insensitive design aims at producing modular system whithout touching the internal logic of the core modules. This is why the additional buffering space is located within the shells. The bottom line is that if n is the maximum number of tokens that can be present at any given timestamp on a input place p of a shell S_i, then the input queue of S_i must be sized with a length equal to n.

Figure 5.8 is a snapshot of the firing activity of marked graph $\mathcal{MG}_3$ taken at timestamp 45. Observe that place P_x contains two tokens at this timestamp. In fact, by analyzing the periodic behavior of $\mathcal{MG}_3$ it is possible to determine which transition fires at every timestamp. This allows us to compute statically how many tokens there are in each place at each timestamp and, consequently, what is the maximum number of tokens that may ever stay at a given place (over all timestamps). In this example, t_a fires every other timestamp, and P_x, which lies at the interface from the faster to the slower SCC, contains alternatively one token or two tokens. Therefore, it is necessary to add an extra storage slot between t_e and t_a to buffer the additional token. While place P_x originally represents an output register of the core associated to transition t_e, the necessary additional slot must be implemented as a unit-length input queue within the shell S_a associated to t_a because it is shell S_a that synchronizes the input coming from S_e with those coming from S_d.

[5]Recalling the step semantics, it is easy to prove that there can be accumulation of tokens only on the input places of a transition with multiple incoming arcs. Hence, among the primitives of library $\mathcal{L}_{IQ}$, token accumulation can only happen in front of a transition associated to a shell with multiple inputs (and not a relay station). This is consistent with the goals of the infinite-queue model, which does not capture the back-pressure mechanism.

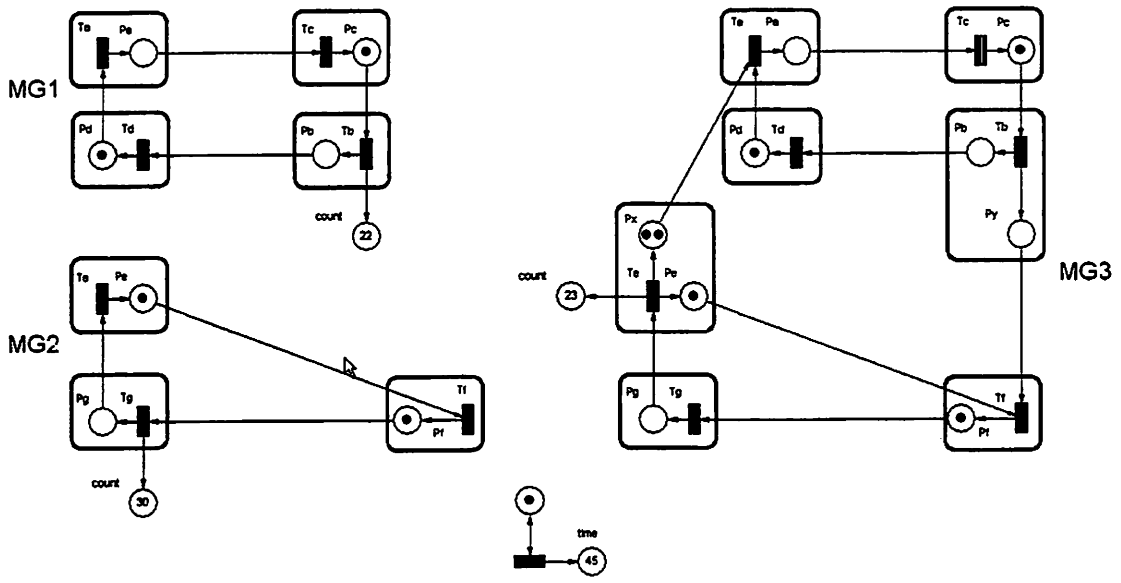

Figure 5.8: The marked graph of Figure 5.5 after 45 timestamps.

Using the marked-graph model it is easy to have a confirmation that a latency-insensitive system does never deadlock, i.e. it is live *by-construction.*

Theorem 5.7 *The* infinite-queue marked-graph model $\mathcal{MG}_S$ *of a latency-insensitive system implementation S is live.*

Proof. Theorem 5.1 guarantees that any marked graph $\mathcal{MG}$ is live when the initial marking M_0 assigns at least one token on each directed cycle of $\mathcal{MG}$. Hence, it sufficient to prove that in $\mathcal{MG}_S$ there is no cycle c that does not have any token on its places. By Definition 5.14 every initially empty place $p \in P_0$ of $\mathcal{MG}_S$ is contained in exactly one channel $q \in Q(\mathcal{MG}_S)$. Without loss of generality, let consider any place p and its channel $q = \{t_0, p_0, t_1, p_1, \ldots, p, \ldots, t_{n-1}, p_{n-1}, t_n\}$. By Definition 5.13 place p_0 contains one token. Recall that every place of a marked graph has only one input and one output transition. Also, every internal transition of a channel has only one input and one output place. Therefore, places p_0 and p belong exactly to the same cycles in $\mathcal{MG}_S$. Thanks to the generality of choosing p, all cycles of $\mathcal{MG}_S$ contain at least one place p_0 having a token in the initial marking. □

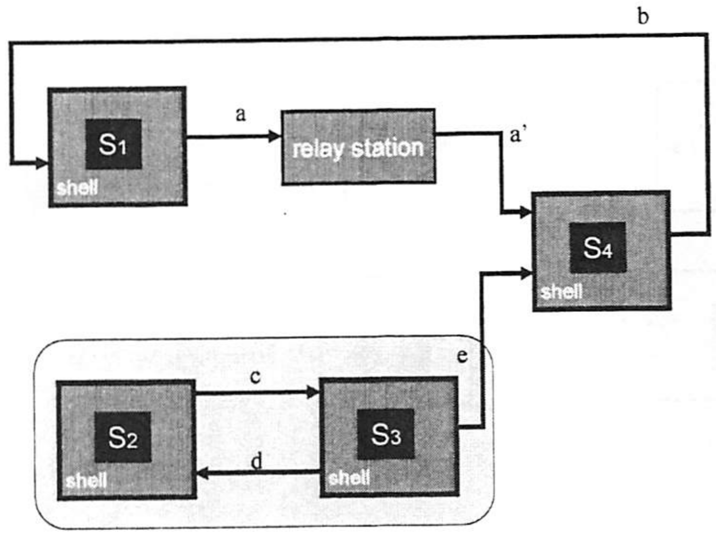

Figure 5.9: A non-strongly connected latency-insensitive system without back-pressure.

The previous result corresponds to the fact that a latency-insensitive system is obtained from a patient system by encapsulating stallable cores with shells and, where necessary, inserting relay station on the channels between the shells. Therefore, in a latency-insensitive system it is not possible to have a feedback loop that contains only relay stations without also, at least, one shell/pair core.

Example Figure 5.9 shows an implementation of a latency-insensitive system S that is not strongly connected. The diagram Figure 5.9 could be legitimately interpreted as a sub-system $\mathcal{S}_{down}$ (consisting of two shells S_1 and S_4 and one relay station) that receives input data from another sub-system $\mathcal{S}_{up}$ (consisting of shells S_2 and S_3). From each sub-system viewpoint, the other sub-system represents the environment. System S is represented by the marked-graph model of Figure 5.10, which is made of the primitives from library $\mathcal{L}_{IFQ}$ illustrated in Figure 5.6. The marking in Figure 5.14 is the initial marking. Figure 5.11 shows the marking for the same marked graph after 30 timestamps have passed by from the initial marking. Observe that there is unbounded accumulation of tokens at place e between shell S_3 and shell S_4. This accumulation is not surprising because $\mathcal{S}_{up}$ has throughput

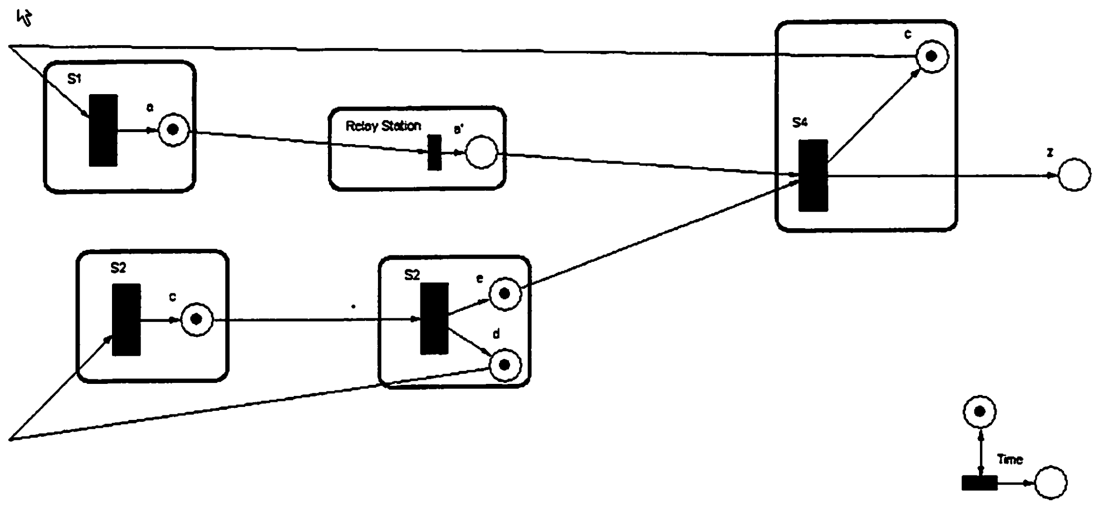

Figure 5.10: Infinite-queue model for the latency-insensitive system of Figure 5.13.

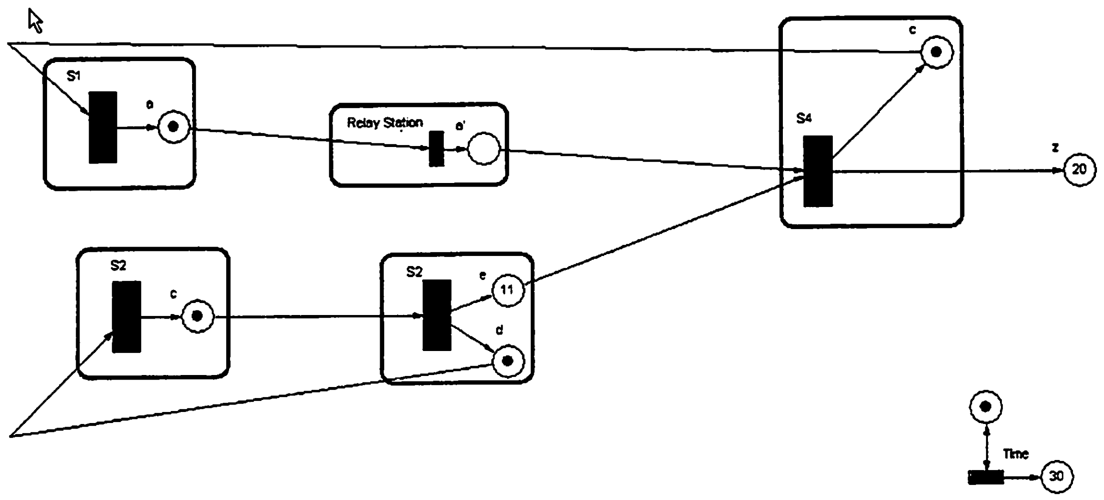

Figure 5.11: The marked graph of Figure 5.10 after 30 timestamps.

$\vartheta(\mathcal{S}_{up}) = 1$ while $\mathcal{S}_{down}$ has maximum sustainable throughput $\vartheta(\mathcal{S}_{down}) = \frac{2}{3}$. Based on Definition 5.11, the maximum sustainable throughput of the whole system is $\vartheta(\mathcal{S}) = \frac{2}{3}$. However, in this case, the implementation has a correct behavior only if at place e there is an infinite queue. ■

The previous example shows the pros and cons of the infinite-queue model. If the latency-insensitive system S is a *closed* system, i.e. designers can modify all its parts, then the infinite queue model is sufficient to reason on the interaction of the sub-systems of S and how this affects the functionality and performance of S. In the above example, since infinite queues are not feasible, designers must decide how to modify the system, e.g. either slow-down $\mathcal{S}_{up}$ or speed-up $\mathcal{S}_{down}$. However, the infinite-queue model falls short for the case of system S, which is an *open* system, i.e. designers have to specify and implement S without knowing anything about the environment in which S operates (but for the fact that it understands the latency-insensitive protocol). This is the case of the previous example assuming that designers cannot modify the structure of $\mathcal{S}_{up}$, while they can only use the latency-insensitive protocol to require $\mathcal{S}_{up}$ to stall. The model presented in the next section makes it possible to capture this case because it handles the back-pressure mechanism.

5.2.4 Performance Analysis with the Finite-Queue Model

The second constructive model replaces the infinite-queue assumption with two "weaker" assumptions:

1. the shells have limited storage space in their input queues; specifically, each input queue of every shell in the system has a length equal to K, where K is a finite natural number;

2. the shells communicate by means of channels that implement a latency-insensitive protocol with back-pressure (like the one discussed in Section 4.2).

This model is called the *K-finite-queue model*, or simply *finite-queue model* when the value of K is clear from the context.

The right-hand side of Figure 5.12 illustrates two primitive marked graphs from the library $\mathcal{L}_{2FQ}$ of the 2-finite-queue model. These primitive can be compared with the prim-

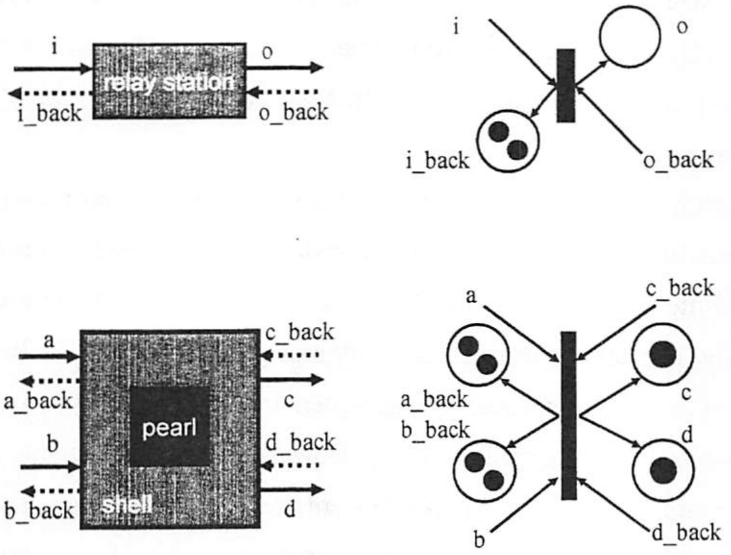

Figure 5.12: Primitive marked graphs from the 2-finite-queue model library $\mathcal{L}_{2FQ}$.

itives of library $\mathcal{L}_{IQ}$ in Figure 5.6 which model the same components (a relay station and two-input two-output shell) but assuming infinite-length queues. The visible difference is only in the presence of additional arcs and places to represent the back-pressure mechanism.

A relay station is modeled by a transition with two incoming arcs, two outgoing arcs, an initially empty place, and a place that is initialized with two tokens. The initially empty place fulfills the same two roles as before, i.e. it captures both the storage capability in the forward direction and the stalling symbol as the initialization value within the relay station. The additional place together with the additional incoming arc and outgoing arc are necessary to model back-pressure. In particular, the place initialized with two tokens represents the storage capability of the relay station, which is equal to two [6].

A shell-core pair with m input channels and n output channels is again modeled as a transition, but this time it has twice the number of arcs and places. More precisely, it

[6]Strictly speaking, this model is actually the *correct* model of a relay station, because a relay station is formally defined as a buffer of capacity equal to two (Definition 3.19). Instead, it is more proper to say that the model of Figure 5.6 represents just a sequential buffer, a stateful repeater, of potentially infinite capacity.

has $m+n$ incoming arcs and $m+n$ outgoing arcs because there is an additional arc for each channel to express back-pressure. Specifically, there are n outgoing arcs linked to *forward-output places* that contain one token each and m outgoing arcs linked to *backward-output places* that contain two tokens each. The main difference is obviously given by the backward-output places, whose initial marking represents the fact that all input queues of this shell have a finite length equal to two. Different queue lengths can be captured straightforwardly by changing the number of tokens in the initial marking.

The finite-queue model uses tokens for a twofold purpose: a token traversing the marked graph either represents an informative event or the availability of storage space (in a relay station or a shell queue) for buffering an incoming informative event. Dually, the lack of a token in a place represents either the absence of an informative event (i.e. the presence of a stalling event) or the lack of available storage space (the relay station or the queue is filled). However, it is still reasonable to interpret simply a token as an informative event and the absence of tokens in a place as a stalling event. In fact, this is in complete harmony with the formalization of the theory of latency-insensitive protocols given in Chapter 3, where no distinction was made between "forward stalling" and "backward stalling" and the emphasis was all on distinguishing informative events from stalling events [7].

The shell-core pair modeled by the primitive marked graph of Figure 5.12 is the two-input two-output shell of Figure 4.2 having the following characteristics:

- the shell has input queues with finite length (equal to two in this particular case);
- the shell only stores informative events in the input queues;
- the input queue is "by-passable" whenever two conditions hold: (1) it is empty and (2) the shell does not need to stall its core. In this case an incoming informative event is immediately passed through to the core for being processed.

[7] In particular, recall that the buffers and relay stations were formally defined in Section 3.3 without even specifying the direction of their signals, i.e. without assuming that they are functional processes. The distinction between *void flag* signal and *stop flag* signal was introduced as part of the refinement step that produces the implementation of Section 4.2.

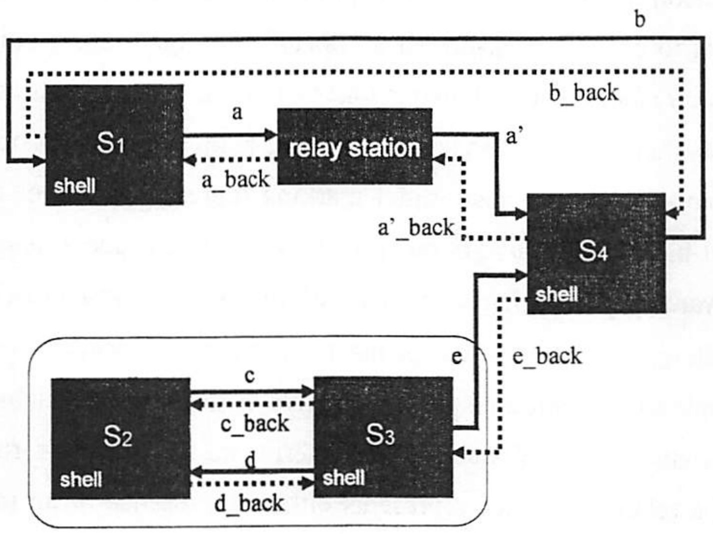

Figure 5.13: Finite-queue model for the latency-insensitive system of Figure 5.9.

Definition 5.13 of channel is modified as follow in the K-finite-queue model to account for the back-pressure mechanism.

Definition 5.15 *A* channel with back-pressure q *of a* marked graph $\mathcal{MG} = (P,T,F,W,M_0)$ *is a directed elementary path* $\{t_0, p_0, t_1, p_1, \dots, t_{n-1}, p_{n-1}, t_n\}$ *together with a set of places* $P_{bp} = \{p'_1, p'_2, \dots, p'_{n-1}, p'_n\}$ *such that* $\forall n \in \mathbb{N}$, $\forall i \in [1, n-1]$ *and* $\forall K \in \mathbb{N}$

$$\begin{aligned} M(p_0) &= 1 \\ \forall p \in\ & t_n\bullet \ \Big(M(p) = 1\Big) \\ M(p_i) &= 0 \\ M(p'_i) &= K \\ |t_i \bullet| &= 2 \\ |\bullet t_i| &= 2 \end{aligned}$$

The elements of P_{bp} *are the* backward places. *Places* $p_0, \dots, p_{n-1}$ *are the* forward places.

The definitions given for a channel naturally extends to a channel with back-pressure. A marked-graph channel with back-pressure models precisely the channel of the latency-insensitive system defined in Section 4.2.1.

Example The bottom diagram of Figure 5.7 illustrates a marked-graph channel with back-pressure q. This channel has latency $\lambda(q) = 8$. ■

An K-finite-queue marked-graph model is a marked graph where no place has initial marking greater than K, each initially empty place is a forward place of a channel with back-pressure and each place with initial marking greater than 1 is the backward place of a channel with back-pressure.

Definition 5.16 *A* K-finite-queue marked-graph model *of a latency-insensitive system implementation S is the marked graph constructed through the following steps:*

1. *build an an infinite-queue marked-graph model $\mathcal{MG}$ of S;*

2. *transform every channel $q \in \mathcal{MG}$ into a channel with back-pressure through the addition of backward places having initial marking equal to K.*

Hence, with respect to library $\mathcal{L}_{KFQ}$, each internal transition of a channel together with its backward and forward output places corresponds to a primitive marked graph for a relay station, while each of the remaining transitions corresponds to a primitive marked graph for a shell.

Example Figure 5.13 shows a latency-insensitive system implementation based on back-pressure. This implementation is equivalent to the one of Figure 5.9, which does not use back-pressure. Figure 5.14 shows a marked graph $\mathcal{MG}$ that models the implementation of Figure 5.13. and is built using the primitives from library $\mathcal{L}_{2FQ}$ illustrated in Figure 5.12. The marking in Figure 5.14 is the initial marking. Observe the presence of the backward output places that contain two tokens each in the initial marking. Overall, $\mathcal{MG}$ has five

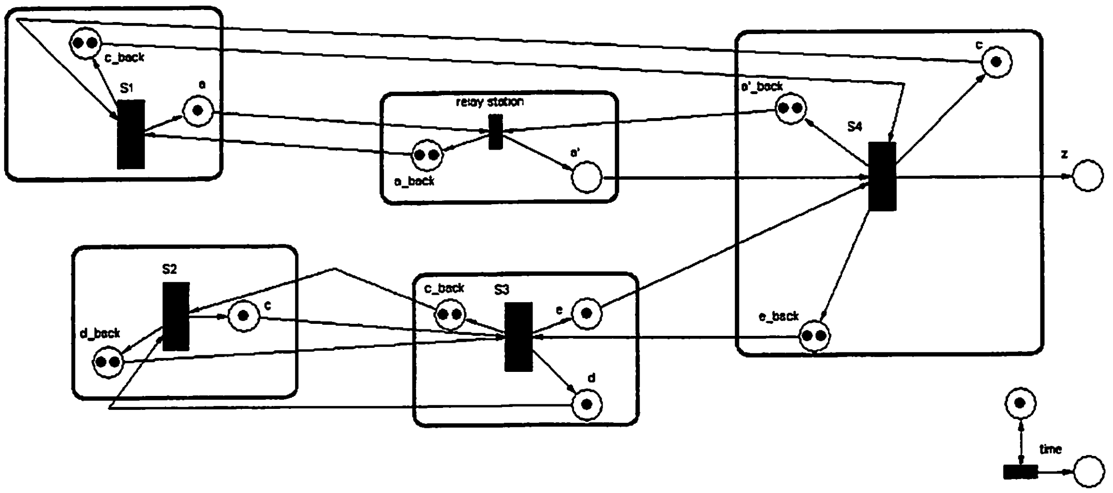

Figure 5.14: Finite-queue model for the latency-insensitive system of Figure 5.13 (queues length = 2).

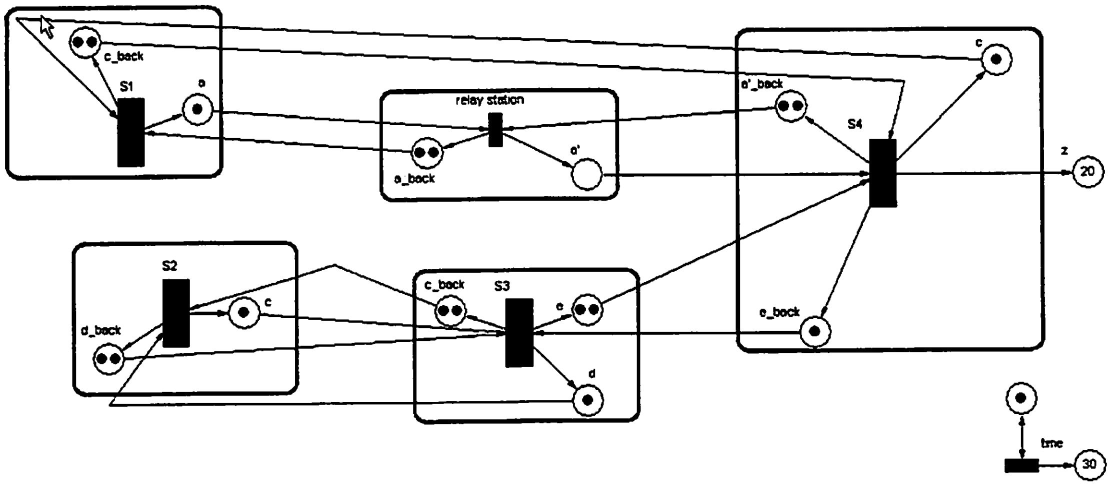

Figure 5.15: The marked graph of Figure 5.14 after 30 timestamps.

channels:

$$\begin{aligned}
q_{\{S_1,S_4\}} &= \{S_1, a, RS, a', S_4\} \quad \text{with} \quad \lambda(q_{\{S_1,S_4\}}) = 2 \\
q_{\{S_4,S_1\}} &= \{S_4, b, S_1\} \quad \text{with} \quad \lambda(q_{\{S_4,S_1\}}) = 1 \\
q_{\{S_2,S_3\}} &= \{S_2, c, S_3\} \quad \text{with} \quad \lambda(q_{\{S_2,S_3\}}) = 1 \\
q_{\{S_3,S_2\}} &= \{S_3, d, S_2\} \quad \text{with} \quad \lambda(q_{\{S_3,S_2\}}) = 1 \\
q_{\{S_3,S_4\}} &= \{S_3, e, S_4\} \quad \text{with} \quad \lambda(q_{\{S_3,S_4\}}) = 1
\end{aligned}$$

Compared to the marked graph of Figure 5.10, which models the same system but with the primitives from library $\mathcal{L}_{IQ}$, $\mathcal{MG}$ contains the same two "global" cycles ($\{S_1, RS, S_4\}$, and $\{S_2, S_3\}$) plus several additional cycles due to the insertion of back-pressure. Of these eight are elementary cycles [8]: two cycles mirror the original global cycles (mirror cycles) and six are "local cycles" corresponding to the four un-pipelined channels and the two sub-channels composing the pipelined channel (local back-pressure cycles). Figure 5.15 shows the marking of $\mathcal{MG}$ after 30 timestamps have passed by from the initial marking. Observe that there is no unbounded accumulation of tokens at place e between shell S_3 and shell S_4. The reason is precisely the back-pressure mechanism which, in marked-graph terms, forces the number of tokens in the cycle around transition S_3 and S_4 to remain constant (equal to three). ■

The following theorem states an important result: given a latency-insensitive system S, an implementation based on back-pressure and input queues of length equal or greater that two can sustain the same maximum throughput as a implementation with infinite queues and no back-pressure. Naturally, the importance of this results consists of its practical consequences as infinite queues cannot be physically built, while a back-pressure mechanism with finite queues is easy to implement.

Theorem 5.8 *Given a latency-insensitive system S, the marked-graph model $\mathcal{MG}_S^K$ built using any K-finite-queue model library $\mathcal{L}_{KFQ}$ with $K \geq 2$ has maximum sustainable through-*

[8]As discussed in Section 5.3.1, to compute the cycle time of a strongly connected marked graph is sufficient to consider only the elementary cycles.

put $\vartheta(\mathcal{M}\mathcal{G}_S^K)$ equal to the maximum sustainable throughput $\vartheta(\mathcal{M}\mathcal{G}_S^\infty)$ of the marked-graph model $\mathcal{M}\mathcal{G}_S^\infty$ of S that is built using the infinite-queue model library $\mathcal{L}_{IQ}$.

Proof. Since both $\mathcal{M}\mathcal{G}_S^K$ and $\mathcal{M}\mathcal{G}_S^\infty$ model the same system the former can be obtained from the latter by inserting r backward-output places (together with the corresponding $2 \cdot r$ arcs) on its channels. This operation creates new cycles. Let C be the set of cycles of $\mathcal{M}\mathcal{G}_S^\infty$. Then, the set of cycles C' of $\mathcal{M}\mathcal{G}_S^K$ contains the same $|C|$ cycles plus $|C|+r$ new elementary cycles. Each of the additional $|C|$ elementary cycles is a cycle m_c mirroring a given cycle c in C. Recalling Definition 5.8 of cycle metric, it is easy to see that $\forall c \in C(\mu(c) < \mu(m_c))$ because c and m_c have the same number of transitions and places, but each place in m_c contains two tokens while the places of c contain either one or zero tokens. Each of the r additional cycles is a *local back-pressure cycle* $g = \{t_a, p_a, t_b, p_b\}$ with two transitions and two places. Transition t_a and t_b may model either a relay station or a shell-core pair. Hence, there are four possible combinations to consider:

- if t_a models a shell-core pair and t_b a relay station then $\mu(g) = \frac{2}{K+1}$;
- if t_a models a relay station and t_b a shell-core pair then $\mu(g) = \frac{2}{K}$;
- if both t_a and t_b model shell-core pairs then $\mu(g) = \frac{2}{K+1}$;
- if both t_a and t_b model relay stations then $\mu(g) = \frac{2}{K}$.

Recalling the hypothesis $K \geq 2$, any local back-pressure cycle g has cycle metric $\mu(g) \leq 1$. Therefore, if $|C| > 1$, i.e. $\mathcal{M}\mathcal{G}_S^\infty$ contains at least one cycle, then a cycle $c^\star$ in C has the largest cycle metric $\mu(c^\star)$ among all cycles of $\mathcal{M}\mathcal{G}_S^K$. From Definition 5.12 of maximum sustainable throughput and Theorem 5.6 it follows that $\vartheta(\mathcal{M}\mathcal{G}_S^K) = \vartheta(\mathcal{M}\mathcal{G}_S^\infty) = \frac{1}{\mu(c^\star)}$.

Finally, if $|C| = 0$, i.e. if $\mathcal{M}\mathcal{G}_S^\infty$ is acyclic, then $\vartheta(\mathcal{M}\mathcal{G}_S^\infty) = 1$ by Definition 5.12. Instead, $\mathcal{M}\mathcal{G}_S^K$ is strongly connected. Its elementary cycles, however, are only mirror cycles g with cycle metric $\mu(g) \leq 1$ and, therefore, by Definition 5.12, also $\vartheta(\mathcal{M}\mathcal{G}_S^K) = 1$. □

An important design guideline on the sizing of the shell input queues follows from the previous result: to increase the lengths of the queues beyond two slots is not useful when using back-pressure. On the other hand, the following result confirms that the *"number two"* plays a special role in latency-insensitive design [9]. In fact, an implementation of S

[9]Recall Lemma 3.8 which motivates Definition 3.19 of relay station.

based on back-pressure and finite queue of length *one* is functionally correct, but is not able to offer the same performance.

Corollary 5.9 *Given a latency-insensitive system S with at least one relay station, the marked-graph model $\mathcal{MG}_S^1$ built using a* 1*-finite-queue model library $\mathcal{L}_{1FQ}$ has maximum sustainable throughput not greater than* 0.5.

Proof. Following the same lines of Theorem 5.8, it is sufficient to focus on the local back-pressure cycle $g = \{t_a, p_a, t_b, p_b\}$ where t_a models a relay station and t_b a shell-core pair. This cycle has cycle metric $\mu(g) = \frac{2}{K} = \frac{2}{1}$. Therefore, the maximum sustainable throughput $\vartheta(\mathcal{MG}_S^1)$ of $\mathcal{MG}_S^1$ is capped at 0.5 by the reciprocal of $\mu(g)$. □

Example Figures 5.16 shows a marked graph modeling an implementation of the system of Figure 5.13 that is built using the primitives from library $\mathcal{L}_{1FQ}$. Compare the marked graph with the one in Figure 5.14 that models an implementation of the same system but it is built using the primitives from library $\mathcal{L}_{2FQ}$. The structure of the marked graph is obviously the same, but the initial marking is different: the backward output places contain only one token to represent the reduced storage capacity [10]. Figure 5.17 shows the marking for the same marked graph after 30 timestamps have passed by from the initial marking of Figures 5.16. Not surprisingly, the system has been able to process only 15 tokens during an interval of 30 time units. ■

I conclude this section proving liveness and boundedness for marked-graph models built using library $\mathcal{L}_{KFQ}$. The following result guarantees that given a latency-insensitive system S, any implementation based on back-pressure and finite queues does not deadlock.

Theorem 5.10 *The* K-finite-queue marked-graph model *$\mathcal{MG}_S$ of a latency-insensitive system implementation S is live.*

Proof. It is necessary to prove that there are no cycles of initially empty places. This naturally follows from the observation that marked graph $\mathcal{MG}_S$ is obtained from an infinite-queue marked-graph model $\mathcal{MG}_S^\infty$ of S by adding some backward places having initial

[10]Here the relay station are replaced with the buffers $B_{1,1}^1(i,j)$ defined in Section 3.3.2. As discussed in Section 3.3.4 these buffers can only sustain a throughput equal to 0.5. This, however, is the maximum throughput sustainable by the whole implementation as guaranteed by Corollary 5.9.

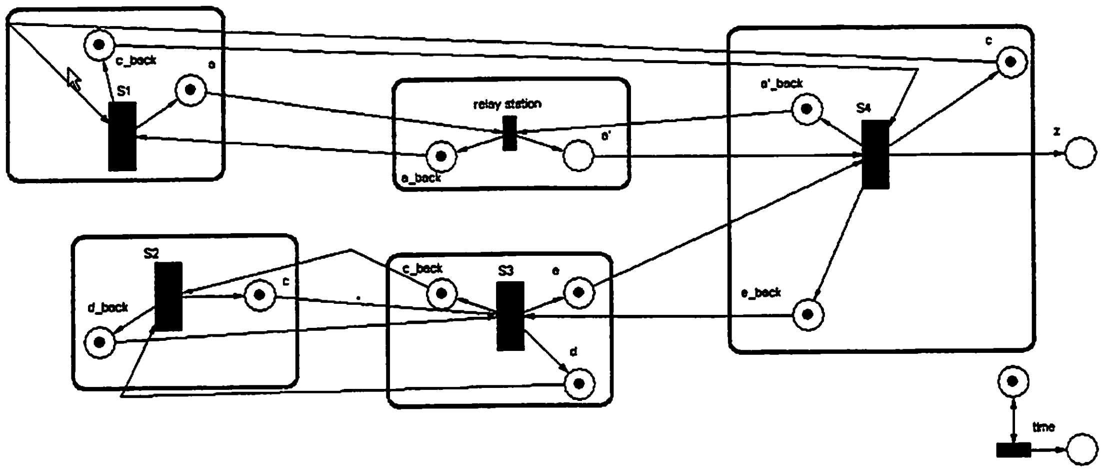

Figure 5.16: Finite-queue model for the latency-insensitive system of Figure 5.13 (queues length = 1).

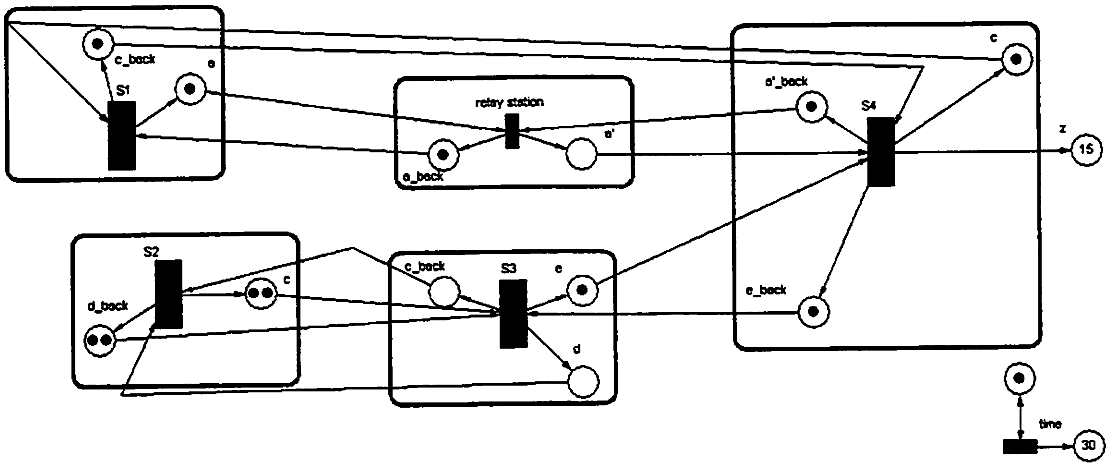

Figure 5.17: The marked graph of Figure 5.16 after 30 timestamps.

marking equal to $K \in \mathbb{N}$ (Definition 5.16). Since $\mathcal{MG}_S^\infty$ is always live (by Theorem 5.7), $\mathcal{MG}_S$ is also live. In fact, the simple insertion of places that are not initially empty cannot create a cycle of initially empty places and, therefore, the preservation of the liveness condition of Theorem 5.1 is guaranteed. □

The following result confirms that the introduction of back-pressure removes the need for the infinite queue assumption. Given a latency-insensitive system S, any implementation based on back-pressure and finite queues is guaranteed not to overflow.

Theorem 5.11 *The* K-finite-queue marked-graph model $\mathcal{MG}_S$ *of a latency-insensitive system implementation* S *is* $(K+1)$*-bounded.*

Proof. The boundedness of $\mathcal{MG}_S$ follows directly from Theorem 5.2 and the observation that every place in marked graph $\mathcal{MG}_S$ belongs to at least one cycle. In fact, each place corresponds to a distinct local back-pressure cycle $g = \{t_a, p_a, t_b, p_b\}$. Therefore, to compute the maximum number of tokens that can accumulate at each place, it is sufficient to consider the four possible scenarios for a channel cycle g as discussed in the proof of Theorem 5.8:

- if t_a models a shell-core pair and t_b a relay station then all places on g are $K+1$-bounded;
- if t_a models a relay station and t_b a shell-core pair then all places on g are K-bounded;
- if both t_a and t_b model shell-core pairs then all places on g are $K+1$-bounded;
- if both t_a and t_b model relay stations then all places on g are K-bounded.

Hence, $\mathcal{MG}_S$ is $(K+1)$-bounded. □

The previous result is consistent with the physical reality of an implementation such as the reference implementation discussed in Section 4.2, which is based on back-pressure and queues of length two. The corresponding marked-graph model is 3-bounded and three is precisely the number of back-to-back storage elements that are present on a connection between two shells or from one shell to a relay station (this number goes up to four on a connection between two relay stations or from a relay station to a shell).

5.3 Related Work

I summarize here the previous work that is most related to the modeling techniques discussed in this chapter. First, I compare the concepts of cycle time, maximum profit-to-time ratio, and maximum cycle mean and I clarify why algorithms solving the maximum cycle mean problem can be used to compute the throughput of a latency-insensitive system. Then, I quickly survey previous literature on AND/OR causality as well as on the use of Petri nets to model asynchronous systems and analyze their performance. Finally, I explain the relationships between marked graphs and synchronous data flows.

5.3.1 Maximum Profit-To-Time Ratio and Maximum Cycle Mean

The cycle time of a timed marked graph is a concept similar to the *maximum profit-to-time ratio* [140]. Given a doubly weighted cyclic directed graph G, the profit-to-time ratio of a cycle $c \in G$ is given by the sum of the weights $w(a_i)$ of each arc a_i of c divided by the sum of the "times" $d(a_i)$ of each of these arcs. Hence, to solve the maximum profit-to-time ratio problem amounts to find a *critical cycle c* whose ratio coincides with

$$\max_{\forall c \in G} \left(\frac{w(c)}{d(c)} \right)$$

where $w(c)$ is the sum of the weights of the arcs of cycle c, and $d(c)$ is the sum of their delays. Notice that it is sufficient to consider only the elementary cycles of G, i.e. cycles where no vertex is repeated. In [140], Lawler provides an $O(|V| \cdot |A| \cdot logB)$ algorithm to solve this problem where V is the set of vertices of G, A is the set of arcs, and B is a variable related to the desired precision of the results.

As discussed in [71], the maximum cycle mean problem is a special case of the *maximum profit-to-time ratio* problem. In fact, the *maximum cycle mean* of a weighted cyclic directed graph G is defined as

$$\max_{\forall c \in G} \left(\frac{w(c)}{|c|} \right)$$

where $w(c)$ is the sum of the weights of the arcs of cycle c, and $|c|$ is equal to the number of arcs of c. Therefore, if $\forall a_i \in c\big(d(a_i) = 1\big)$, then the profit-to-time ratio of c coincides with the cycle mean of c. In 1978 Karp published an elegant theorem for calculating the maximum cycle mean together with a companion algorithm of complexity $O(|V| \cdot |A|)$ [127]. Since then, several other algorithms have been proposed to solve the *maximum cycle mean problem* as surveyed in [71, 101]. Relationships with the dual optimization problems, i.e. finding the minimum cycle mean and the minimum cost-to-time ratio, are discussed in [71].

In modeling latency-insensitive systems with marked graphs each transition is assumed to have unit delay. Hence, operationally, one can take a marked graph $\mathcal{MG}$ and build a corresponding weighted directed graph G by transforming each transition into a vertex, each place into an arc, and labelling each arc with a weight equal to the number of tokens contained in the corresponding place (for any given marking). Then, computing the maximum cycle mean of G returns also the cycle time of $\mathcal{MG}$.

5.3.2 AND/OR Causality in Modeling Discrete Event Systems

Discrete event systems with maximum and minimum constraints are a pervasive model in control theory, computer science, and operation research. A maximum constraint on a given signal s corresponds to an AND-causality relation: a new event of s can occur only after a new event for each signal on which s depends has occurred. Instead, a minimum constraint corresponds to an OR causality relation where one single event is sufficient to trigger a new event of s. If a discrete event system can be modeled using only maximum (or, dually, minimum) constraints then it can be studied by linear methods based on *max-plus* algebra [6]. In [97] Gunawardena offers a brief historical account of the main works on timed systems with maximum constraints before introducing the model of *timed* AND,OR *automata* as a framework to study systems with mixed constraints, which are also the subject of [98]. Interestingly enough, Gunawardena reveals that the original motivation of his work is to extend the performance analysis of asynchronous circuits by Burns [32] to accommodate system with "disjunctive behavior", but that some of his results can also be applied to the clock schedule verification problem for synchronous circuits [203, 227]. Gunawardena's conclusive comment is: "it is amusing that the same underlying mathematics can be applied to study timing behavior of both synchronous and asynchronous hardware."

5.3.3 Performance Analysis of Asynchronous Systems Using Petri Nets

Ramchandani is the first to apply timed Petri nets to the analysis of asynchronous concurrent systems [195]. Williams proposes two types of marked graphs, called respectively *dependency graphs* and *folded graphs*, as a specialized model targeting the efficient computation of the exact throughput and latency for deterministic self-timed pipelines [244]. To model asynchronous circuits Burns introduces event-rule systems that are equivalent to marked graphs. He defines the cycle period of an asynchronous system as the asymptotic average time separation between consecutive occurrences of the same event. To compute it, he proposes an efficient algorithm based on the primal-dual technique for solving a special linear programming problem [31, 32, 30]. Lee extends Burns' models to accommodate OR-causality [146]. Xie and Beerel propose symbolic techniques to analyze the performance of asynchronous systems with non-fixed delays (due to data-dependency or random environment requests) [245]. More recently, they have presented a new approach based on stochastic timed Petri nets in a paper [247] that includes also a review of the various efforts on performance analysis of asynchronous circuits. Finally, the collection edited by Yakovlev *et al.* offers a summary of the state of the art in the application of Petri nets to the design of digital systems and circuits [249].

5.3.4 Performance Analysis of Embedded Systems

In [161], Mathur *et al.* address the problem of analyzing the performance of an embedded system obtained as a composition of processes. For each process, they assume to have constraints on its execution rate, which is defined as the number of executions per unit time. These constraints are either imposed by the designers or by the environment in which the system operates. After defining a process graph to capture the structure of the system, they propose an interactive rate analysis framework to compute bounds on the execution rate of each process. Their main algorithm is based on the relationship between the execution rate of each process and the maximum cycle mean in the process graph.

5.3.5 Marked Graphs versus Data Flow Models

Being ordinary Petri nets where each place has exactly one input transition and exactly one output transition, marked graphs are a model of computation equivalent to *homogeneous synchronous data flows* [143]. Synchronous data flows [18, 143] are a restricted version of data flow models of computation originally pioneered by Dennis [75] and are closely related to the *computation graphs* proposed by Karp and Miller in 1966 [128].

In a data flow model, a program is represented as a directed graph in which vertices, called actors, represent functional computations and arcs represent FIFO channels that queue data values (represented by tokens). In a synchronous data flow (SDF) the number of tokens produced (consumed) by the firing of an actor v is a quantity $k(a_i)$ that is constant [11] for each input (output) arc a_i of v. An SDF is homogeneous when $\forall i(k(a_i) = 1)$.

Marked graphs are equivalent to homogeneous SDFs. Therefore, they can be seen as a restricted version of SDFs that cannot support different sample rates. In fact, SDFs are better suited to the modeling of mixed-grain data-flow programming of multi-rate digital signal processing (DSP) systems. In their seminal work [142, 143, 145], Lee and Messerschmitt prove important properties of SDFs and provide efficient techniques to determine statically (at compile time) whether an arbitrary SDF graph is live and bounded. When this is the case, it is possible to build a periodic schedule that becomes the basis for a bounded-memory implementation that is guaranteed not to deadlock (*static scheduling*). Such schedules can be either sequential or parallel depending on whether the target architecture is a single processors or a multiprocessor system. In general, there are many possible valid schedules that can be built. An interesting optimization criterion is to find the schedule that minimize the amount of storage space on the FIFO channels between the actors (*minimum buffer schedule*). For each channel such amount corresponds to the maximum number of tokens that are queued during one period of the schedule. However, in general it is necessary to rely on heuristics because the *minimum buffer schedule problem* is proven to be NP-complete [88] even for an arbitrary homogeneous SDF [177, 18].

I conclude mentioning Boolean data flows (BDFs), originally defined by Lee [141]

[11] Being constant these quantities are *statically* known at compile time. Therefore, some researchers prefer to interpret the acronym SDF as *static data flow*.

and then studied by Buck [28]. BDFs are an extension of SDFs where the number of tokens produced/consumed by each actor is either fixed (as for SDFs) or a function of a boolean-valued token that is produced/consumed by the actor. This difference is sufficient to make the BDF model *Turing complete* and the problem of finding a bounded-memory implementation of a BDF model undecidable [115].

5.4 Concluding Remarks

I presented a modeling approach for the quantitative analysis of the performance of a latency-insensitive system. The approach is based on the "marked graphs" model of computation and exploits their many properties. Specifically, I proposed two constructive models: the infinite-queue model and the finite-queue model. Each of them uses a simple set of building blocks and the compositional properties of patient processes to construct a marked graph representing the given latency-insensitive system. The two models are different because they express alternative system implementations which are based on different design styles for the shells. The infinite-queue model is useful for designing closed latency-insensitive systems because it can be used to check their boundedness and statically size the shell queues to guarantee correct operations without the need for back-pressure. The finite-queue model addresses implementation of latency-insensitive protocols based on back-pressure and can handle a system operating in an environment that may be stalled as well as may require the system to stall. Both models can be used to compute exactly and statically the highest throughput that the system can sustain. For a given latency-insensitive system, this quantity turns out to be the same in both cases because it is independent from the shell implementation style and only depends on where (and in which number) the relay stations have been inserted. Indeed, I demonstrate that a physical implementation based on back-pressure and finite queue of length two is able to sustain the same throughput as a virtual implementation with infinite queues. In summary, constructive models make it possible to prove that a latency-insensitive system is live by-construction and to determine the impact on the system performance due to the insertion of relay stations on the communication channels between shells. How to optimize this impact is the subject of Chapter 6.

Chapter 6

Performance Optimization

In which a balanced quest for the top is restlessly sought.

No matter how many relay stations are introduced on the channels of a latency-insensitive system, its functional correctness is guaranteed to be preserved: the system may produce more stalling events on the output channels as well as exercise more back-pressure on the input channels but, nevertheless, its processing activity progresses without deadlocking. Naturally, however, the effectiveness of latency-insensitive design is strongly related to the ability of maintaining a sufficient communication throughput in the presence of increased channel latencies. In this chapter I discuss the performance optimization of latency-insensitive systems. I explain the role played by the system topology with respect to channel pipelining, the factors that influence the decision of whether to use back-pressure or not, and the important trade-offs related to shell encapsulation. I provide design guidelines to optimize the system throughput and I introduce the concept of recycling. By combining three operations (inserting relay stations, moving them across shell-core pairs, and redrawing the boundaries of the shells around the cores), recycling enables the exploration of latency/throughput trade-offs in order to achieve the right balance between communication and computation latencies.

The content of this chapter was partially presented at the 36th Design Automation Conference [38] based on an ad-hoc formalism (called *lis-graphs*, i.e. latency-insensitive system graphs), which turned out to be equivalent to marked graphs and, therefore, unable to add real value.

6.1 The Global Impact of Channel Pipelining

Using the marked-graph models introduced in Section 5.2, I formalize the operation of wire pipelining and explain the dramatic impact that the local insertion of relay stations can have on the global behavior of a latency-insensitive system. In this regard, I discuss the role played by the topology of the system and the factors to consider when making a decision whether to base the system implementation on back-pressure or not. I conclude the section providing general design guidelines to minimize the impact of relay station insertion and I illustrate them with a case study.

6.1.1 Channel Pipelining

In order to formalize the practice of wire pipelining, it is convenient to extend to marked graph models the notion of latency-equivalence that was given in Section 3.2.3, Two marked graph models are latency equivalent when they only differ for the latency of their channels.

Definition 6.1 *A* reference marked graph model $\mathcal{MG}_{ref} = (P, T, F, W, M_0)$ *is a marked graph model such that*

$$\forall q \in Q(\mathcal{MG}_{ref}) \left(\lambda(q) = 1\right)$$

Hence, from Definition 5.13 of channel latency it follows that the reference marked-graph model does not have any internal transition in its channels. In fact, $\mathcal{MG}_{ref}$ models a strict latency-insensitive system S_{strict}, i.e. a system without relay stations, while each member of its latency-equivalence class models a patient system S.

Definition 6.2 *Two marked graph models* $\mathcal{MG}' = (P', T', F', W', M_0')$ *and* $\mathcal{MG}' = (P'', T'', F'', W'', M_0'')$, *are* latency equivalent $\left(\mathcal{MG}' \equiv_\tau \mathcal{MG}''\right)$ *when they can be derived from the same reference model* $\mathcal{MG}_{ref}$ *by increasing the latency of the channels in* $Q(\mathcal{MG}_{ref})$.

Hence, the channels of latency-equivalent marked-graph models correspond pairwise, i.e.

$$\forall q' \in Q(\mathcal{MG}') \ \exists q'' \in Q(\mathcal{MG}'') \ \left(head(q') = head(q'') \ \wedge \ tail(q') = tail(q'')\right)$$

and vice versa.

Given the specification of a stallable sequential system a latency-equivalent implementation S can be derived following the five main steps discussed in Section 4.1.1. The present section discusses the formalization of the last step, i.e. the post-layout optimization performed by means of channel pipelining. Channel pipelining can be performed separately on each critical channel in the implementation S. A channel is critical when some of the wires that implement it on the final layout have a delay higher than the period ψ of the nominal clock of S. From the analysis of the layout it is possible to determine for each wire what is the smallest multiple of ψ that is larger than its delay. If this multiple is greater than one then the wire represents a *design exception* that needs to be fixed. This reasoning justifies the following definitions of annotated marked-graph model and critical channel.

Definition 6.3 *A marked-graph model $\mathcal{MG}_S = (P,T,F,W,M_0)$ of a latency-insensitive system implementation S with nominal clock period ψ is* annotated *when a* normalized delay *$\gamma(q,\psi)$ is associated to each channel $q \in Q(\mathcal{MG})$ with*

$$\gamma(q,\psi) = \left\lceil \frac{delay(q)}{\psi} \right\rceil$$

where $delay(q)$ is the timing delay of the slowest among the wires of S implementing q.

The quantity $\gamma(q,\psi)$ expresses the delay of a channel in clock-cycle units.

Definition 6.4 *An annotated marked-graph model $\mathcal{MG}_S = (P,T,F,W,M_0)$ of a latency-insensitive system implementation S with respect to a nominal clock period ψ, is* legal *when*

$$\forall q \in Q(\mathcal{MG}) \left(\gamma(q,\psi) \leq \lambda(q)\right)$$

A channel $q \in Q(\mathcal{MG}_S)$ such that $\gamma(q,\psi) > \lambda(q)$ is a critical channel *of $\mathcal{MG}_S$.*

Hence, each critical channel of $\mathcal{MG}_S$ captures the presence in S of a wire that connects the I/O ports of two components (relay stations or shells) and has a delay larger than the nominal clock period ψ. The latency insensitive methodology makes it easy and systematic to fix these design exceptions: each critical wire is pipelined with the insertion of the necessary number of relay stations in order to divide it in segments whose delay is smaller than ψ.

The third step of the design flow described in Section 4.1.1 dictates that all the wires connecting one module with another module be grouped into a point-to-point channel. The *channeling* operation is motivated by the practical idea of "keeping close" those wires that must span the same distance while deriving the layout of S. This strategy minimizes the delay variations among these wires and makes it possible to uniformly pipeline them based on the delay of the slowest among them [1].

Hence, pipelining a wire in S corresponds to increasing the latency of the corresponding channel in $\mathcal{M}\mathcal{G}_S$ such that $\gamma(q,\psi) \leq \lambda(q)$. Likewise, at the system-level, the non-legal annotated marked-graph model $\mathcal{M}\mathcal{G}_S$ is transformed into a latency-equivalent, legal, annotated marked-graph model $\mathcal{M}\mathcal{G}'_S$ by simply incrementing the latency of its channels by the necessary quantity. Since $\mathcal{M}\mathcal{G}'_S \equiv \mathcal{M}\mathcal{G}_S$, for all behaviors of $\mathcal{M}\mathcal{G}_S$ there is a correspondent equivalent behavior of $\mathcal{M}\mathcal{G}'_S$. This transformation is called *channel pipelining*.

Lemma 6.1 Channel pipelining. *Let $\mathcal{M}\mathcal{G}_S$ be an non-legal annotated marked-graph model of a latency-insensitive system implementation S with respect to a nominal clock period ψ. A latency-equivalent legal annotated marked-graph model $\mathcal{M}\mathcal{G}'_S$ is obtained from $\mathcal{M}\mathcal{G}_S$ by inserting $\Delta\lambda(q)$ internal transitions on each channel $q \in Q(\mathcal{M}\mathcal{G}_S)$ with $\Delta\lambda(q) = \gamma(q,\psi) - \lambda(q)$.*

Proof. Obviously, with the insertion of the internal internal transitions,

$$\forall q \in Q(\mathcal{M}\mathcal{G}'_S)\ \Big(\lambda'(q) \ = \ \lambda(q) + \Delta\lambda(q) \ = \ \gamma(q,\psi)\Big)$$

Therefore, by Definition 6.4 $\mathcal{M}\mathcal{G}'_S$ is legal. Further, by Definition 6.2, $\mathcal{M}\mathcal{G}'_S \equiv_\tau \mathcal{M}\mathcal{G}_S$ since $\mathcal{M}\mathcal{G}_S$ and $\mathcal{M}\mathcal{G}'_S$ differ only for their channel latencies. □

Although channel pipelining is an easy way to correct the final implementation of system S and satisfy the timing constraints imposed by the nominal clock ψ, it does not always come without a cost. In fact, augmenting the latency of some channels of $\mathcal{M}\mathcal{G}_S$ (i.e., inserting relay stations on the communication channels of S) may increase the cycle time $\pi(\mathcal{M}\mathcal{G}_S)$ and, reciprocally, decrease the throughput $\vartheta(\mathcal{M}\mathcal{G}_S) = \vartheta(S)$. This is always the

[1]Naturally, if the wires between two modules are laid out separately, they can be considered as separate channels and the latency-insensitive design methodology can still be applied. However, this makes the design implementation less modular and complicates its optimization: it may be more difficult to maintain high throughput and the area overhead of the shells increases.

case if the critical channel q that needs to be pipelined belongs to a critical cycle [2] of $\mathcal{M}\mathcal{G}_S$ simply because the numerator of the cycle metric ratio $\mu(c)$ increases by the quantity $\Delta\lambda(q)$ while the denominator remains unaffected (Definition 5.8). It may also happen that inserting internal transitions on some channels transforms a non-critical cycle of $\mathcal{M}\mathcal{G}_S$ into a critical cycle of $\mathcal{M}\mathcal{G}'_S$. In any case, the impact of any channel pipelining transformation on the system throughput can be easily pre-assessed by calculating the consequent *throughput degradation* $\Delta\vartheta(\mathcal{M}\mathcal{G}_S, \mathcal{M}\mathcal{G}'_S) = \vartheta(\mathcal{M}\mathcal{G}_S) - \vartheta(\mathcal{M}\mathcal{G}'_S)$, using one of the following methods:

- find the cycle time $\pi(\mathcal{M}\mathcal{G}'_S)$ of $\mathcal{M}\mathcal{G}'_S$, with one of the algorithms discussed in Section 5.3.1 and simply calculate $\Delta\vartheta(\mathcal{M}\mathcal{G}_S, \mathcal{M}\mathcal{G}'_S) = \vartheta(\mathcal{M}\mathcal{G}_S) - \frac{1}{\pi(\mathcal{M}\mathcal{G}'_S)}$;
- determine the set C_{crit} of cycles of $\mathcal{M}\mathcal{G}_S$ containing at least one critical channel q_i that has to be pipelined. Then, increment the cycle metric $\mu(c)$ for each $c \in C_{crit}$ by the quantity $\frac{1}{|c|} \cdot \sum_i \Delta\lambda(q_i)$. Let μ^* be the maximum among all these cycle metrics, i.e. $\mu^* = \max_{c \in C_{crit}} \{\mu(c)\}$. Then,

$$\Delta\vartheta(\mathcal{M}\mathcal{G}_S, \mathcal{M}\mathcal{G}'_S) = \begin{cases} 0 & \text{if } \mu^* \leq \pi(\mathcal{M}\mathcal{G}_S) \\ \dfrac{\mu^* - \pi(\mathcal{M}\mathcal{G}_S)}{\pi(\mathcal{M}\mathcal{G}_S) \cdot \mu^*} & \text{otherwise} \end{cases}$$

The second method is more convenient when channel pipelining is applied incrementally on few channels at the time.

Example Figure 6.1 shows three infinite-queue marked-graph models for three distinct implementations of the MAC circuit S_{mac} shown in Figure 1.2. This is a fairly small circuit that nowadays is available as a simple IP core. Therefore, it is unlikely that its internal wires need any post-layout optimization in terms of wire pipelining. Still, I use it here as a simple example to illustrate the concepts that I presented.

The MAC circuit contains only one cycle corresponding to the feedback path around the *add/sub-mux* block. The left-hand side marked graph $\mathcal{M}\mathcal{G}_{ref}$ is the reference marked-graph model for S_{mac}. I consider two implementations without back-pressure and, therefore, I analyze them with the infinite-queue marked graph model.

[2] Besides the common modifier, "critical", the notion of critical cycle of a marked graph, given in Definition 5.10, is completely unrelated to the notion of critical channel of an annotated marked graph model given in Definition 6.4.

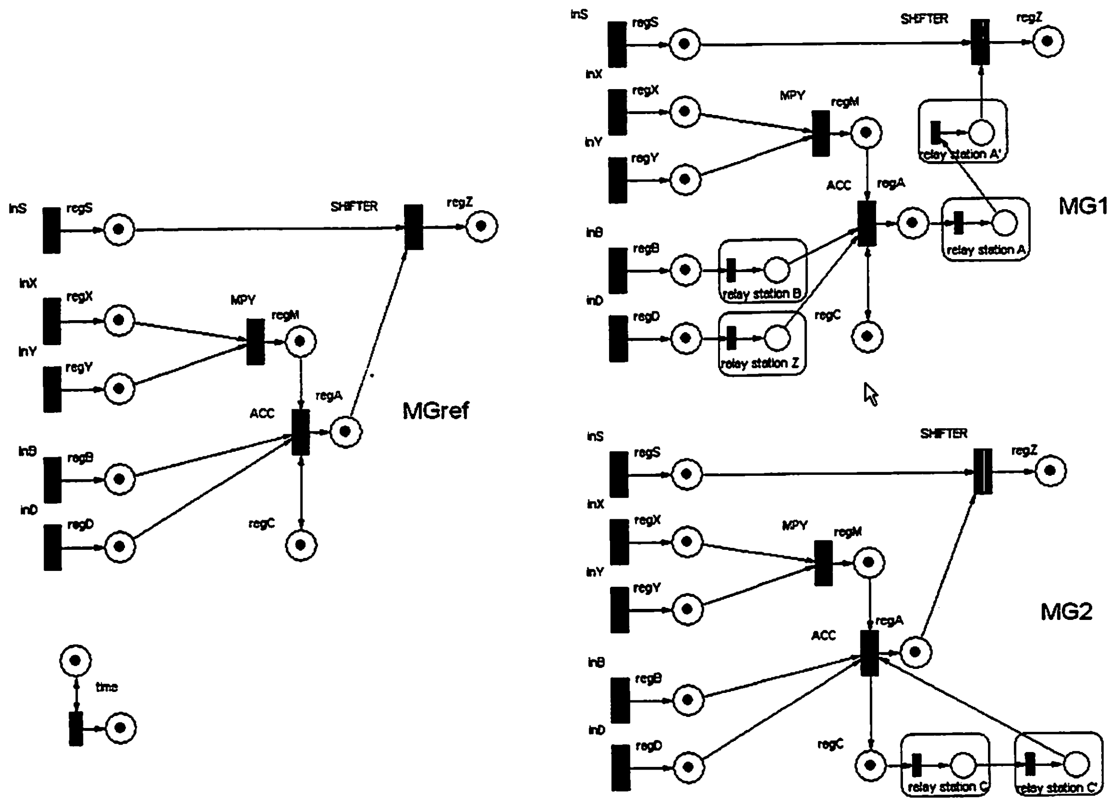

Figure 6.1: Marked-graphs for three implementations of the MAC circuit of Figure 1.2.

The first implementation has 3 critical channels ($q_{\{inD,acc\}}$, $q_{\{acc,shifter\}}$, and $q_{\{inB,acc\}}$) with $\gamma(q_{\{inB,acc\}},\psi) = 2$, $\gamma(q_{\{inD,acc\}},\psi) = 2$ and $\gamma(q_{\{acc,shifter\}},\psi) = 3$ for a given nominal clock period ψ. Then, the legalization step requires the insertion of one relay station on the first two channels and two on the third in order to have the matching latencies $\lambda(q_{\{inB,acc\}}) = 2$, $\lambda(q_{\{inD,acc\}}) = 2$ and $\lambda(q_{\{acc,shifter\}}) = 3$. The resulting legalized infinite-queue marked graphs model $\mathcal{MG}_1$ is shown on the right-hand side of Figure 6.1 with the initially empty place representing the four relay stations in the initial marking. The insertion of the relay stations gives a latency-equivalent design without performance degradation. The marking of $\mathcal{MG}_1$ after one hundred timestamps is shown in Figure 6.4. The behavior of the corresponding latency-insensitive system is reported in Figure 6.2, where the progressive natural numbers simply denote distinct informative events. The *acc* mod-

time	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regX})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regY})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regM})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regB})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{rsB})$	=	τ 1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regZ})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{rsZ})$	=	τ 1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regC})$	=	1 τ 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regA})$	=	1 τ 2 3 4 5 6 7 8 9...
$\sigma(\mathit{rsA})$	=	τ 1 τ 2 3 4 5 6 7 8 9...
$\sigma(\mathit{rsA'})$	=	τ τ 1 τ 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regT})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{outT})$	=	1 τ τ 2 τ 3 4 5 6 7 8 9...
$\sigma(\mathit{outW})$	=	1 τ τ 2 τ 3 4 5 6 7 8 9...

Figure 6.2: Sequence of events on the channels of marked graph $\mathcal{MG}_1$ in Figure 6.4.

time	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regX})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regY})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regM})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regB})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{regZ})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{rsC})$	=	τ 1 τ τ 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...
$\sigma(\mathit{rsC'})$	=	τ τ 1 τ τ 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...
$\sigma(\mathit{regC})$	=	1 τ τ 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...
$\sigma(\mathit{regA})$	=	1 τ τ 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...
$\sigma(\mathit{regT})$	=	1 2 3 4 5 6 7 8 9...
$\sigma(\mathit{outT})$	=	1 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...
$\sigma(\mathit{outW})$	=	1 2 τ τ 3 τ τ 4 τ τ 5 τ τ 6 τ τ 7 τ τ 8 τ τ 9...

Figure 6.3: Sequence of events on the channels of marked graph $\mathcal{MG}_2$ in Figure 6.4.

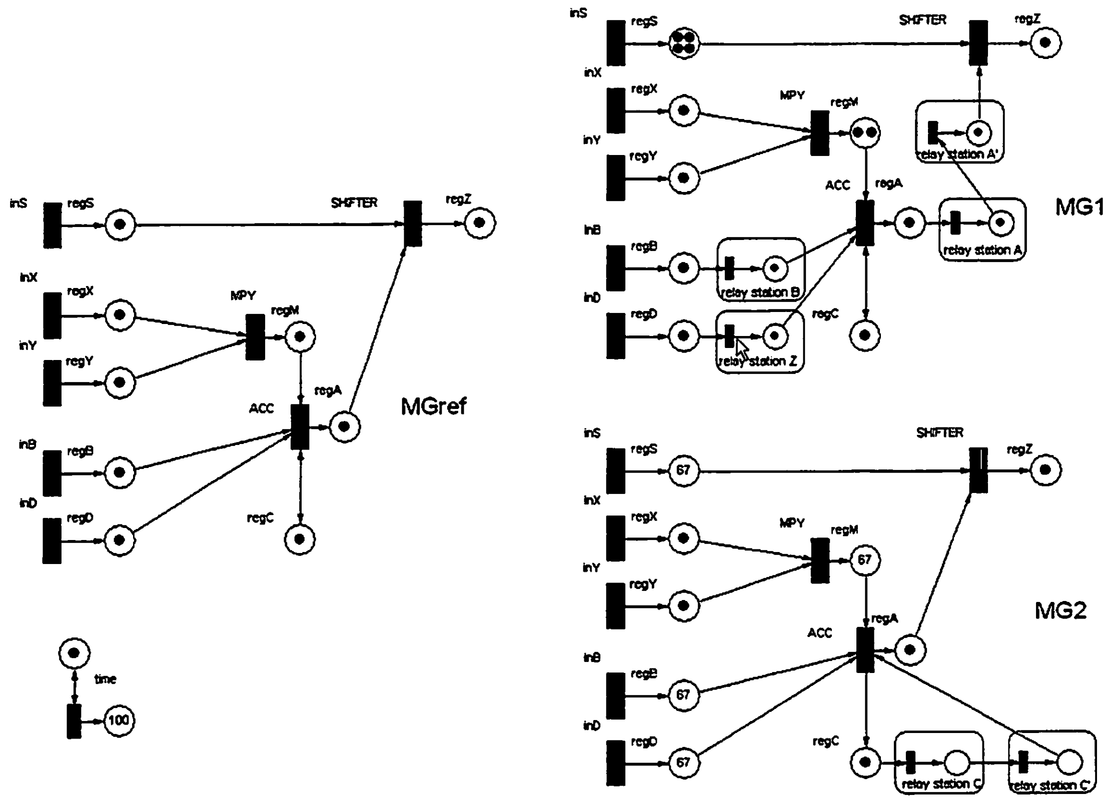

Figure 6.4: The marked graphs of Figure 6.1 after 100 timestamps.

ule stalls once at the first timestamp, while the *shifter* module stalls three times (at the first, second and fourth timestamp). After four timestamps the stalling events (token vacancies) have left the system, which has reached its steady-state, processing data with throughput $\vartheta(\mathcal{MG}_1) = 1$. The shell encapsulating the *acc* module must have a queue of length one at the input port of the feedback path. Similarly, the shell encapsulating the *shifter* module must have a queue of length two at the input port of the channel associated to place *regS*. All the other shells do not need to have any queue and, since back-pressure is not used, relay stations can simply be implemented as unit-capacity stateful repeaters (e.g. like a normal flip-flop).

The second implementation contains only one critical channel $q_{\{acc,acc\}}$ (a self-loop) on the feedback path of the *acc* module, with $\gamma(q_{\{acc,acc\}}, \psi) = 3$. The resulting legalized marked graphs model $\mathcal{MG}_2$ with two relay stations on channel $q_{\{acc,acc\}}$ is shown with its initial marking also on the right-hand side of Figure 6.1. Since channel $q_{\{acc,acc\}}$ belongs to a cycle of the $\mathcal{MG}_2$ (it is a self-loop!), the system throughput becomes $\vartheta(G') = \frac{1}{3}$, with degradation $\Delta\vartheta(G, G') = 66\%$, as illustrated in Figures 6.3 and 6.4. Furthermore, this implementation is not feasible under the assumption that the environment cannot be slowed down to produce data at the rate of $\frac{1}{3}$ [3]. In fact, as illustrated by the token accumulations in Figure 6.4, infinite queues would be necessary in various places: e.g, in the shell of the *acc* module to buffer data coming from the channels associated to places *regA*, *regB*, and *regZ*, as well as in the shell of the *shifter* module to buffer data on the channel associated to place *regS*. ■

The previous example illustrates the dramatic difference in terms of global impact that the insertion of relay stations on a local channel can have. In the first implementation the performance is unaffected and finite queues are sufficient, in the second implementation the performance is cut to one-third of its original value and there is unbounded token accumulation. The next section discusses how this impact can be explained based on the computational structure of the specific latency-insensitive system.

6.1.2 The Role of System Topology

Using the modeling techniques of Chapter 5, it is possible to classify the impact of channel pipelining on a latency-insensitive system implementation S based only on the topology of the marked-graph model $\mathcal{MG}_S$, which reflects the computational structure of S. Four main scenarios are possible:

1. *$\mathcal{MG}_S$ is an acyclic graph.* The insertion of relay stations on any channel of $\mathcal{MG}_S$ does not have impact on the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$. Some stalling events are observed at the output ports of the system, thereby forcing the corresponding informative events to arrive with some clock cycles of additional latency, but, eventually, the system reaches a steady state and no more stalling events

[3]In the marked-graph model, slowing down the operational environment to a rate of $\frac{1}{3}$ can be represented by adding an "environment source" transition-place pair connected to the input transitions of the system and, then, inserting a feedback path on the source transition with two initially empty places.

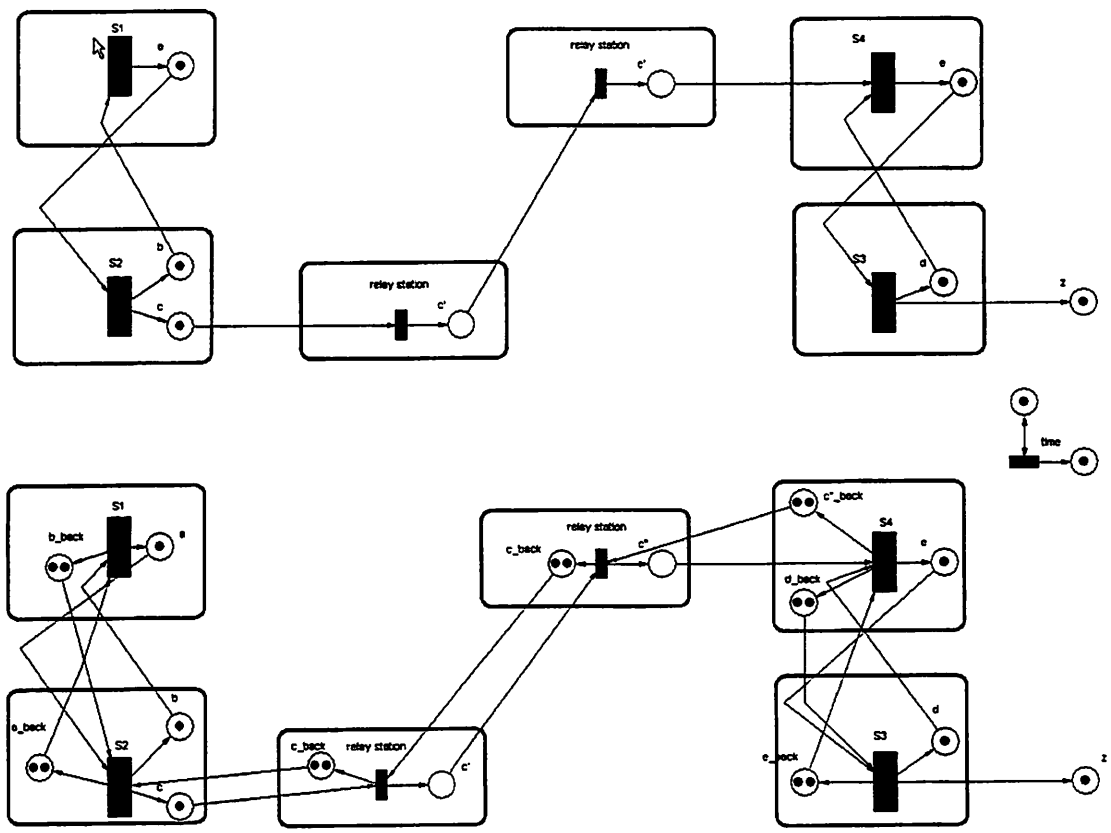

Figure 6.5: Models of cyclic latency-insensitive system with relay station on acyclic path.

are observed. Also, no unbounded token accumulation occurs within the system. Queues of finite length are necessary in those shells that have multiple input channels and at least one of these channels receives some stalling events. These channels, however, can be easily identified based on the location of the relay stations and the overall topology of $\vartheta(\mathcal{M}\mathcal{G}_S)$.

2. *$\mathcal{M}\mathcal{G}_S$ is a cyclic graph with several strongly connected components (SCC) and the relay station are inserted between them.* To analyze this scenario without loss of generality it is sufficient to consider the case of only two SCCs. The insertion of relay stations on any channel connecting two distinct SCCs of $\mathcal{M}\mathcal{G}_S$ does not have any impact on the maximum sustainable throughput $\vartheta(\mathcal{M}\mathcal{G}_S)$ either. This fact may

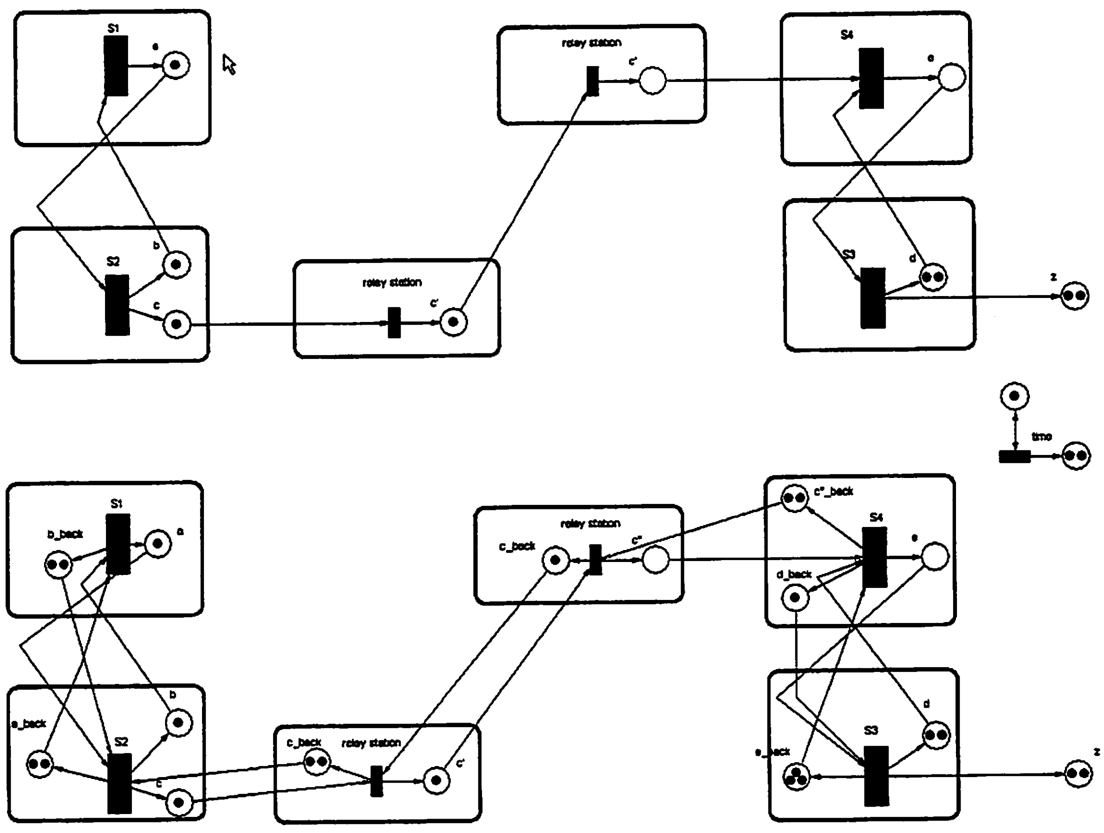

Figure 6.6: The marked graphs of Figure 6.5 after 1 timestamp.

initially seem counterintuitive. After all the down-link SCC S_{down} is a cyclic system and the shell s_1 of S_{down}, when receives the stalling events from the pipelined channel, reacts producing stalling events that cycle around within S_{down}. However, shell s_1 is also the shell that "retires" these stalling events after they have completed a cycle around the system.

The scenario is illustrated in Figure 6.5 both for the case of an infinite-queue model (top diagram) and the case of a 2-finite queue model (bottom diagram). The system contains two SCCs having two shell-core pairs each and no internal relay stations (therefore they share the same unit throughput). The two SCCs are connected by a channel q that contains two relay stations (channel latency $\lambda(q) = 3$). The same

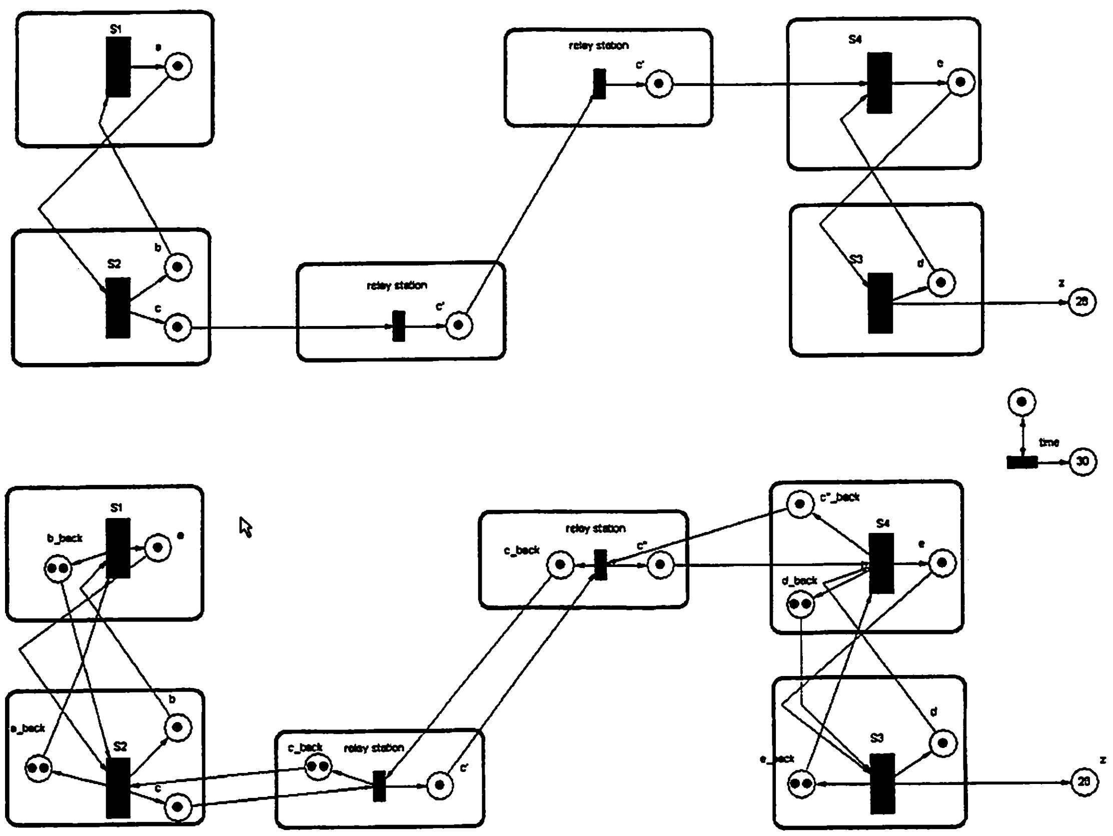

Figure 6.7: The marked graphs of Figure 6.5 after 30 timestamps.

marked graphs are reported in Figure 6.6 after one timestamp has passed from the initial marking: in the infinite-queue model, shell s_4 has received a stalling event (a token vacancy) from q and stalled (i.e. produced a stalling event), while shell S_3 has produced a stalling event that is accumulated in place d together with the previous event (which has not been processed by s_4 yet). At the following timestamp, shell s_4 receives the second and last informative event from q, stalls again, but now also s_3 stalls and the token count on place d remains the same. At the third timestamps, shell s_4 finally consumes an informative event while s_3 stalls for the second and last time. The token count on d goes back to one and from now on stays equal to one as no shell stalls any longer. Figure 6.7 shows the same marked graph after thirty timestamp have passed. The impact of channel pipelining on the system performance is limited to a latency of two cycles with respect to the output sequence of a latency-equivalent strict system. The impact in terms of area overhead is limited to the insertion of a queue of size two within s_4 in correspondence of the input channel from s_3. The behavior of the 2-finite queue marked-graph model is completely equivalent.

3. *$\mathcal{MG}_S$ is a single strongly connected component.* The insertion of relay stations on any channel of $\mathcal{MG}_S$ has always a negative impact on the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$. The impact, which varies depending on the structure of the cycles of $\mathcal{MG}_S$, can be exactly calculated using the techniques discussed in Chapter 5. No unbounded token accumulation occurs within the system whether the implementation is based on back-pressure or not. Unbounded token accumulation can occur at the boundary between the system and the environment in which it operates. This issue is discussed in detail in Section 6.1.3.

4. *$\mathcal{MG}_S$ is a cyclic graph with several strongly connected components and the relay station are inserted within some of them.* For each distinct SCC this case boils down to the previous case and the corresponding maximum sustainable throughput can be calculated exactly. The interaction between the SCCs, however, needs to be managed carefully because this is the case where unbounded token accumulation can occur inside the system. In fact, considering the component graph G^{SCC}, if a SCC S_1 lies in an up-link position with respect to another SCC S_2 and $\vartheta(S_1) > \vartheta(S_2)$ then unbounded token accumulation is guaranteed to happen [4]. To avoid unbounded token accumulation it is necessary either to implement back-pressure or to slow-down the faster SCC. The trade-off between these two alternatives is discussed in Section 6.1.3. Section 6.1.4 explains a strategy to slow-down the faster SCC through the insertion of a proper number of uncalled-for relay stations.

[4]See also the discussion in Section 5.2.2.

6.1.3 The Role of Back-Pressure

The choice of whether to use a back-pressure mechanism in completing the physical implementation of a latency-insensitive system S that operates within an environment E is not straightforward. From Chapter 5, and particularly Theorem 5.8, it is clear that this choice does not affect the maximum sustainable throughput $\vartheta(S)$ of the system. In other words, as long as the environment is always capable of providing new true packets and does not generate ever stalling requests, back-pressure is not a factor on determining the system performance. On the other hand, back-pressure is a factor in guaranteeing that any specification of a latency-insensitive system S can be physically implemented under any possible configuration of channel pipelining. In fact, a back-pressure mechanism can always replace the need for (*unfeasible*) infinite-length queues in providing a correct, physical, system implementation that runs at the same throughput and does not experience any queue overflow. Back-pressure, however, comes with the assumption that the operational environment is ready to stall whenever the system sends back a stalling event. In this regard, the completion of a physical implementation (like the one represented by the marked-graph model $\mathcal{MG}_2$ of Figure 6.4) generally boils down to a choice between two alternatives in order to handle the token accumulation problem:

- use back-pressure and reduce *dynamically* the token production rate of the environment E to match the maximum sustainable throughput of S;
- do not use back-pressure and reduce *statically* the token production rate of E by having E running with a nominal clock frequency $\phi(E)$ that matches the effective clock frequency $\phi_{eff}(S)$ of S, which, as defined in Section 4.3.1, corresponds to the nominal clock frequency $\phi(S)$ times throughput $\vartheta(S)$.

This choice, however, may be restricted when S is an *open* system, i.e. a system that must be designed without being able to control also the design of the other systems that constitute its operational environment E. In this case, if E is latency-insensitive then back-pressure must be used, while if E is not latency-insensitive then the final design of S will be good only for those environments running at the effective clock frequency of S.

When S is not an open system, the above decision is only influenced by considerations on the area and power overhead of the alternative solutions (since their performance is the same). The results presented in Section 5.2.4 guarantee that with an implementation based on back-pressure it is sufficient to size the lengths of all the shell input queues to be equal to two. Therefore, back-pressure enables the creation of a very modular design. Furthermore, the area impact of each shell can be easily estimated from the information on the number of input channels of the corresponding core. In addition, it is necessary to account for the additional wires implementing the back-pressure signal and the double storage space within each relay station. The alternative is not to use back-pressure, thereby removing these additional wires and deploying a unit-capacity stateful repeater in the place of each relay station. However, in this case, it is necessary to equalize the throughput values of each SCC component in the system to avoid unbounded accumulation. Furthermore each input queue of every shell in the system must be sized *ad hoc* and its length may possibly be bigger than two. This is another reason why it is important to make an attempt for a balanced design. The next section presents a simple strategy to equalize the throughput across many SCCs, while Section 6.2.3 discusses more sophisticated techniques to achieve a design where communication and computation are well balanced.

6.1.4 Throughput Equalization in the Absence of Back-Pressure

In this section, I consider a marked-graph model $\mathcal{MG}_S$ that is cyclic and contains more than one strongly connected component. This may be the result of (1) modeling a latency-insensitive system S that has such topology or (2) modeling together the system S and its environment E as part of a design where there is no feedback-loop from S to E. Under the assumption that $\mathcal{MG}_S$ represents an implementation without back-pressure, after performing channel pipelining it is necessary to equalize the throughput across all SCCs of $\mathcal{MG}_S$ in order to avoid unbounded token accumulation. Under the additional assumption that $\mathcal{MG}_S$ is implemented as a synchronous design with a single-clock domain of nominal period ψ, the simplest way to perform throughput equalization is through the insertion of extra relay stations that are not required for pipelining purposes.

Besides avoiding unbounded token accumulation, it may be convenient to minimize the number and size of the shell queues versus the number of extra relay stations. The following procedure can be executed to achieve both goals while legalizing the marked-graph model $\mathcal{M}\mathcal{G}_S$:

1. *Channel delay annotation*: based on the information from the layout derive the annotated marked-graph model $\mathcal{M}\mathcal{G}'_S$.

2. *Channel pipelining*: legalize $\mathcal{M}\mathcal{G}'_S$ by inserting the required number of relay stations for each of its critical channels as specified by Lemma 6.1.

3. *SCC throughput calculation*: derive the component graph G^{SCC} of $\mathcal{M}\mathcal{G}'_S$ and for each SCC $S_i \in G^{SCC}$ compute the maximum sustainable throughput $\vartheta(S_i)$ as discussed in Section 5.2.2;

4. *Target throughput calculation*: calculate the target throughput
$$\vartheta^*(\mathcal{M}\mathcal{G}_S) = \min_{\{S_i \in G^{SCC}\}} \big(\vartheta(S_i)\big)$$

5. *Detection of critical and potentially critical cycles*: for each $S_i \in G^{SCC}$ compute the set of critical cycles and potentially critical cycles $C_{crit}(S_i)$. This set contains each cycle c_k such that the reciprocal of its cycle metric is higher than the target throughput, i.e.
$$\frac{1}{\mu(c_k)} = \frac{a_k}{b_k} > \vartheta^*$$
A potentially critical cycle satisfies the above condition without being a critical cycle of S_i. Let $C_{crit}(\mathcal{M}\mathcal{G}_S)$ be the union of the sets $C_{crit}(S_i)$.

6. *Throughput equalization*: equalize the throughput values of all SCCs by inserting the necessary number n_k of *extra relay station* on each cycle c_k in the set $C_{crit}(\mathcal{M}\mathcal{G}_S)$. This corresponds to augmenting by a quantity $n_k \in \mathbb{N}$ the numerator a_k of $\mu(c_k)$. To find the quantities n_k is necessary to solve Problem A.3, the *decreasing rate equalization problem*, discussed in the Appendix (together with an algorithm to solve it).

The above procedure can be slightly modified if the goal is only to avoid the use of back-pressure. In this case, one can exploit the fact that a SCC S_i with maximum sustainable throughput $\vartheta(S_i)$ can always sustain the throughput of a SCC S_j with $\vartheta(S_j) < \vartheta(S_i)$. In other words if S_j is the up-link component the throughput equalization occurs implicitly even in absence of back-pressure. Therefore, while traversing in topological order the component graph G^{SCC} it is possible to detect local situations where the throughput of an SCC does necessarily need to be equalized. Normally, the price to pay is longer input queues within the shells of the faster SCCs.

The above procedure assumes that each potentially critical cycle of a SCC S_i contains at least one channel that is not shared with any other cycle and, therefore, that can be pipelined without affecting the cycle metric of the other cycles of S_i. Although one can conceive cyclic structure where this is not the case, in practice, it is a reasonable assumption. In any case, the algorithm presented in the Appendix to solve the *cycle balancing problem* discussed in Section 6.2.3 handles also the case when this assumption does not hold.

6.1.5 Case Study: an MPEG Video Encoder. Part One

Figure 6.8 illustrates the functional block diagram of an MPEG-2 *video encoder*, a system that was implemented first as a chip-set and then as a system-on-chip by Ikeda *et. al* [83, 84]. For the purposes of this example, I assume that the functional block diagram corresponds also to the block diagram of the final implementation and that, at each clock cycle, every block in the pipeline provides a new relevant data item to the down-link block: in other words, at every cycle, on each channel there is either a stalling event or an informative event, and the latter corresponds to a data value that is not a *don't care* value for the receiving block [168]. Naturally, this may not always be the case for a system specified at this level of granularity, because, for instance, the *quantizer* module may take more than one clock cycle before it is able to produce a result which can trigger a new computation of the down-link *inverse quantizer* module. Still, these kinds of *don't care* values may only help from the performance viewpoint because they can be exploited to reduce the stalling a module.

Figure 6.9 illustrates the reference infinite-queue marked-graph model $\mathcal{MG}^\infty$ for the MPEG-2 video encoder. Being a reference graph, it does not have any initially empty place (relay station) and every channel q consists exactly of one place between two transitions

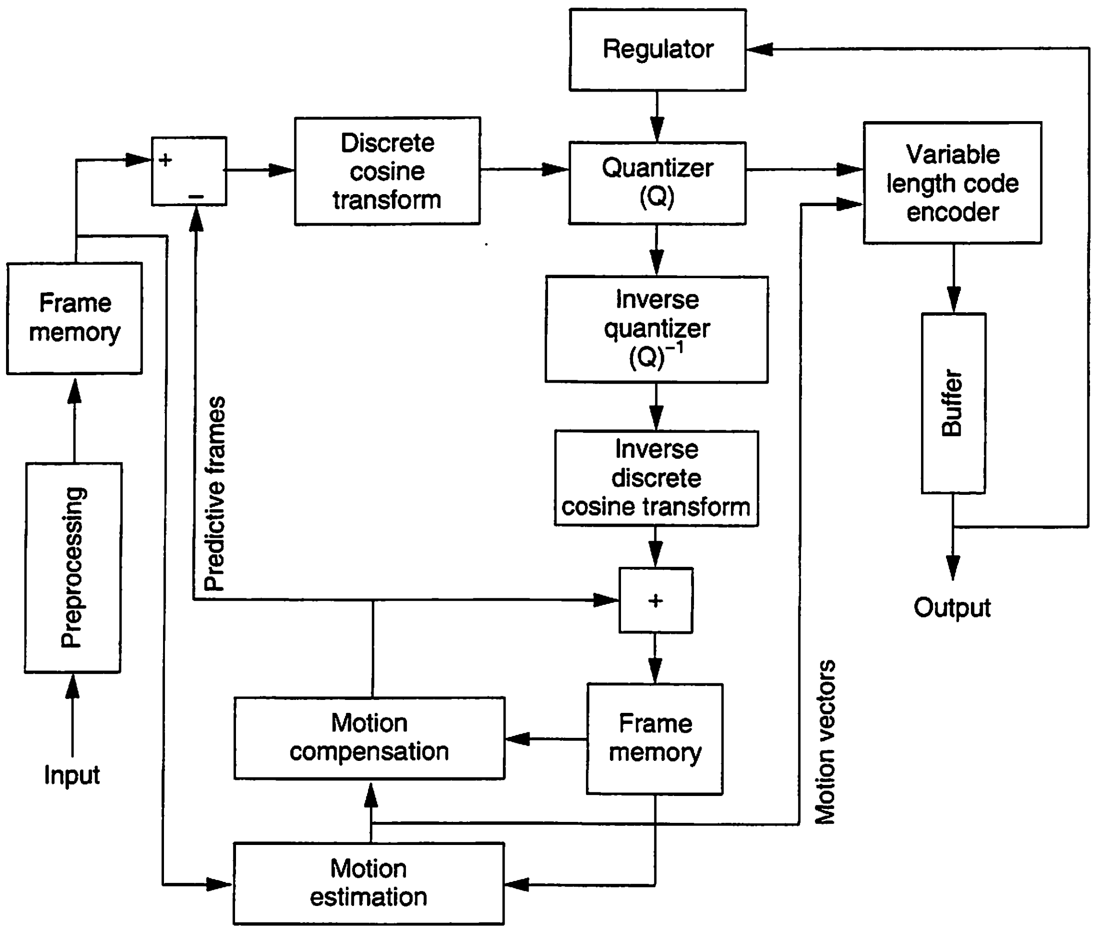

Figure 6.8: MPEG-2 video encoder (functional block diagram).

(two shell-pair cores), thereby having latency $\lambda(q) = 1$.

Table 6.1 reports the six cycles of $\mathcal{M}\mathcal{G}^{\infty}_{ref}$ together with the sum of the tokens presented in their places on the initial marking and their (unit) cycle times. Table 6.2 is a matrix whose entry (q_i, c_j) denotes that channel q_i belongs to cycle c_j. Most channels of $\mathcal{M}\mathcal{G}^{\infty}_{ref}$ are common to more than one cycle, e.g. channel $q_{\{t_{16},t_{20}\}}$ between transition t_{16} (*sum-2*) and transition t_{20} (*frame memory-2*) belongs to five cycles (c_1, c_2, c_4, c_5, and c_6). Others channels are contained only in one cycle, e.g. channel $q_{\{t_6,t_{13}\}}$ between transition t_6 (*motion estimation*) and transition t_{13} (*VLC encoder*) is only part of cycle c_6. Finally, some channels

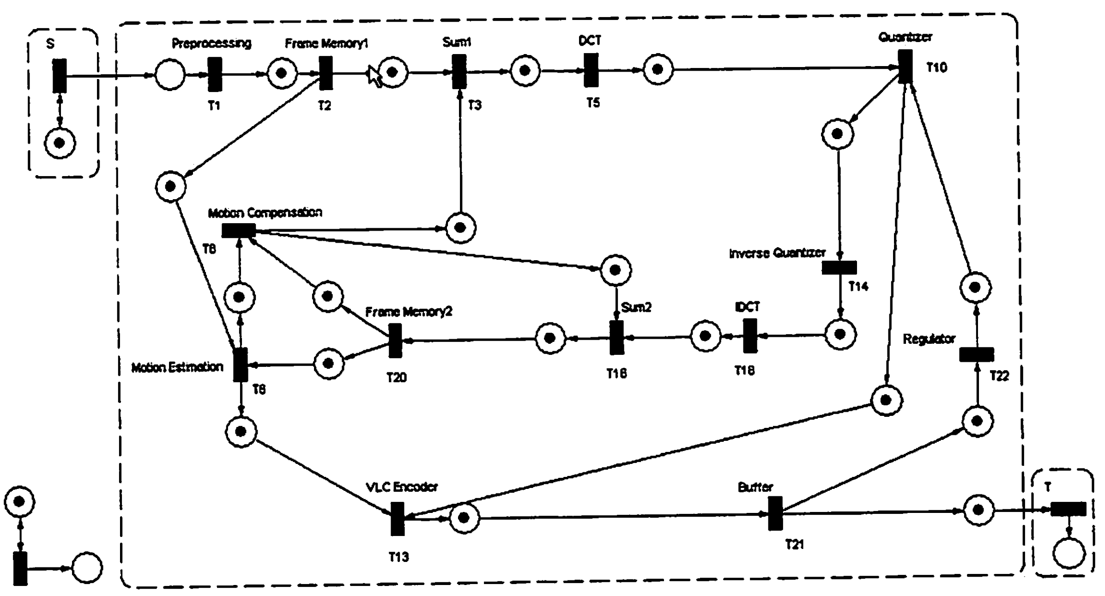

Figure 6.9: Reference infinite-queue marked graph model for the MPEG-2 video encoder of Figure 6.8.

are not part of any cycle, e.g. channel $q_{\{t_2,t_3\}}$ between transition t_2 (*frame memory 2*) and transition t_3 (*sum-1*), and, therefore, to increase their latency does not affect the system throughput.

As discussed in Section 6.1.1, the throughput degradation that may occur as a consequence of pipelining a given channel can be calculated in advance. Intuitively, to pipeline channels that belong to cycles of relatively small cardinality has a worse impact on the system throughput. This follows directly from Definition 5.8 of cycle metric: for a given number of relay stations that must be inserted on a channel q that belongs to a cycle c_i, the smaller is the cardinality $|c_i|$ the worse is the loss in throughput for $\mathcal{MG}^\infty$ [5].

Figure 6.10 reports the results of a simple analysis that can be completed based only on the topology of $\mathcal{MG}^\infty$. The six curves in the chart are associated to the cycles of Table 6.1 and their shapes should be interpreted as follows: each point of curve c_i shows the amount of system throughput degradation which is detected after setting the total sum $\lambda(c_i)$ of the latency of the channels of c_i equal to integer x, with $x \in [0,20]$. Obviously, the underlying

[5]The worst case is clearly represented by self-loops like in the case of the MAC circuit of Figure 6.1.

c_i	cycle transitions	$M_0(c_i)$	$\mu(c_i)$
c_1	$\{t_8, t_{16}, t_{20}\}$	3	3/3
c_2	$\{t_6, t_8, t_{16}, t_{20}\}$	4	4/4
c_3	$\{t_{10}, t_{13}, t_{21}, t_{22}\}$	4	4/4
c_4	$\{t_{10}, t_{14}, t_{18}, t_{16}, t_{20}, t_8, t_3, t_5\}$	8	8/8
c_5	$\{t_{10}, t_{14}, t_{18}, t_{16}, t_{20}, t_6, t_8, t_3, t_5\}$	9	9/9
c_6	$\{t_{10}, t_{14}, t_{18}, t_{16}, t_{20}, t_6, t_{13}, t_{21}, t_{22}\}$	9	9/9

Table 6.1: Cycles and cycle times for the marked graph of Figure 6.9.

q_i	c_1	c_2	c_3	c_4	c_5	c_6
$q_{\{t_1,t_2\}}$						
$q_{\{t_2,t_3\}}$						
$q_{\{t_2,t_6\}}$						
$q_{\{t_3,t_5\}}$				⋆	⋆	
$q_{\{t_5,t_{10}\}}$				⋆	⋆	
$q_{\{t_6,t_{13}\}}$						⋆
$q_{\{t_6,t_8\}}$		⋆			⋆	
$q_{\{t_8,t_3\}}$				⋆	⋆	
$q_{\{t_8,t_{16}\}}$	⋆	⋆				
$q_{\{t_{10},t_{13}\}}$			⋆			
$q_{\{t_{10},t_{14}\}}$				⋆	⋆	⋆
$q_{\{t_{13},t_{21}\}}$			⋆			⋆
$q_{\{t_{14},t_{18}\}}$				⋆	⋆	⋆
$q_{\{t_{16},t_{20}\}}$	⋆	⋆		⋆	⋆	⋆
$q_{\{t_{18},t_{16}\}}$				⋆	⋆	⋆
$q_{\{t_{20},t_6\}}$		⋆			⋆	⋆
$q_{\{t_{20},t_8\}}$	⋆			⋆		
$q_{\{t_{21},t_{22}\}}$			⋆			⋆
$q_{\{t_{22},t_{10}\}}$			⋆			⋆

Table 6.2: Channels vs. cycles matrix for the marked graph of Figure 6.9.

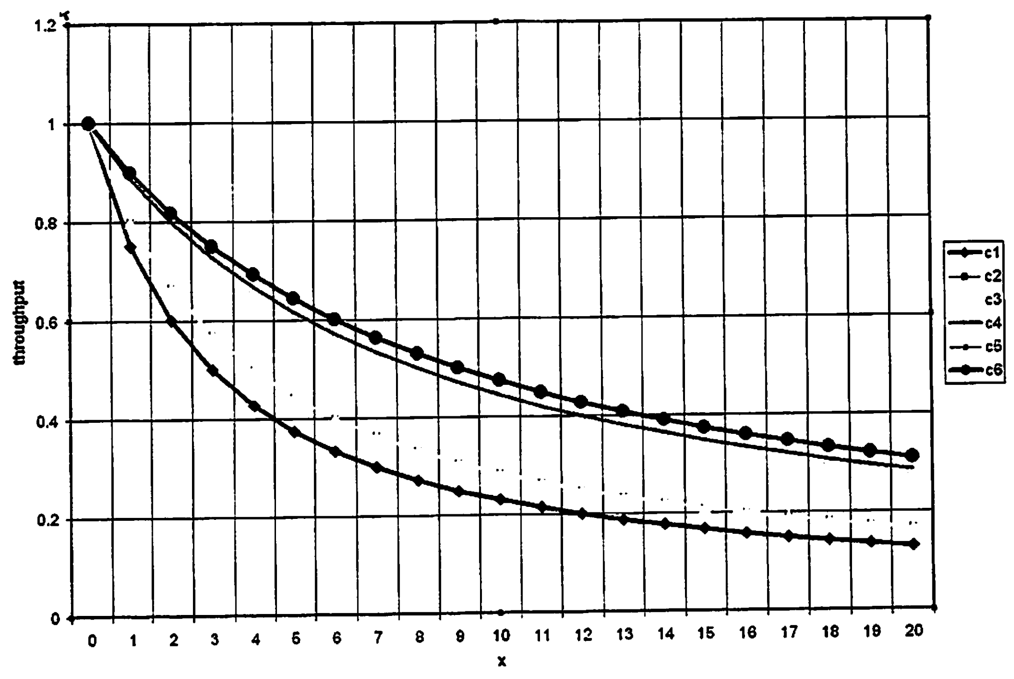

Figure 6.10: MPEG-2 encoder: analysis of throughput degradation (worst-case scenario).

assumption is that c_i is a critical cycle of $\mathcal{MG}_S$, and this limits the choice of those channels of c_i whose latency can be augmented. For example, assume that $\lambda(c_2) = 5$, as a result of summing of a latency $\lambda(q_{\{t_{16},t_{20}\}}) = 4$, $\lambda(q_{\{t_{20},t_6\}}) = 1$, $\lambda(q_{\{t_6,t_8\}}) = 0$, and $\lambda(q_{\{t_8,t_{16}\}}) = 0$. In this case c_2 is definitely not a critical cycle. In fact, its cycle metric is $\mu(c_2) = \frac{5+4}{4} = \frac{9}{4} = 2.250$, while, even if $\lambda(q_{\{t_{20},t_8\}}) = \lambda(q_{\{t_8,t_{16}\}}) = 0$, cycle c_1, having $\lambda(c_1) = \lambda(q_{\{t_{16},t_{20}\}}) = 4$, has a larger cycle metric, exactly $\mu(c_1) = \frac{4+3}{3} = \frac{7}{3} = 2.333$. Not surprisingly, Figure 6.10 confirms that to avoid the risk of major performance losses is necessary to avoid being forced to pipeline channels which belong to a critical cycle or a potentially critical cycle, i.e. a cycle of relatively small cardinality. While performing channel pipelining, this choice may be quite restricted. For example, if the length of the wires implementing channel $q_{\{t_{20},t_8\}}$ between the *frame memory-2* module (transition t_{20}) and the *motion compensation* module (transition t_8) is very big, cycle c_1 will ultimately dictate the system throughput.

These considerations must be kept in mind while partitioning the functionality of the system in tasks to be assigned to different IP cores. It is true that the latency insensitive methodology guarantees that no matter how bad is the final implementation of the system (in terms of lengths of the wires implementing the communication channels), it is always possible to fix it by adding relay stations. Still, to achieve good performance, it is necessary to adopt a design strategy based on the following guidelines:

- all modules encapsulated within a shell should put comparable timing constraints on the global clock (i.e. the delays of the longest combinatorial paths inside each module should be similar);
- modules belonging to the same SCC should be kept close in the final layout; this is quite intuitive, but less intuitive is the observation that latency-insensitive design offers the additional freedom of placing interacting SCCs far apart; this trade-off is possible because, as discussed in Section 6.1.3, pipelining inter-SCC channels does not have any impact on the performance of a latency-insensitive system;
- within a SCC, modules that belong to the same cycle should be placed close together in the final layout; the smaller is the cardinality of the cycle, the closer its components should be;
- one of the goals of physical design should be the optimization of the following trade-off: minimize the length of those wires which implement channels that belong to both small and big cycles while increasing, if necessary, those wires implementing channels that belong only to cycles of bigger cardinality.

The last point is illustrated in Figure 6.9 and Table 6.2. In the case of the MPEG-2 video encoder, a good trade-off is to reduce the length of the wires connecting the *sum-2* module (transition t_{16}) with the *frame memory-2* module (transition t_{20}), while increasing in exchange the length of the wires connecting the *motion estimation* module (transition t_6) with the *VLC encoder* module (transition t_{13}). In fact, this trade-off would reduce the normalized delay $\gamma(q_{\{t_{16},t_{20}\}},\psi)$ of $q_{\{t_{16},t_{20}\}}$, which belongs to 5 cycles including the small cycles c_1 and c_2, in exchange for an increase of the normalized delay $\gamma(q_{\{t_6,t_{13}\}},\psi)$ of channel $q_{\{t_6,t_{13}\}}$, which belongs only to the biggest cycle c_6. Consequently, if $\mathcal{MG}^{\infty}_{ref}$ needs to be legalized, there are considerably more options for increasing the latency of channels that do not impact the overall system throughput. For instance, a layout S_1 where

$\gamma(q_{\{t_{16},t_{20}\}},\psi) = 3$, while $\gamma(q_{\{t_6,t_{13}\}},\psi) = 1$, must be legalized through the pipelining of channel $q_{\{t_{16},t_{20}\}}$ with two relay stations in order to have $\lambda(q_{\{t_{16},t_{20}\}}) = 3$. Instead, a layout S_2 with $\gamma(q_{\{t_6,t_{13}\}},\psi) = 3$ and $\gamma(q_{\{t_{16},t_{20}\}},\psi) = 1$ requires the two relay stations be inserted on $q_{\{t_6,t_{13}\}}$. Assuming that in both cases the remaining channels do not need any pipelining, the throughput values of the two implementations are $\vartheta(S_1) = \frac{3}{5} = 0.6$ and $\vartheta(S_2) = \frac{9}{11} = 0.818$. Hence, the latter gives a 36% performance improvement with respect to the former.

6.2 Recycling

As the case study of the previous section shows, the interaction among the components of a modern system on silicon can be quite complex. SOC designers face many critical decisions, including: which component to use to implement a given functionality, where to place the chosen components, and which communication scheme to implement.

Several pre-designed IP core modules are generally available for a given functionality, and they offer different benefits: e.g., a deeply pipelined module may provide higher throughput in exchange for a bigger design overhead in terms of area and power dissipation. While choosing the best suited IP cores designers must solve the problem of balancing the differences in throughput and latency among them. Also, a complex functionality can be implemented by either assembling multiple modules or choosing a single pre-designed one. These decisions are further complicated by the fact that a *local* design choice may have a *global* impact on the system performance. Finally, the shift from computation- to communication-bound design, discussed in Section 2.2, makes matters worse by adding the new challenge of designing a distributed system where communication and computation latencies are well-balanced.

In this context, one of the main advantages offered by latency-insensitive design is the new freedom in *moving around latency*—at any stage of the design process and without necessarily forcing the redesign of any core module—in order to optimize "latency vs. throughput" trade-offs and balance communication and computation. Latency can be varied in a controlled way that is transparent to the core modules, by inserting and pushing around

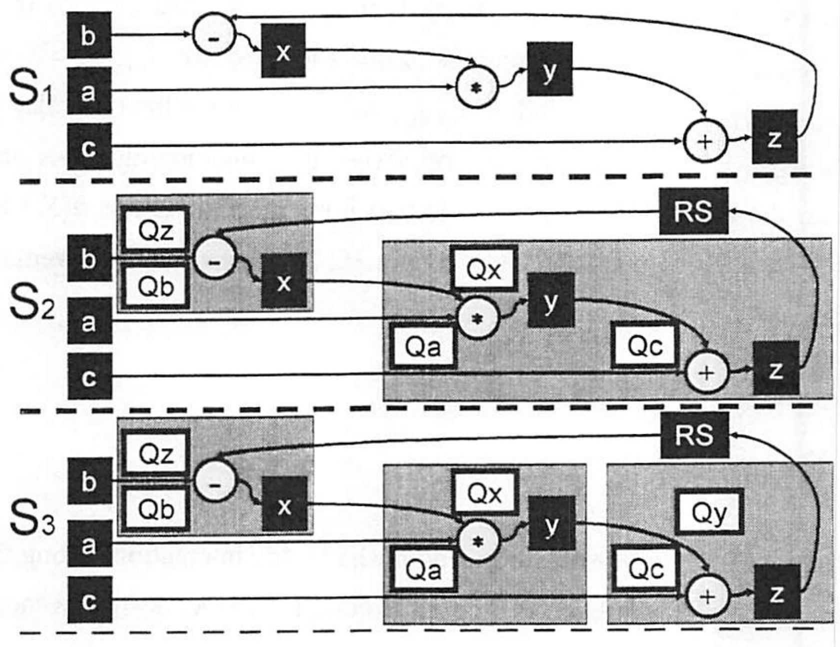

Figure 6.11: Strict system specification of a pipelined data-path and two alternative latency-insensitive implementations.

relay stations and by redrawing the boundaries of the shells. The combination of these operations is referred to as *recycling*.

6.2.1 The Role of Shell Encapsulation

Among the other factors, the performance of a latency-insensitive system is also affected by how shell encapsulation is performed. Where do draw the boundaries of each shell has a direct impact on the maximum sustainable throughput of a cyclic latency-insensitive system as demonstrated by the following example.

Example Figure 6.11 shows a simple system specification S_1 together with two distinct latency-insensitive implementation S_2 and S_3. System S_1 represents a *pipelined data-path*,

timestamp n	:	0	1	2	3	4	5	...
z	:	0	3	2	0	−2	9	...
b	:	0	2	4	2	8	10	...
x	:	1	0	−1	2	2	10	...
a	:	1	2	3	4	5	6	...
y	:	2	1	0	−3	8	10	...
c	:	1	1	0	1	1	0	...

Table 6.3: The behavior of the strict system S_1 in Figure 6.11.

whose functionality is captured by the following equations

$$\begin{aligned} x_{n+1} &= b_n - z_n \\ y_{n+1} &= a_n * x_n \\ z_{n+1} &= c_n + y_n \end{aligned}$$

where $x_1 = 1, y_1 = 2, z_1 = 0$ and $n \in \mathbb{N}$ denotes the clock timestamp. Therefore S_1 consists of three sequential modules, each computing a basic arithmetic operation and storing its value into a clocked register. Table 6.3 reports a behavior of system S_1, which obviously has throughput $\vartheta(S_1) = 1$.

Latency-insensitive system S_2 is obtained encapsulating the first sequential module (producing the values of signal x) into a shell and the other two modules into a second shell. A relay station *RS* has been inserted to pipeline the channel on the feedback path (signal z) which is assumed to be critical. Also, for the sake of simplicity, I assume that each shell contains an infinite queue for buffering the input signals that arrive with constant (unit) rate. Due to the presence of the relay station, system S_2 is not able to sustain unit throughput. Hence, it is necessary to slow down the rate of production of the environment to match the maximum sustainable throughput of S_2 [6].

Considering the stalling of the cores and the consequent input buffering in the queues,

[6]The considerations made in this example would not change with the assumption of using back-pressure in order to dynamically slow down the environment.

the functional activity of the latency-insensitive implementation S_1 is captured by the following equations [7]:

$$
\begin{aligned}
x_{n+1} &= \begin{cases} \tau & \text{if}\, RS_n = \tau \\ b_n - RS_n & \text{if}\, RS_n \neq \tau \wedge Qb.empty \\ Qb.pop - RS_n & \text{otherwise} \end{cases} \\
Qb_{n+1} &= Qb.push[b_n] \;\text{ if } \Big((RS_n = \tau) \vee (RS_n \neq \tau \wedge \neg Qb.empty) \Big) \\
y_{n+1} &= \begin{cases} y_n & \text{if}\, x_n = \tau \\ a_n * x_n & \text{if}\, x_n \neq \tau \wedge Qa.empty \\ Qa.pop + x_n & \text{otherwise} \end{cases} \\
Qa_{n+1} &= Qa.push[a_n] \;\text{ if } \Big((x_n = \tau) \vee (x_n \neq \tau \wedge \neg Qa.empty) \Big) \\
RS_{n+1} &= z_n \\
z_{n+1} &= \begin{cases} \tau & \text{if}\, x_n = \tau \\ c_n + y_n & \text{if}\, x_n \neq \tau \wedge Qc.empty \\ Qc.pop + y_n & \text{otherwise} \end{cases} \\
Qc_{n+1} &= Qc.push[c_n] \;\text{ if } \Big((x_n = \tau) \vee (x_n \neq \tau \wedge \neg Qc.empty) \Big)
\end{aligned}
$$

where $n \in \mathbb{N}, x_0 = 1, y_0 = 2, z_0 = 0$ and $RS_0 = \tau$.

Table 6.4 reports a behavior of system S_2. The restriction of this behavior to the inter-shell communication signals (i.e. all signals but y which is encapsulated within the second shell) is latency equivalent to the corresponding behavior of S_1 which is reported in Table 6.3. The signal traces in Table 6.4 are a confirmation of what can be calculated based on the topology of S_2, i.e. that the maximum sustainable throughput is $\vartheta(S_2) = \frac{2}{3}$. In particular, at timestamp $n = 1$, signal x presents a stalling event that forces the larger shell to stall the computation of both y and z; in fact, the y register already contains a value ($y = 1$) that could be use to compute the next value ($z = 2$) of z, but the control logic of the shell (discussed in Section 4.2.2) prevents this value from being computed at this cycle since the entire logic of the core is stalled. Hence, z carries a stalling event at timestamp $n = 2$.

[7]These equations are simplified thanks to the observation that queues Qx and Qz are never activated.

The other latency-insensitive system implementation, system S_3, presents a more granular shell encapsulation with respect to S_2: each sequential module has its own shell that controls its operations. Therefore, the behavior of S_3 is slightly different. The corresponding equations are:

$$\begin{aligned}
x_{n+1} &= \begin{cases} \tau & \text{if } RS_n = \tau \\ b_n - RS_n & \text{if } RS_n \neq \tau \wedge Qb.empty \\ Qb.pop - RS_n & \text{otherwise} \end{cases} \\
Qb_{n+1} &= Qb.push[b_n] \text{ if } \Big((RS_n = \tau) \vee (RS_n \neq \tau \wedge \neg Qb.empty) \Big) \\
y_{n+1} &= \begin{cases} \tau & \text{if } x_n = \tau \\ a_n * x_n & \text{if } x_n \neq \tau \wedge Qa.empty \\ Qa.pop + x_n & \text{otherwise} \end{cases} \\
Qa_{n+1} &= Qa.push[a_n] \text{ if } \Big((x_n = \tau) \vee (x_n \neq \tau \wedge \neg Qa.empty) \Big) \\
RS_{n+1} &= z_n \\
z_{n+1} &= \begin{cases} \tau & \text{if } y_n = \tau \\ c_n + y_n & \text{if } y_n \neq \tau \wedge Qc.empty \\ Qc.pop + y_n & \text{otherwise} \end{cases} \\
Qc_{n+1} &= Qc.push[c_n] \text{ if } \Big((y_n = \tau) \vee (y_n \neq \tau \wedge \neg Qc.empty) \Big)
\end{aligned}$$

where $n \in \mathbb{N}, x_0 = 1, y_0 = 2, z_0 = 0$ and $RS_0 = \tau$.

Table 6.5 reports the behavior of system S_3 which is latency equivalent to the corresponding behaviors of S_1 and S_2. Since the core that computes the values for z is not forced to stall simultaneously with the core computing the values for y, the maximum sustainable throughput of system S_3 is $\vartheta(S_3) = \frac{3}{4}$, i.e. 1.125 times higher than S_2. This speed-up is achieved at the cost of some shell logic duplication. ■

Therefore, shell encapsulation is another variable to consider in designing a latency-insensitive systems. The basic trade-off is between encapsulating together many sequential modules, thereby reducing the shell overhead, and performing a fine-grain shell encapsulation that, in general, helps raising the system throughput. There is no "one-size-fits-all" solution. Different designs require different decisions based on their topology, the number

timestamp n	:	0	1	2	3	4	5	6	7	8	...	
z	:	0	3	$\boxed{\tau}$	2	0	$\boxed{\tau}$	−2	9	$\boxed{\tau}$	...	
RS	:	$\boxed{\tau}$	0	3	$\boxed{\tau}$	2	0	$\boxed{\tau}$	−2	9	...	
b	:	0	2	4	2	8	10	...	...	...	...	
Qb	:		0	2	4	2	8	10	...	...	...	
x	:	1	$\boxed{\tau}$	0	−1	$\boxed{\tau}$	2	2	$\boxed{\tau}$	10	...	
a	:	1	2	3	4	5	6	...	...	...	...	
Qa	:			2	3	4	5	6	...	...	...	
y	:	2	1	1	0	−3	−3	8	10	...	...	
c	:	1	1	0	1	1	0	...	...	...	...	
Qc	:			1	0	1	1	0	...	...	...	...

Table 6.4: The behavior of the latency-insensitive system S_2 in Figure 6.11.

timestamp n	:	0	1	2	3	4	5	6	7	8	...	
z	:	0	3	2	$\boxed{\tau}$	0	−2	9	$\boxed{\tau}$	...	...	
RS	:	$\boxed{\tau}$	0	3	2	$\boxed{\tau}$	0	−2	9	$\boxed{\tau}$	...	
b	:	0	2	4	2	8	10	...	...	...	...	
Qb	:		0	2	4	2	8	10	...	...	...	
x	:	1	$\boxed{\tau}$	0	−1	2	$\boxed{\tau}$	2	10	...	...	
a	:	1	2	3	4	5	6	...	...	...	...	
Qa	:			2	3	4	5	6	...	...	...	
y	:	2	1	$\boxed{\tau}$	0	−3	8	$\boxed{\tau}$	10	...	...	
c	:	1	1	0	1	1	0	...	...	...	...	
Qc	:				0	1	1	0	...	...	...	...

Table 6.5: The behavior of the latency-insensitive system S_3 in Figure 6.11.

and the location of critical channels, and the characteristics of the core modules. The good news is that recycling offers new opportunities for SOC design exploration both in the case when the SOC components are newly designed core modules as well as when pre-designed IP core are extensively reused. In the latter case, even when only a library of *black-box* cores is available, it is still possible to choose between alternative implementations, e.g. a 2-stage pipelined multiplier versus a 5-stage pipelined multiplier.

6.2.2 Recycling Transformations

Given a marked graph model $\mathcal{MG}_S$ modeling a latency-insensitive system implementation S, the *recycling transformations* make it possible to derive automatically a set $\mathcal{R}(S)$ of alternative implementation of S. The elements of $\mathcal{R}(S)$ differ from S for the number and the position of the relay stations as well as the number of shells/core pairs. Hence, from a marked-graph viewpoint, they differ for the number of initially empty places and the number of transitions.

Definition 6.5 *Let $\mathcal{MG}_S = (P, T, F, W, M_0)$ be a marked graph model of a latency-insensitive system implementation S. A* recycling transformation *returning a* recycled *marked graph model of $\mathcal{MG}'_S = (P', T', F', W', M'_0)$ of an alternative latency-insensitive system implementation S' is one of the followings:*

- relay station insertion*: a transition t_j and an initially empty place p_j are inserted after a place $p_i \in P$ with $p_i \bullet = \{t_k\}$ such that*

$$\begin{aligned} T' &= T \cup \{t_j\} \\ P' &= P \cup \{p_j\} \\ F' &= F \cup \left\{(p_i,t_j),(t_j,p_j),(p_j,t_k)\right\} \setminus (p_i,t_k) \\ W' &= F' \rightarrow 1 \\ \forall p \in P \left(M'_0(p)\right. &= \left.M_0(p)\right) \wedge M'_0(p_j) = 0 \end{aligned}$$

- relay station removal*: a transition t_j and an initially empty place p_j with $(t_j, p_j) \in F$,*

$M_0(p_j) = 0$) $\bullet t_j = \{p_i\}$, and $p_j\bullet = \{t_k\}$ are removed from $\mathcal{MG}_S$ such that

$$\begin{aligned}
T' &= T \setminus \{t_j\} \\
P' &= P \setminus \{p_j\} \\
F' &= F \cup (p_j,t_k) \setminus \Big\{(p_j,t_i),(t_i,p_i),(p_i,t_k)\Big\} \\
W' &= F' \rightarrow 1 \\
\forall p \in P' \Big(M_0'(p) &= M_0(p)\Big)
\end{aligned}$$

- shell splitting: *an initially non-empty place p_j and a transition t_j are inserted after a transition t_i with $\forall p \in t\bullet \big(M_0(p) = 1\big)$ such that*

$$\begin{aligned}
T' &= T \cup \{t_j\} \\
P' &= P \cup \{p_j\} \\
F' &= F \cup \Big\{(t_i,p_j),(p_j,t_j)\Big\} \cup \Big\{(t_j,p) | (t_i,p) \in F\Big\} \setminus (t_i,p) \\
W' &= F' \rightarrow 1 \\
\forall p \in P \Big(M_0'(p) &= M_0(p)\Big) \wedge M_0'(p_j) = 1
\end{aligned}$$

- shell collapsing: *a transition t_j with $\forall p \in t\bullet \big(M_0(p) = 1\big)$ is removed from $\mathcal{MG}_S$ such that*

$$\begin{aligned}
T' &= T \setminus \{t_j\} \\
P' &= P \setminus t_j\bullet \\
F' &= F \setminus \Big\{(t_j,p) | p \in t_j\bullet\Big\} \\
W' &= F' \rightarrow 1 \\
\forall p \in P' \Big(M_0'(p) &= M_0(p)\Big)
\end{aligned}$$

and $\forall q_{in}, q_{out} \in Q(\mathcal{MG}_S)$ with

$$head(q_{in}) = t_j \wedge tail(q_{in}) = t_i \wedge tail(q_{out}) = t_j \wedge head(q_{out}) = t_k$$

if $\nexists q \in Q(\mathcal{MG}_S)$ with $tail(q) = t_i \wedge head(q) = t_k$ then

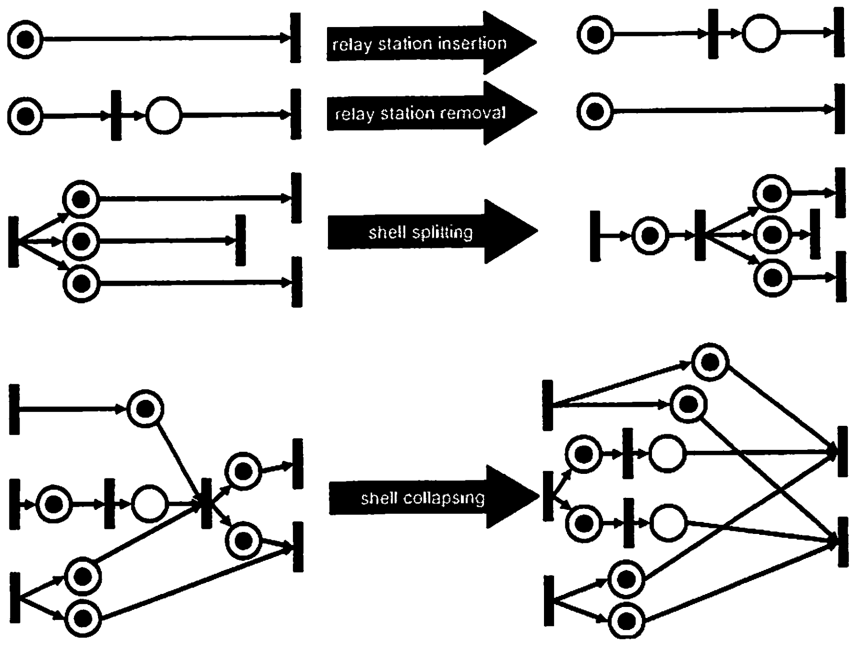

Figure 6.12: Examples of recycling transformations.

– *if* $(t_i\bullet = \bullet t_j = p)$ *then*

$$\begin{aligned} P' &= P' \cup \{p'\} \setminus \{p\} \\ F' &= F \cup \left\{(t_i,p'),(p',t_k)\right\} \setminus \left\{(t_i,p),(p,t_j)\right\} \\ M'_0(p') &= 0 \end{aligned}$$

– *if* $(t_i\bullet \neq \bullet t_j)$ *then* $\left(\textit{assuming } int(q_{in}) = \{t_1,t_2,\dots,t_l\} \textit{ and } l = \lambda(q_{in}) - 1\right)$

$$\begin{aligned} T' &= T' \cup \{t'_1,t'_2,\dots,t'_l\} \setminus \{t_1,t_2,\dots,t_l\} \\ P' &= P' \cup \{p'_0,p'_1,\dots,p'_l\} \setminus \{p_0,p_1,\dots,p_l\} \\ F' &= F \cup \left\{(t_i,p'_0),(p'_0,t'_1),(t'_1,p'_1),\dots,(t'_l,p'_l),(p'_l,t_k),\right\} \\ &\quad \setminus \left\{(t_i,p_0),(p_0,t_1),(t_1,p_1),\dots,(t_l,p_l),(p_l,t_k),\right\} \\ \forall m \in [1,l] \left(M'_0(p_i)\right. &= \left. 0\right) \wedge M'_0(p'_0) = 1 \end{aligned}$$

Figure 6.12 illustrates the four recycling transformations. Relay station insertion and removal only change the latency of the channel q of $\mathcal{MG}_S$ where they are applied (i.e. this latency is incremented or decremented by one unit). The recycled marked graph $\mathcal{MG}'_S$ has the same cycle structure as $\mathcal{MG}_S$ and, by Definition 6.2, is latency equivalent to it. After shell splitting the recycled marked-graph model $\mathcal{MG}'_S$ has also the same cycle structure as $\mathcal{MG}_S$: two additional channels $q_{\{t_i,t_j\}}$ and $q_{\{t_j,t_k\}}$ that belong to exactly the same cycle as $q_{\{t_i,t_k\}}$. After shell collapsing the recycled marked graph $\mathcal{MG}'_S$ is still cyclic but may have less cycles than $\mathcal{MG}_S$ since, while removing the channels passing through t_j and creating new channels to connect each up-link tail transition t_i to each down-link head transition t_k, it may be the case that a channel $q_{\{t_i,t_k\}}$ is already present in $\mathcal{MG}_S$. Then a new channel is not necessary (a marked-graph model is not a multi-graph). When shell collapsing is performed the functionality of the core process is duplicated as many times as the number of its output channels and each copy is inserted into the corresponding down-link shell. Conversely, shell splitting amounts to divide the functionality of a core modules s into two new core modules s_1 and s_2 and to encapsulate them in separate back-to-back shells. This division can be performed only if the two resulting modules are both sequential elements, i.e. they do not present a combinational path from their inputs to their outputs. Therefore, for a generic block the division is performed along an "internal cut" of storage elements of s. From an IP core library viewpoint, the division could correspond to choose a four-stage pipelined version of a module instead of a two-stage one and then put the first two stages in a shell and the last two in another.

Shell collapsing and shell splitting return a recycled marked-graph model $\mathcal{MG}'_S$ that, strictly speaking by Definition 6.2, would not be latency-equivalent to $\mathcal{MG}_S$. In practice, they are latency-equivalence because putting two back-to-back sequential cores into the same shell gives a process that is latency-equivalent to the sub-system given by the serial composition of the original ones.

The recycling transformations can be unambiguously distinguished based on their effect on the cycle metric $\mu(c) = \frac{b_k}{a_k}$ of each cycle c of $\mathcal{MG}_S$ that contains the channel q which is involved in the transformation. The cycle metric $\mu(c')$ of the corresponding cycle c' in the recycled marked-graph model $\mathcal{MG}_S$ changes as follows:

- *relay station insertion*: $\mu(c') = \frac{b_k + 1}{a_k}$
- *relay station removal*: $\mu(c') = \frac{b_k - 1}{a_k}$
- *shell splitting*: $\mu(c') = \frac{b_k + 1}{a_k + 1}$
- *shell collapsing*: $\mu(c') = \frac{b_k - 1}{a_k - 1}$

Hence, the transformations all share the property of *monotonically* affecting the cycle metrics to which the targeted channel belongs: relay station insertion and shell collapsing increase the cycle metric of c' while relay station removal and shell splitting decrease it [8]. Naturally, if c' is the only critical cycle of $\mathcal{MG}_S$ these transformations have an impact on its throughput.

Since all the transition of a marked graph model have unit delay the numerator of cycle metric $\mu(c)$ coincides with the number of transitions in c, i.e. its cardinality $|c|$. Hence, Definition 5.8 can be rewritten as follows.

Definition 6.6 *The* extended form of the cycle metric *of a cycle $c \in C(\mathcal{MG})$ in a marked graph model $\mathcal{MG} = (P, T, F, W, M_0)$ is written as*

$$\mu(c) = \frac{\sum_{t_i \in T} x(t_i, c) \cdot n(t_i)}{\sum_{p_j \in P} y(p_j, c) \cdot m(p_j)}$$

where $\forall t_i \in T$, $\forall p_j \in P$

- $n(t_i) = 1$
- $m(p_j) = \begin{cases} 0 & \text{if } p_j \in P_0 \\ 1 & \text{otherwise} \end{cases}$
- $x(t_i, c) = \begin{cases} 1 & \text{if } t_i \in c \\ 0 & \text{otherwise} \end{cases}$

[8]To be precise, shell splitting (collapsing) decreases (increases) the cycle metric only if this is greater than one, while leaves it equal to one otherwise.

- $y(p_j, c) = \begin{cases} 1 & \textit{if } p_j \in c \\ 0 & \textit{otherwise} \end{cases}$

In the above definition functions x and y capture analytically how the transitions and the places are shared among intersecting cycles of $\mathcal{MG}_S$. Then, for each transition t_i of $\mathcal{MG}_S$, the term $x(t_i, c) \cdot n(t_i)$ accounts for the unit latency contribution associated to t_i (in case t_i belongs to c). Similarly, for each place p_j of $\mathcal{MG}_S$ the term $y(p_j, c) \cdot m(p_j)$ accounts for the throughput contribution associated to place p_j if this place belongs to c and if it presents a token in the initial marking M_0 (i.e. does not belong to the set P_0 of initially empty places).

For the purpose of tracking the impact on the cycle metrics, the transformation of a marked-graph model $\mathcal{MG}_S = (P, T, F, W, M_0)$ into a recycled marked graph $\mathcal{MG}'_S = (P', T', F', W', M'_0)$ can be expressed based on the structure of $\mathcal{MG}_S$ with the help of two integer-value functions, $n'(t_i) : T \rightarrow [-1, 1]$ and $m'(p_j) : P \rightarrow [-1, 1]$, which encode the following actions:

- $n'(t_i) = -1$: remove transition t_i, i.e. $T' = T \setminus \{t_i\}$;
- $n'(t_i) = 0$: leave transition t_i untouched, i.e. $T' = T$;
- $n'(t_i) = 1$: add a transition t'_i next to t_i, $T' = T \cup \{t'_i\}$;
- $m'(p_j) = -1$: remove an initially non-empty place p_j, i.e. $P' = P \setminus \{p_j\}$;
- $m'(p_j) = 0$: leave place p_j untouched, i.e. $P' = P$;
- $m'(p_j) = 1$: add an initially non-empty place p_j, i.e. $P' = P \cup \{p_j\}$;

Then, the recycling transformation can be encoded using functions $n'(t_i), m'(p_j)$ as shown in Table 6.6.

The recycling transformations can be applied independently or together. In both cases the cycle metrics of the cycles in the recycled marked-graph model can be expressed in closed form using functions $n'(t_i), m'(p_j)$ and the functions of Definition 6.6.

recycling transformation	$n'(t_i)$			$m'(p_j)$		
	-1	0	$+1$	-1	0	$+1$
relay station insertion			$\star$			
relay station removal	$\star$					
shell splitting			$\star$			$\star$
shell collapsing	$\star$			$\star$		

Table 6.6: Encoding of recycling transformations.

Definition 6.7 *The* recycled cycle metric *of cycle $c \in C(\mathcal{MG}')$ in marked-graph model $\mathcal{MG}'_S = (P', T', F', W', M'_0)$ obtained recycling marked-graph model $\mathcal{MG} = (P, T, F, W, M_0)$ is given by*

$$\mu_{\mathcal{R}}(c) = \frac{\sum_{t_i \in T} x(t_i, c) \cdot \left[n(t_i) + n'(t_i)\right]}{\sum_{p_j \in P} y(p_j, c) \cdot \left[m(p_j) + m'(p_j)\right]}$$

Thanks to the relationship between the cycle metrics and the cycle time of $\mathcal{MG}'_S$, the collection of recycling transformations represents an effective toolkit to carry a design exploration of the latency-insensitive system under the guidance of latency/throughput trade-offs. In, the following section I define the *cycle balancing problem* as the formal structure to perform such design exploration.

6.2.3 The Cycle Balancing Problem

Given a marked graph $\mathcal{MG}_S$ modeling a latency-insensitive system implementation S, the cycle balancing problem is defined as follows:

Problem 6.1 (Cycle Balancing Problem (CPB))

Given: *A marked graph $\mathcal{MG}_S = (P, T, F, W, M_0)$ with a set of cycles $C(\mathcal{MG}_S)$.*

Minimize: *The cost*

$$g = \sum_{t_i \in T} |n'(t_i)| + \sum_{p_j \in P} |m'(p_j)|$$

over all integer variables $n'(t_1), \ldots, n'(t_{|T|})$ *and* $m'(p_1), \ldots, m'(p_{|P|})$, *with*

$$\forall i \in [1, |T|], (n'(t_i) \in [-1, 1] \ \wedge \ \forall j \in [1, |P|], (m'(p_j) \in [-1, 1]$$

Subject to: $\exists q \in \mathbf{Q}^+, q \geq 1, \ \forall k \in [1, |C(\mathcal{MG}_S)|]$:

$$\frac{\sum_{t_i \in T} x(t_i, c_k) \cdot \left[n(t_i) + n'(t_i)\right]}{\cdot \sum_{p_j \in P} y(p_j, c_k) \cdot \left[m(p_j) + m'(p_j)\right]} = q$$

where functions $n(t_i), m(p_j), x(t_i, c_k), y(p_j, c_k)$ *are those defined in Definition 6.6.*

□

It is easy to see that the cycle balancing problem has always at least a trivial solution. In fact, the variable configuration that is obtained by setting $\forall p_j$, $\left(m'(p_j) = 1 \Leftrightarrow m(p_j) = 0\right)$ and $\forall t_i$, $(n'(t_i) = 0)$ satisfies the problem constraints with $q = 1$. In general several solutions are possible, with the same or different values of q. Each solution corresponds to a distinct configuration ς of the integer variables $n'(t_1), \ldots, n'(t_{|T|})$ and $m'(p_1), \ldots, m'(p_{|P|})$ which unambiguously identifies a new marked graph $\mathcal{MG}'_S$ whose cycles have all the same cycle metric. Marked-graph $\mathcal{MG}'_S$ models an alternative, balanced, latency-equivalent system implementation.

Algorithm CYCLEBALANCER, presented in the Appendix, returns a list of solutions including the optimal one. The list is ranked in order of optimality, i.e. by increasing values of the cost function g. Hence, the optimality criterion is to "to minimize the number of recycling transformations while equalizing the cycle metrics". This criterion is ideal if the CYCLEBALANCER algorithm is executed repeatedly as part of an interactive design environment that aims at assisting SOC designers to complete a design exploration as a sequence of refinements steps. As portions of the design start to crystallize and the designers converge toward the final implementation, the number of "proposed" changes and their magnitude naturally diminishes. An alternative criterion could be "to minimize the equalized cycle metric", i.e. to maximize the throughput of $\mathcal{MG}$. In fact, CYCLEBALANCER accepts as input parameters the specification of ranges in order to constrain the values for the corresponding variables.

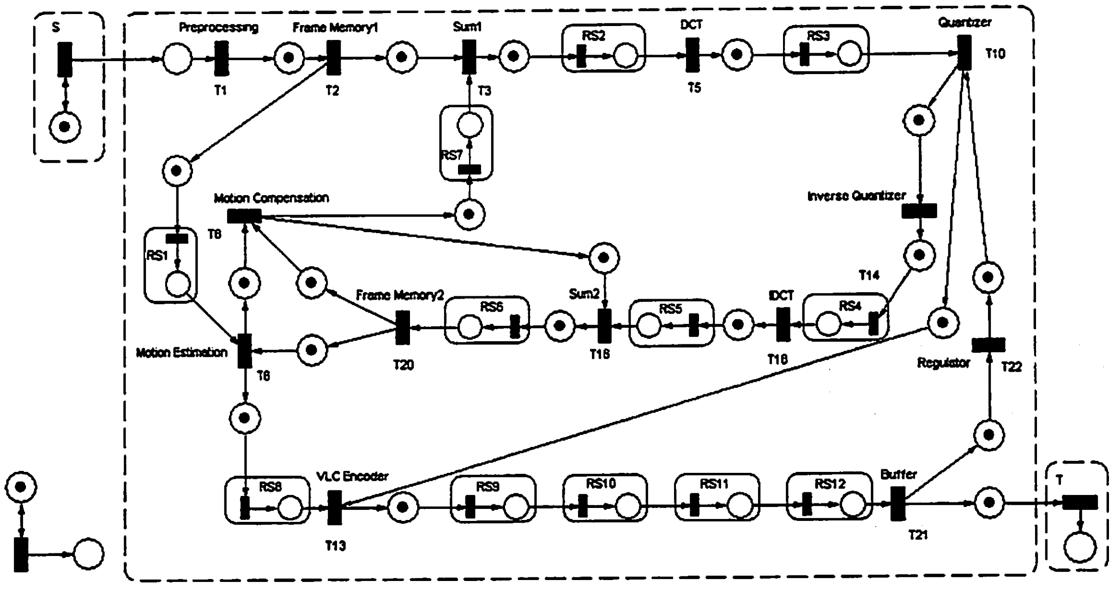

Figure 6.13: Un-balanced implementation of the MPEG-2 video encoder of Figure 6.8 with $\vartheta(\mathcal{MG}_S) = 0.50$.

6.2.4 Case Study: an MPEG Video Encoder. Part Two

Figure 6.13 illustrates a marked-graph $\mathcal{MG}_S$ modeling an implementation S of the MPEG-2 video encoder of Figure 6.8. With respect to the reference model $\mathcal{MG}^\infty$ of Figure 6.9, discussed in Section 6.1.5, this implementation contains twelve relay stations distributed around the channels of the system. Table 6.7 describes the six cycles in the system after the insertion of relay stations. This table can be compared to the corresponding Table 6.1 for the reference implementation. Each cycle has a different cycle metric varying in the interval $[1.25, 2]$ from $\frac{5}{4}$ of cycle c_2 to $\frac{8}{4}$ of cycle c_3. Consequently, c_3 is the critical cycle and $\mathcal{MG}_S$ has maximum sustainable throughput $\vartheta(\mathcal{MG}_S) = 0.5$. Figure 6.14 illustrates the behavior of the system operating in an environment with a unit token-production rate (i.e. an environment that is not equalized with respect to S). After one hundred timestamps, the system has been able to process only half of the tokens and there is unbounded token accumulation in correspondence of the *sum-1* module and the *motion estimation* module. Therefore, to have a correct, albeit relatively poor, behavior it is necessary to equalize the environment by reducing its throughput to 0.5. The resulting scenario is shown in Fig-

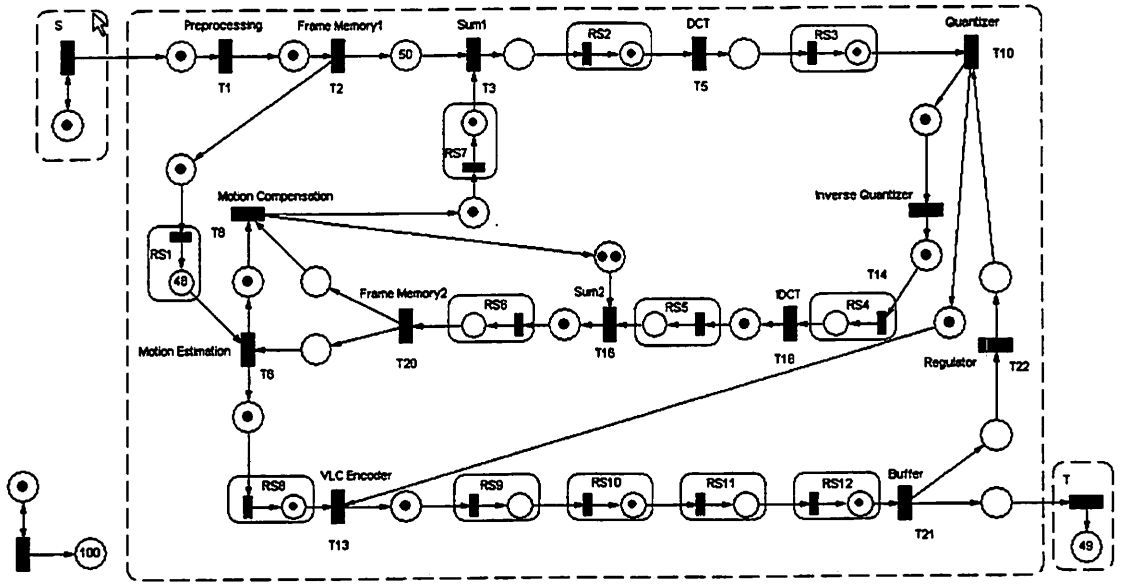

Figure 6.14: The marked graph of Figure 6.13 after 100 timestamps without equalized environment.

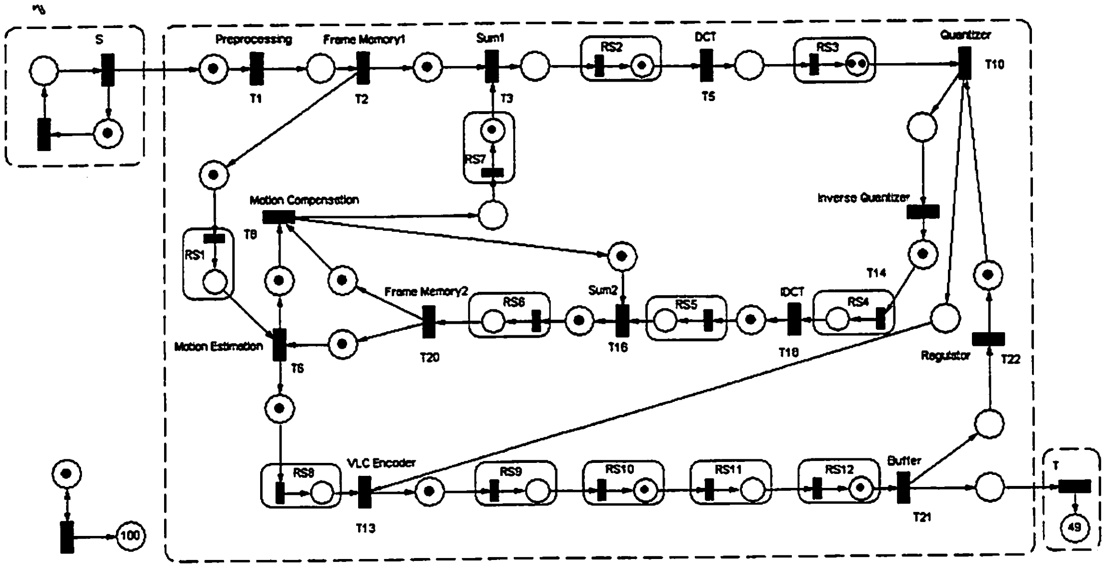

Figure 6.15: The marked graph of Figure 6.13 after 100 timestamps with equalized environment.

c_i	cycle transitions	$M_0(c_i)$	$\mu(c_i)$	$\frac{1}{\mu(c_i)}$
c_1	$\{t_8, t_{16}, rs_6, t_{20}\}$	3	4/3	0.75
c_2	$\{t_6, t_8, t_{16}, t_{20}\}$	4	5/4	0.80
c_3	$\{t_{10}, t_{13}, rs_9, rs_{10}, rs_{11}, rs_{12}, t_{21}, t_{22}\}$	4	8/4	0.50
c_4	$\{t_{10}, t_{14}, rs_4, t_{18}, rs_5, t_{16}, rs_6, t_{20}, t_8, rs_7, t_3, rs_2, t_5, rs_3\}$	8	14/8	0.57
c_5	$\{t_{10}, t_{14}, rs_4, t_{18}, rs_5, t_{16}, rs_6, t_{20}, t_6, t_8, rs_7, t_3, rs_2, t_5, rs_3\}$	9	15/9	0.60
c_6	$\{t_{10}, t_{14}, rs_4, t_{18}, rs_5, t_{16}, rs_6, t_{20}, t_6, rs_8, t_{13}, t_{21}, t_{22}\}$	9	17/9	0.53

Table 6.7: Cycles and cycle times for the marked graph of Figure 6.13.

ure 6.15. Still $\mathcal{MG}_S$ is not balanced since its cycles have different cycle metrics.

The application of recycling transformation to $\mathcal{MG}_S$ under the constraints of the cycle balancing problem makes it possible to consider alternative, and hopefully better, implementations: for instance, the balanced implementation illustrated in Figure 6.16. This implementation is obtained with the application of thirteen distinct recycling transformations: three "shell collapsing", two "shell splitting", six relay station insertions, and two relay station removals. However, the combination of the three "shell collapsing" and the two "shell splitting" boils down to just two "shell collapsing" with one core duplication (the *motion estimation* module). Also, two of the relay station insertions together with the two relay station removals can be simply interpreted as *pushing around two relay stations*. In summary, the final list of the design changes is the followings:

- the *motion estimation* core module is duplicated: one copy remains as a stand-alone shell/core pair (t_6), while the other is encapsulated within the same shell as the *VLC encoder* (t_{13});

- the *sum-1* core module and the *DCT* core module are encapsulated together within the same shell (t_5);

- relay station rs_4 is moved from channel $q_{\{t_{14},t_{18}\}}$ between the *inverse quantizer* shell and the *IDCT* shell to channel $q_{\{t_{20},t_6\}}$ between the *frame memory-2* shell and the *motion estimation* shell;

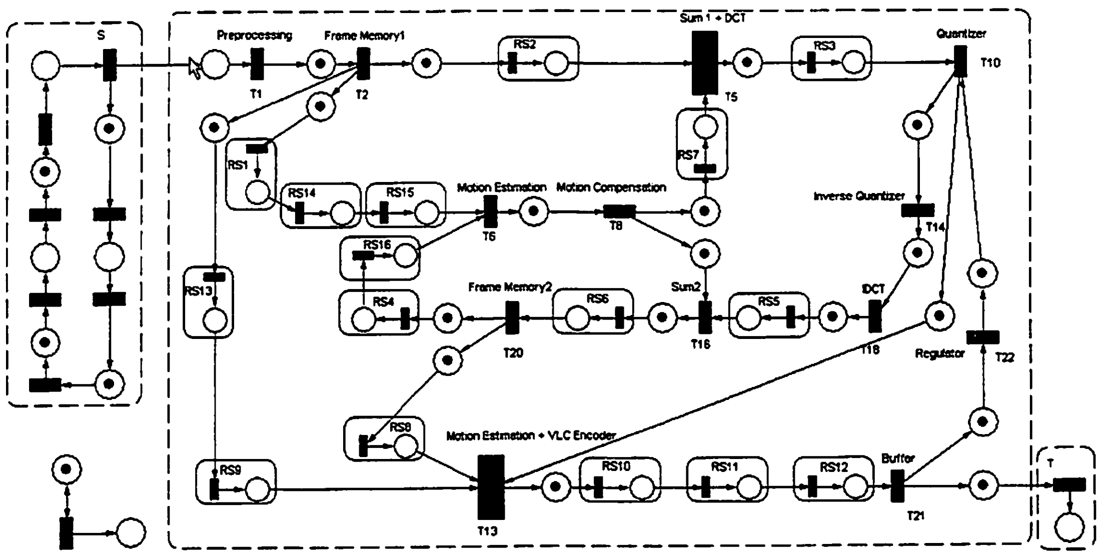

Figure 6.16: Balanced implementation of the marked graph of Figure 6.13 with $\vartheta(\mathcal{MG}) = 0.57$ obtained via recycling.

- relay station rs_9 is moved from channel $q_{\{t_{13},t_{21}\}}$ between the *motion estimation & VLC encoder* shell and the *buffer* shell to channel $q_{\{t_2,t_{13}\}}$ between the *frame memory-1* shell and the *motion estimation & VLC encoder* shell;
- relay station rs_{13} is added on channel $q_{\{t_2,t_{13}\}}$ between the *frame memory-1* shell and the *motion estimation & VLC encoder* shell;
- relay stations rs_{14} and rs_{15} are added on channel $q_{\{t_2,t_6\}}$ between the *frame memory-1* shell and the *motion estimation* shell;
- relay station rs_{16} is added on channel $q_{\{t_{20},t_6\}}$ between the *frame memory-1* shell and the *motion estimation* shell;

Figure 6.17 shows the marked graph modeling the balanced system implementation $\mathcal{MG}'_S$ after one hundred timestamps have passed.

The result of this recycling is an implementation $\mathcal{MG}'_S$ of the latency-insensitive system S that:

- is balanced since the cycle metric of every cycle is a multiple of $\frac{7}{4}$;

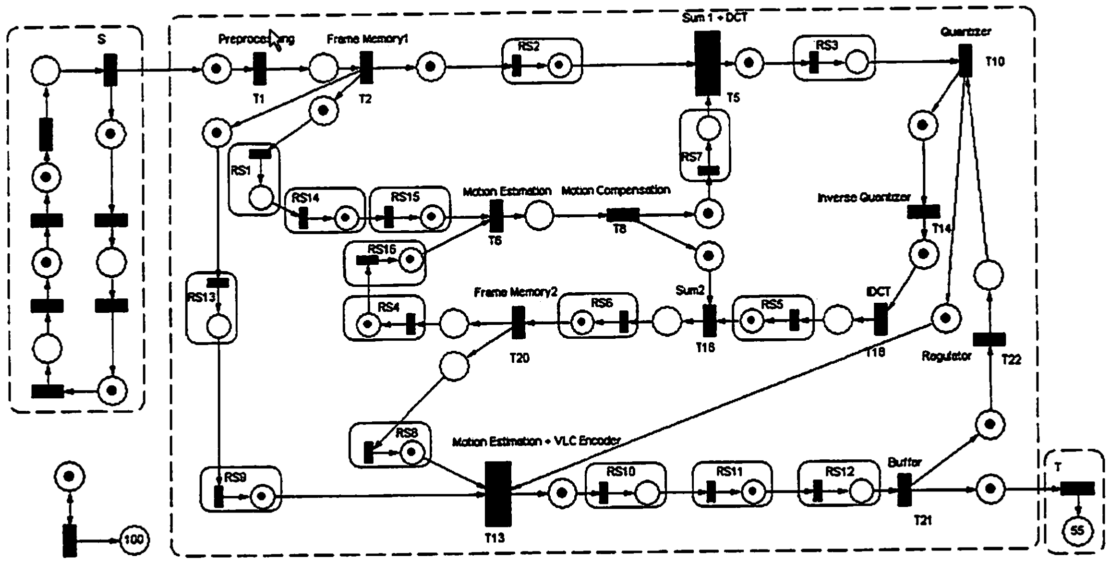

Figure 6.17: The marked graph of Figure 6.16 after 100 timestamps.

- even tough it presents 33% more relay stations than the unbalanced implementation $\mathcal{MG}_S$, it offers a maximum sustainable throughput that is 1.143 times higher than $\vartheta(\mathcal{MG}_S)$;

Figure 6.18 shows the model of another alternative implementation $\mathcal{MG}_S''$ of the latency-insensitive system S that is obtained solving the same cycle balancing problem. This implementation is the result of the application of nine distinct recycling transformations: one "shell collapsing", two "shell splitting", two relay station insertions, and four relay station removals. In this case the three shell transformations are not combined, while again the two relay station insertions together with two relay station removals can be simply interpreted as pushing around two relay stations. The final list of the design changes with respect to implementation $\mathcal{MG}_S$ is the following:

- the *inverse quantizer* core module is split into two stages *inverse quantizer stage-1* (t_{14}) and *inverse quantizer stage-2* (t_{14bis}) each encapsulated in a separate shell;

- the *motion estimation* core module is split into two stages *motion estimation stage-1* (t_6) and *motion estimation stage-2* (t_{6bis}) each encapsulated in a separate shell;

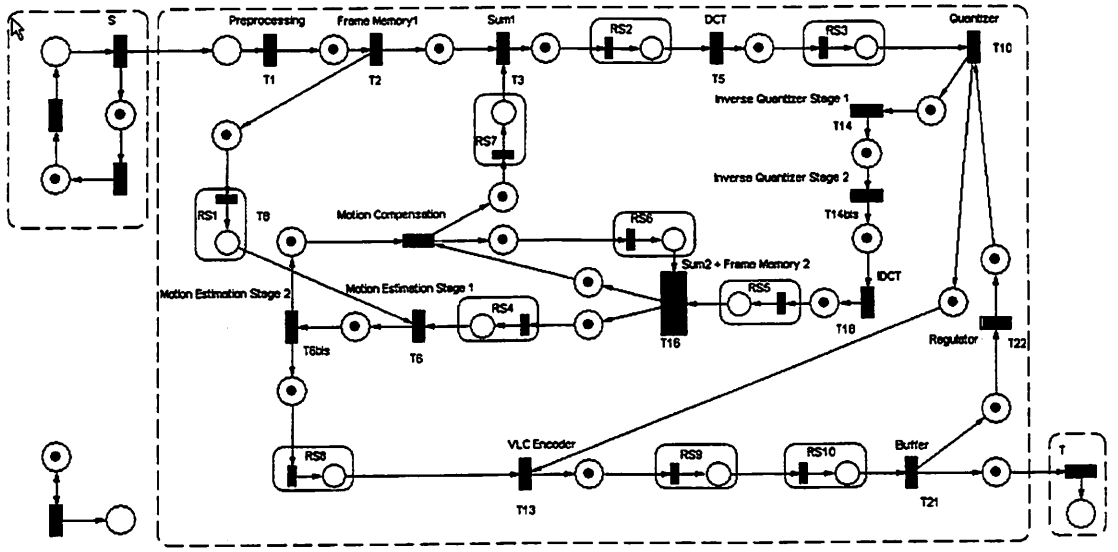

Figure 6.18: Balanced implementation of the marked graph of Figure 6.13 with $\vartheta(\mathcal{M}\mathcal{G}) = 0.66$ obtained via recycling.

- the *frame memory-2* core module and the *sum-2* core module are encapsulated together within the same shell (t_{16});

- relay station rs_4 is moved from channel $q_{\{t_{14},t_{18}\}}$ between the *inverse quantizer* shell and the *IDCT* shell to channel $q_{\{t_{16},t_6\}}$ between the *sum-2 & frame memory-2* shell and the *motion estimation* shell;

- relay station rs_6 is moved from channel $q_{\{t_{16},t_{20}\}}$ between the *sum-2* shell and the *frame memory-2* shell to channel $q_{\{t_8,t_{16}\}}$ between the *motion compensation* shell and the *sum-2 & frame memory-2* shell;

The result of this recycling is a balanced implementation $\mathcal{M}\mathcal{G}''_S$ of the latency-insensitive system S with the following characteristics:

- it is balanced since the cycle metric of every cycle is a multiple of $\frac{3}{2}$;

- it presents 20% less relay stations than the unbalanced implementation $\mathcal{M}\mathcal{G}_S$, and it offers a maximum sustainable throughput that is 1.333 times higher than $\vartheta(\mathcal{M}\mathcal{G}_S)$;

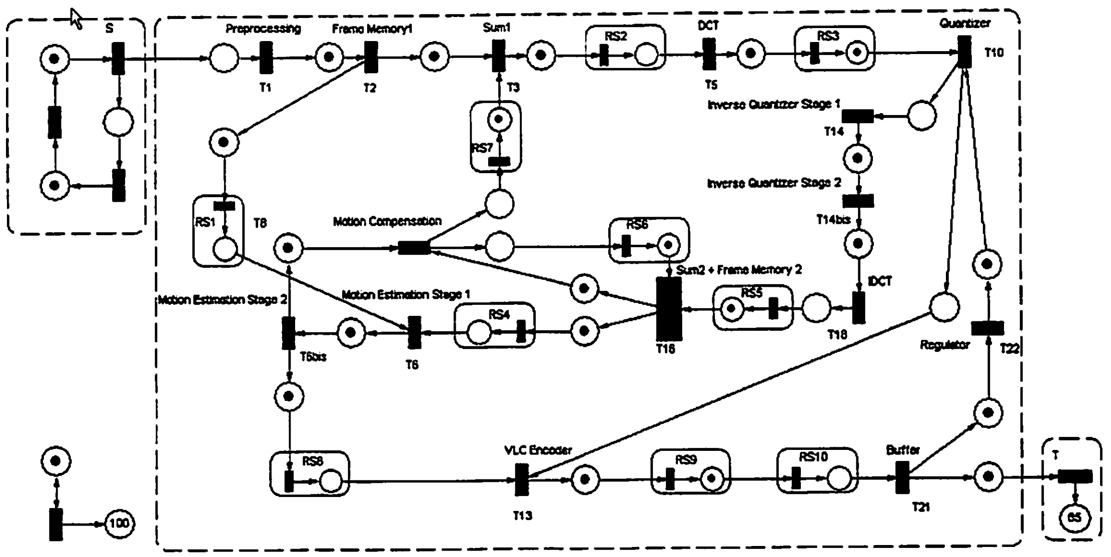

Figure 6.19: The marked graph of Figure 6.18 after 100 timestamps.

Figure 6.19 shows the marked graph modeling the balanced system implementation $\mathcal{M}\mathcal{G}_S''$ after one hundred timestamps have passed.

This case study illustrates the capability of applying the recycling transformation under the constraints of the cycle balancing problem. The first balanced implementation $\mathcal{M}\mathcal{G}_S'$ demonstrates a point which is slightly counterintuitive: the insertion of several relay stations together with a reorganization of those already present may result in a significant increase of throughput. If this is properly coupled—by means of a careful layout of the modules—with the decrease in the period of the nominal clock that channel pipelining naturally offers, the final performance improvement may be striking.

The second balanced implementation $\mathcal{M}\mathcal{G}_S''$ reports a large throughput increase but, in doing so, it also removes two of the twelve relay stations originally presented in the system. Clearly, this is possible only if the corresponding channel is shortened by reducing the distance between the source and the sink module. This confirms the need for applying recycling in the context of a global SOC design exploration strategy that combines the choice of the right version of the IP core for each module with a careful organization of the system topology. In latency-insensitive design engineers find the structure to address these challenges globally and formally, with means other than simulation and intuition.

6.3 Related Work

Software designers have long known that functionality and performance are orthogonal requirements that can be satisfied independently of each other by optimizing small modules and using a model of computation, e.g. Kahn process networks [122], that ensures the independence of computed results on the performance of each module. The idea of recycling latency-insensitive systems follows the same principle, which is ultimately the principle of orthogonalization of concerns. Consequently the combination of latency-insensitive design together with recycling provides the means for performance engineering in SOC design. So far, performance and functionality have been intertwined in the ASIC flow that is centered around RTL synthesizable hardware-description language (HDL) specifications (as discussed in Section 2.2.3). Recycling offers a capability which has been long available within a clock cycle, by means of various combinational logic speed-up techniques, but which has been prohibitively complex to manage at the sequential logic level. Traditional modeling paradigms based on control/data-flow graph [107] or extended finite state machine [50] break down when they need to confront a delay of "one more or one less clock cycles" in one of the components of a complex synchronous circuit. Asynchronous techniques, which could represent a more natural way to manage this problem, are too far from mainstream use to be applicable in an ASIC/SOC design flow and they suffer from the drawbacks discussed in Section 3.5.1. Still, designers know that when the latency of a circuit is about 20*ns*, they can split it into four or five pipeline stages, thus achieving either 200 or 250*MHz* operation. The problem is that no design automation technique today supports them in this task. Previously, even high-level synthesis approaches that considered latency as a parameter, to optimize area and throughput, were looking only at one process at a time. Recycling considers multiple processes simultaneously by modeling the global interaction among the processes. Khouri *et al.* looked at process interaction locally, in a greedy fashion [131]. On the other hand, Mathur *et al.* considered only rate analysis, rather than using rates for efficient synthesis and optimization [161].

Recycling can be seen as a generalization of the well-known concept of retiming [150], since it includes the performance aspect of interconnection into the retiming graph, and makes it possible to optimize *simultaneously* the performance impact of more or less pipelining and of more or less communication latency. On the other hand, while the number of latches distributed along a cycle of the sequential circuit cannot change during retiming, recycling theoretically enables the insertion of an arbitrary number of relay stations on any wire of a latency-insensitive system.

In general, the insertion of relay stations can be completed with an automatic tool as part of the physical design process (similarly to the *buffer insertion* techniques available in current design flows [7, 59, 90]). For the derivation of the final chip implementation, recycling can be combined with the result of recent works on throughput-driven on-chip communication synthesis [152, 211].

6.4 Concluding Remarks

I applied the constructive modeling techniques introduced in Chapter 5 to the particular problem of channel pipelining and to the general issue of finding balanced implementations for latency-insensitive systems.

The channels of a latency-insensitive system can be used to model sets of global wires on the chip. The idea is to build a distributed communication architecture that relies on a collection of point-to-point elastic pipelined channels instead of centralized communication resources. In this regard, the added value of latency-insensitive design is twofold: (1) wire pipelining can be performed automatically without changing the interface logic of the components and (2) the logic controlling the channels is more robust to the delay variations of global interconnect because it is implemented in a distributed fashion by the latency-insensitive protocol.

The topology of the latency-insensitive system plays a critical role with respect to channel pipelining as the insertion of a relay station on a local channel can have a negative impact on the system throughput. I presented design guidelines to optimize the system performance based on the analysis of its topology. In the derivation of the final layout, physical design EDA tools should try to keep close together modules that belong to the same strongly connected component (SCC), while interacting SCCs can be arbitrarily apart since the inter-SCC communication channels can be arbitrarily pipelined without affecting the system performance. Also, within a SCC, the optimization criterion should be to keep close together those components that communicate by means of channels that belong to cycle of relatively small cardinality. Components communicating by means of channels that belong to big cycles (i.e. highly pipelined feedback paths) can be fairly distributed.

In general, the latency-insensitive protocol can be implemented with or without back-pressure. Back-pressure is necessary for the design of open systems and offers a modular implementation with a design overhead that is simple to estimate. Finite queues without back-pressure can be used for the implementations of systems that are not open and as long as all components in the system process data at the same throughput. This raises the issue of throughput equalization which is intertwined with the issue of achieving a well-balanced implementation of a latency-insensitive system while performing channel pipelining.

To explore alternative design implementations, I defined the concept of recycling, i.e. the combined application of three design transformations: inserting relay stations (channel pipelining), moving them across shell-core pairs, and redrawing the boundaries of the shells around the cores. To optimize the application of recycling and find the right balance between communication and computation latencies, I defined the cycle balancing problem (an algorithm to solve it is presented in the appendix). This is the basis for developing an interactive design framework that automatically analyzes the impact of *local* design decisions on the performance of the *global* system and synthesizes alternative better solutions ranked in order of optimality. The target is the achievement of a balanced design that will enhance the average-case performance of the system.

Recycling can be seen also as an extension to system-level design of retiming, a classic circuit-level optimization techniques. Further, the combination of retiming (a sequential module) and recycling (a system of sequential modules) opens interesting research avenues for sequential circuit optimization. It is the subject of Chapter 7.

Chapter 7

Recycling Synchronous Circuitry

In which unpromising attempts to save time reveal a surprising outcome.

IN this chapter I investigate the application of latency-insensitive design and recycling to the optimization of synchronous circuitry. Latency-insensitive design is inherently a system-level design methodology and recycling is a technique to optimize the pipelining of global wires in a latency-insensitive system with respect to the overall system performance. Still, recycling has some similarities with retiming, the classic technique to optimize sequential circuits at the gate level. Retiming essentially moves latches across logic gates and in doing so changes the number of latches and the longest combinational path delay between them. The goal of retiming is to reduce the critical path within a sequential module and/or reduce the area occupied by latches. Recycling adds and moves relay stations across patient sequential modules with the goal of segmenting long interconnect wires. Theoretically, as I first explained in [40], recycling can also be applied at gate-level in order to optimize the performance of sequential circuits beyond what can be achieved by retiming. In fact, the two techniques can be combined to further enhance the performance of a circuit beyond what is achievable through the application of retiming for speed optimization. I present here a theoretical framework to guide the simultaneous application of recycling are retiming. This model identifies the conditions under which a retimed synchronous circuit can be further sped-up and determines the amount of the resulting performance gain.

7.1 Retiming

Retiming is a classic logic optimization technique for synchronous circuits. Leiserson, Rose and Saxe first introduced retiming to optimize systolic systems [147, 148, 151]. In a subsequent work [149, 150], however, Leiserson and Saxe fully revisited the concept of retiming and showed how generic synchronous circuits can benefit from it under three main optimality criteria:

1. minimize the circuit clock period by adding/removing storage elements;
2. minimize the circuit area by reducing the number of storage elements;
3. minimize the circuit area under a maximum clock-period constraint.

The material in this chapter addresses mainly the first criterion.

In the last two decades, retiming has been adopted as a key optimization technique within every major logic synthesis tool both in academia and industry. In [212], Shenoy discusses the issues that arise in practical implementations of retiming and the research efforts to extend the domain of circuits for which it can be applied. In particular, Shenoy and Rudell have improved the efficiency of retiming so that circuits up to 50 thousand equivalent gates can be retimed for minimum area under a delay constraint [213]. More recently, retiming has been successfully applied to FPGA design [60, 217, 243].

7.1.1 The Basic Idea of Retiming

I present the basic idea of retiming borrowing the original example, shown in Figure 7.1, from [149, 150]. The two graphs in this figure represent two synchronous circuits that are functionally equivalent and can be obtained one from the other via retiming. *Functional equivalence* means that the circuits have the same behavior from the host system viewpoint, i.e. when solicited by the same input trace they present *exactly* the same output trace. Hence, functional equivalence is different from latency equivalence, defined in Section 3.2.3, as well as stuttering equivalence, discussed in Section 3.5.5.

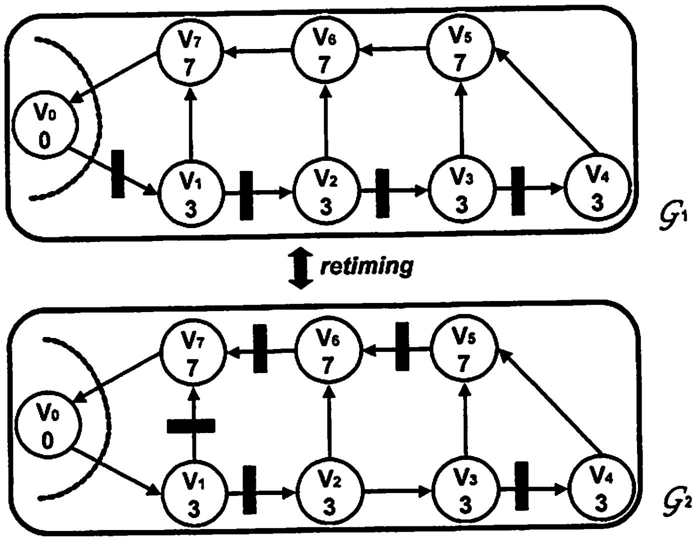

Figure 7.1: Retiming the correlator circuit [150]: the top graph has $\psi(\mathcal{G}_1) = 24$ while the bottom one has $\psi(\mathcal{G}_2) = 13$ (shading highlights critical paths).

Definition 7.1 *A* circuit graph *is a tuple $\mathcal{G} = (V, A, d, w)$, where V is the set of vertices (the combinational circuits), A is the set of arcs (the wires between combinational circuits), d is a vertex labelling function (the largest combinational delay of the corresponding circuit) and w is an arc weight function (the number of storage elements on the arc).*

Let $C(\mathcal{G})$ be the set of cycles of $\mathcal{G}$. For $\mathcal{G}$ to have well-defined physical meaning, the following three constraints must be satisfied:

- $\forall v \in V \; \big(d(v) \geq 0\big)$
- $\forall a \in A \; \big(w(a) \geq 0\big)$
- $\forall c \in C(G), \exists a \in c \; \big(w(a) > 0\big)$

Example The circuit represented by the graphs of Figure 7.1 is an instance of a digital correlator that takes a stream of bits x_i and compares it with a fixed-length pattern $\alpha_1, \ldots, \alpha_J$ in order to produce the output $y_i = \sum_{j=0}^{J} f(x_{i-j}, \alpha_j)$, where $f(x,y)$ is the comparison function returning one if $x = y$ and zero otherwise [149, 150]. Circuit graph $\mathcal{G}_1$ is a direct implementation of this specification for $J = 3$. Its components are instances of two kinds of combinational elements: 4 comparators (v_1, v_2, v_3, v_4) and 3 adders (v_5, v_6, v_7), while vertex v_0 represents the host system. This implementation has 4 storage elements (e.g., edge-triggered flip-flops controlled by a common clock) represented by the dark rectangles. For the delay assignment of the combinational elements, I initially take the same figures as in [150]: 7 time units for the adders and 3 time units for the comparators. ■

Definition 7.2 *A* path *p of a circuit graph $\mathcal{G}$ is an alternating sequence of vertices and arcs $v_1, a_1, \ldots, a_{k-1}, v_k$. The* length *of path p is $|p| = k$. The* path delay *$d(p)$ and the* path weight *$w(p)$ of p are respectively:*

$$\begin{aligned} d(p) &= \sum_{i=1}^{k} d(v_i) \\ w(p) &= \sum_{i=1}^{k-1} w(a_i) \end{aligned}$$

Example Graph $\mathcal{G}_1$ in Figure 7.1 has a path p from v_1 to v_0 passing through v_4 with $d(p) = 33$ and $w(p) = 3$. ■.

Definition 7.3 *A* combinational path *p is a path that does not contain any storage element, i.e. such that $w(p) = 0$.*

The longest combinational path (*critical path*) in a synchronous circuit is a lower-bound on the minimum value of the clock period $\psi(\mathcal{G})$ at which the circuit can operate. This lower-bound is strict under the assumption that the clock overhead can be neglected, otherwise it is necessary to include the value of the expression $\psi_{overhead} = \psi_{skew} + \psi_{jitter} + \psi_{latch}$ containing the estimations of clock skew, clock jitter, and latch set-up time [132].

Example The critical path p_{crit} of $\mathcal{G}_1$ is the path from v_4 to v_7 with $d(p_{crit}) = 24$. ■

Definition 7.4 *A* retiming *of a circuit graph $\mathcal{G} = (V, A, d, w)$ is an integer-valued vertex-labelling $r : V \rightarrow \mathbb{Z}$ specifying a transformation that returns a new graph $\mathcal{G}_r = (V, A, d, w_r)$ such that:*

$$\forall a = (v_i, v_j) \in A \left(w_r(a) = w(a) + r(v_j) - r(v_i) \right)$$

The proof of correctness of retiming can be found in [148].

Example Graph $\mathcal{G}_2$ in Figure 7.1 represents the result of applying retiming for optimal clock period on $\mathcal{G}_1$. By comparing the two graphs, one can verify that $\mathcal{G}_2$ is obtained by repositioning two flip-flops and adding one new flip-flop according to the vertex-labelling r such that $r(v_0) = 0, r(v_1) = -1, r(v_2) = -1, r(v_3) = -2, r(v_4) = -2, r(v_5) = -2, r(v_6) = -1, r(v_7) = 0$. Retiming reduces the clock period to $\psi(\mathcal{G}_2) = 13$ units. Retiming enables interesting trade-offs between area (the circuits differ by one flip-flop) and performance ($\mathcal{G}_2$ gives a 45% speed-up). ■

Circuit retiming for clock period minimization can be casted as a linear programming problem, which can be solved efficiently in $\bigcirc(V^3)$ steps with $\bigcirc(V \cdot A)$-time Bellman-Ford shortest-path algorithm [62, 140]. An asymptotically faster algorithm running in $\bigcirc(V \cdot A)$ time is given in [150].

7.1.2 Using Retiming to Pipeline Combinational Circuits

In [149, 150], Leiserson and Saxe explain how their proposed general framework for retiming can be applied to *optimally pipelining combinational circuits* that do not present feedback paths, i.e. such that their circuit graph representation $\mathcal{G}$ is a directed acyclic graph (DAG). The problem, called *optimum pipelining problem* by DeMicheli [168], can be formulated as follows: *for a given combinational circuit, insert properly storage elements so that the circuit clock period is minimized for a given latency* λ. An instance of this problem can be casted into an instance of the retiming problem for minimum clock period by executing the following procedure:

1. add one input interface vertex v_i to $\mathcal{G}$ and as many arcs as they are necessary from v_i to the primary inputs of $\mathcal{G}$;

2. add one output interface vertex v_o to $\mathcal{G}$ and as many arcs as they are necessary from the primary outputs of $\mathcal{G}$ to v_o;

3. connect v_o to v_i by adding an arc $a_p = (v_o, v_i)$ with weight $w(a_p) = \lambda$;

4. retime the new graph while forcing $w(a_p) = 0$.

The resulting retimed graph represents a pipelined implementation of the original combinational circuit that can operate with a faster clock at the cost of introducing an I/O latency of λ clock periods. The latency/clock-period trade-off curve for the given circuit can be derived by retiming $\mathcal{G}$ for different values of λ.

One could wonder whether or not retiming may be used also to pipeline a generic sequential circuit. For generic sequential circuit I intend a circuit that not only contains storage elements, but some of these elements may sit on feedback paths. For instance, this is the case of the state registers that are necessary to derive a circuit implementation for any finite state machine (FSM) [124, 240] as illustrated in Figure 1.1. The answer to this question is negative due the existence of the retiming invariant rule, which is discussed in the following section.

7.1.3 The Invariant Rule of Retiming

The following result confirms the intuition that the circuit graph cycles (i.e. the feedback paths in the circuit) are a limiting factor of retiming.

Theorem 7.1 (Retiming Invariant Rule). *If $\mathcal{G} = (V, A, d, w)$ is a circuit graph and $r : V \rightarrow \mathbb{Z}$ a retiming on the vertices of $\mathcal{G}$ then:*

$$\forall c \in \mathcal{G} \left(w_r(c) = w(c)\right)$$

Proof. See [149, 150]. □

Therefore, all the possible retimed versions of a circuit modeled by $\mathcal{G}$ must satisfy the above *retiming invariant rule*, which can be stated as follows: *the number of storage elements that lie on any feedback path of a synchronous circuit must remain constant through retiming.*

In [212], Shenoy explains the importance of checking that the invariant rule is preserved when applying retiming in practice and discusses how it is sufficient to check this property on one of the fundamental cycle sets of the graph. These cycles can be efficiently derived by computing a spanning tree of the graph and using the non-tree edges to define the set.

The delay-to-register cycle ratio of a cycle in the circuit graph is defined as the ratio between the sum of combinational delays on the cycle and the sum of storage elements on the cycle.

Definition 7.5 *The* delay-to-register cycle ratio $\rho(c)$ *of a cycle c in a circuit graph $\mathcal{G}$ is defined as*

$$\rho(c) = \frac{\sum\limits_{v \in c} d(v)}{\sum\limits_{a \in c} w(a)}$$

The maximum value (over all cycles) of the delay-to-register cycle ratio limits the speed optimization that can be obtained using retiming.

Theorem 7.2 *If $\mathcal{G} = (V, A, d, w)$ is a circuit graph and $\psi_r(\mathcal{G})$ is the minimum clock period achievable under any retiming transformation $r : V \to \mathbb{Z}$ then*

$$\psi_{lb}(\mathcal{G}) < \psi_r(\mathcal{G}) < \psi_{ub}(\mathcal{G})$$

where

$$\begin{aligned}
\psi_{lb}(\mathcal{G}) &= \left\lceil \max_{c \in C(\mathcal{G})} \rho(c) \right\rceil = \left\lceil \max_{c \in C(\mathcal{G})} \left(\frac{\sum_{v \in c} d(v)}{\sum_{a \in c} w(a)} \right) \right\rceil \\
\psi_{ub}(\mathcal{G}) &= \left(\psi_{lb}(\mathcal{G}) + d_{max} - 1 \right)
\end{aligned}$$

and $d_{max} = \max_{v \in \mathcal{G}} \left\{ d(v) \right\}$.

Proof. Proven by Papaefthymiou [185]. □

Definition 7.6 *A cycle $c \in C(\mathcal{G})$ such that* $\rho(c) = \psi_{lb}(\mathcal{G})$ *is a* critical cycle.

Hence, the previous result sets a lower-bound on the minimum clock period achievable via retiming and this lower-bound is a function of maximum delay-to-register cycle ratio. The lower-bound is theoretical because it may not be necessarily reached by a retiming transformation.

Example Table 7.1 reports the four cycles of the graph circuits of Figure 7.1 together with the corresponding values for the sum of the vertex delays, arc weights and delay-to-register cycle ratios. From Theorem 7.2 it follows that the retiming theoretical lower-bound for the correlator circuit is $\psi_{lb}(\mathcal{G}) = \lceil \rho(c_4) \rceil = 10$. However, practically, the minimum clock period achievable via retiming for the correlator circuit is just $\psi(\mathcal{G}) = 13$. ■

c_i	cycle vertices	$d(c)$	$w(c)$	$\rho(c_i)$
c_1	$\{v_0, v_1, v_7\}$	10	1	10
c_2	$\{v_0, v_1, v_2, v_6, v_7\}$	20	2	10
c_3	$\{v_0, v_1, v_2, v_3, v_5, v_6, v_7\}$	30	3	10
c_4	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	33	4	8.25

Table 7.1: Cycles, delay, weights and delay-to-register cycle ratios for the circuit graph of Figure 7.1.

7.2 Applying Recycling at the Gate Level

As discussed in Chapter 6, recycling is an optimization technique for latency-insensitive systems that essentialy combines three operations: the insertion of relay stations, the movement of relay stations across shell-core pairs and the re-encapsulation of the shells around the cores. In particular, the second operation is clearly reminiscent of the movement of registers across combinational gates that is performed by retiming. There are, however, important differences between recycling and retiming that I discuss in the next section. Then, I study the combined application of recycling and retiming explaining qualitatively when it can deliver a speed-up of the circuit. Finally, I discuss analytically how to estimate such speed-up.

7.2.1 Differences between Recycling and Retiming

When applied in the context of SOC design, recycling addresses essentially the optimization of global wire pipelining. Long and slow communication wires on the chip must be segmented in shorter and faster stages in order to drive them at the same clock frequency as the rest of the system. This clock frequency is dictated by the slowest core in the system. Latency-insensitive design enables automatic wire pipelining via relay station insertion and recycling combines this operation with the encapsulation of the system modules in an attempt to minimize its impact on the system throughput. More generally, recycling can be seen as a technique to re-organize the computational structure of a circuit while exploring efficiently the design solution space and the trade-offs between computation delay and communication latency.

In theory, nothing prevents the application of recycling at the gate-level as long as an efficient shell encapsulation of the gates can be done without excessive area overhead. At this level, the goal would not be to pipeline a long global wire, but to "break" the logic long combinational paths beyond what can be done by moving around simple registers as retiming does. In fact, the two techniques can be combined as long as the following differences are kept in mind:

- both retiming and recycling "push around" storage elements: retiming moves registers across combinational gates, recycling moves relay stations across shells that encapsulate stallable sequential circuits; any "retiming move", however, must satisfy the mathematical equation of Definition 7.4; recycling, instead, can insert and remove relay station from any channels in arbitrary number and without respecting any constraint;

- as long as they are stallable, the core modules can be *arbitrary sequential circuits* and, therefore, contain various storage elements, and implement complex finite state machines (FSM); this is an important difference with respect to retiming, which operates on storage elements across *combinational circuits* (in other words, retiming can be applied only to a graph $\mathcal{G}$ whose vertices are combinational circuits);

- the theory of latency-insensitive protocols guarantees that after performing shell encapsulation on a system S, the resulting latency-insensitive system S' is functionally equivalent to the original one modulo the presence of stalling (τ) symbols; this means that if S' is solicited with the same input traces as S, it produces an output trace that presents exactly the same ordered sequence of data as S, but where two consecutive valid data may be interleaved by one or more stalling events; instead, two retimed versions of the same system S produce *exactly* the same output traces without the interleaving of any spurious symbol; naturally, the stalling events can be easily filtered out at the output ports of a latency-insensitive system to obtain the same *exact* output trace produced by the corresponding retimed version.

As discussed in Section 4.3.1, the nature of the stalling events goes back to the idea of shells

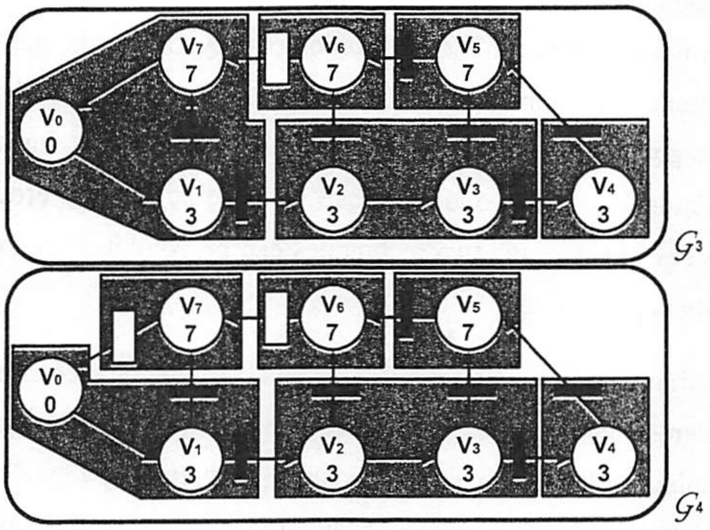

Figure 7.2: Two recycled versions of the correlator circuit of of Figure 7.1 (shading shows the shell wrapping; light rectangles are relay stations initialized to τ).

and relay stations. The simplest way to insert extra storage elements into a synchronous circuit without jeopardizing its functional behavior, is to make sure that they are initialized with values that do not disrupt the state of the down-link sequential processes. Hence, each relay station is initialized with a τ symbol (this can be done via special encoding or by adding an extra wire to a given bus). Then, the interface logic of a shell simply operates according to an AND-causality semantics: if *all* the input channels present valid data, then the shell lets the core receive them, make progress with its computation, and produce valid output data; but, if *even a single* channel presents a τ, then the shell stalls the core and puts new τ symbols on the output channels instead of valid data.

7.2.2 Combining Recycling And Retiming

The idea of combining recycling and retiming is based on the following considerations:

- any stand-alone combinational circuit can be made sequential by inserting one storage element on each of its outputs;

- the resulting sequential circuit is a stallable circuit and it can be made patient by synthesizing a shell that encapsulates it and controls its activity trough a *clock-gating* mechanism (as discussed in Section 4.1.2).

Hence, any retimed circuit, and not necessarily an *optimally retimed circuit*, can be transformed into a latency-insensitive system. Then, its critical paths can be pipelined via relay station insertion to further reduce the minimum clock period at which it can operate. In other words, the theory of latency-insensitive protocols together with recycling can be used to *break* the retiming invariant rule expressed by Theorem 7.1. Naturally, this transformation would generally produce a circuit with a larger area occupation. Therefore, shell encapsulation should be applied carefully to minimize such overhead.

Example The circuit graph $\mathcal{G}_3$ pictured at the top of Figure 7.2 is a recycled version of the correlator that can be derived with the following procedure:

1. apply a new retimed transformation to the optimally retimed graph circuit graph $\mathcal{G}_2$ of Figure 7.2 by using a vertex-labeling r' such that $r'(v_5) = 1$ and $r'(v_6) = 1$ while $r'(v_j) = 0$ for $j \neq 5,6$ (notice that the critical path of the resulting circuit is the path connecting v_6 to v_1 for a delay of 17 time units);

2. partition the circuit in sub-circuits whose outputs are delimited by storage elements and create a shell around each sub-circuit;

3. insert a relay station between vertices v_6 and v_7 to break the critical path [1].

Knowing that relay stations are storage elements, it is easy to verify that the critical path of the resulting graph $\mathcal{G}_3$ is the path connecting v_7 to v_1 for a length of 10 time units. Hence, in first approximation, $\mathcal{G}_3$ can nominally run with a clock period $\psi(\mathcal{G}_3) = 10$, a 23% speed-up with respect to the optimally retimed circuit $\mathcal{G}_2$. This idea can be pushed to its limits by considering circuit graph $\mathcal{G}_4$ at the bottom of Figure 7.2. Here, the insertion of a new relay station between v_7 and v_8 further breaks the critical path in order to reach the minimum possible length. This coincides with the delay of the slowest circuit component, in this

[1] In fact, often it not necessary to insert a relay station. Here it is sufficient to insert a storage element initialized with a τ symbol on each output of the combinational circuit associated to vertex v_6 and, then, encapsulate v_6 within a shell.

case the adder, which is 7 time units. Hence, $\mathcal{G}_4$ can run with a nominal clock period as small as $\psi(\mathcal{G}_4) = 7$, a 46% speed-up with respect to $\mathcal{G}_2$. ■

Besides having such considerable nominal speed-ups with respect to the optimally retimed circuits, it would appear that the ability of inserting extra storage elements, thus overcoming the retiming invariant rule, makes it possible to beat even the theoretical lower-bound expressed by the maximum delay-to-register cycle ratio of Theorem 7.2. But, before making this claim, more considerations need to be made. First, it is not realistic to assume that the insertion of the shell logic does not cause some timing overhead. To take this into account, one should include an additional term ψ_{shell} to the other components of the clock overhead $\psi_{overhead}$. More importantly, even if the impact of ψ_{shell} is negligible with respect to the critical path delay, it is the presence of the τ symbols that affects negatively the amount of speed-up achievable with recycling. As discussed in Chapter 5, after initializing each relay station with a τ symbol, the AND-causality semantics of the shells implies that the number of τ symbols in a cycle of the circuit remains constant during its operations. Hence, τ symbols are periodically detected at the circuit outputs and, since they cannot be considered as valid results, they must be discounted from the assessment of the circuit performance.

In Section 4.3 the throughput $\vartheta(S)$ of a latency-insensitive system S was defined as the amount of true packets produced by S in a given time interval. This definition naturally extends to the case of a synchronous circuit $\mathcal{G}$. Since the recycled circuit is a latency-insensitive system, all the considerations made in Chapter 5 still apply. In particular, $\vartheta(\mathcal{G})$ can be computed statically as the minimum value (over all cycles) of the ratio between the sum of storage elements (that are not relay stations) and the sum of all storage elements (including also the relay stations). This justifies the following definition.

Definition 7.7 *The* throughput $\vartheta(\mathcal{G})$ *of the circuit graph* $\mathcal{G}$ *is defined as*

$$\vartheta(\mathcal{G}) = \min_{c \in C(\mathcal{G})} \left(\frac{\sum_{a \in c} w(a)}{\sum_{a \in c} w(a) + z(a)} \right)$$

where $z(a)$ *is the number of relay stations on arc* a.

Throughput $\vartheta(\mathcal{G})$ is a rational number between zero and one.

Then, based on the analogous definition given in Section 4.3.1, the *effective clock period* $\psi_{eff}(\mathcal{G})$ of a recycled circuit can be computed as the ratio of the nominal clock period over the circuit throughput:

$$\psi_{eff}(\mathcal{G}) = \frac{\psi(\mathcal{G})}{\vartheta(\mathcal{G})}$$

The effective clock period $\psi_{eff}(\mathcal{G})$ is clearly the correct performance metric for circuit $\mathcal{G}$ and should be used instead of the misleading nominal clock period. Consequently, it is possible to verify that an indiscriminate application of recycling does not necessarily pay off from a performance viewpoint.

Example Considering again the circuit graphs of Figure 7.2: circuit $\mathcal{G}_3$ has throughput $\vartheta(\mathcal{G}_3) = 2/3$, and, therefore, effective clock period $\psi_{eff}(\mathcal{G}_3) = 10 \cdot (3/2) = 15$, while circuit $\psi(\mathcal{G}_4)$ has throughput $\vartheta(\mathcal{G}_4) = 1/2$, and effective clock period $\psi_{eff}(\mathcal{G}_4) = 7 \cdot (2/1) = 14$. Therefore, the effective performance of both the recycled circuits is beaten by the optimally retimed circuit $\psi(\mathcal{G}_2)$, which has clock period $\psi(\mathcal{G}_2) = 13$. ■

In summary, it is true that recycling can be applied to any sequential circuit in order to overcome the *retiming invariant rule*, but it is also true that there exists something like a *recycling invariant rule* that may cancel all the gain! Does this mean that recycling can never return a circuit implementation whose performance is better than the optimally retimed version? The following counter-example proves that this is not necessarily the case.

Example Assume that all the vertices in the two circuit graphs of Figure 7.2 have the same delay ∂ but for $d(v_0) = 0$. Then the recycled circuit graph $\mathcal{G}_4$ has an effective clock period $\psi_{eff}(\mathcal{G}_4) = (2/1) \cdot \partial = 2 \cdot \partial$, which is strictly smaller than the clock period of the optimally retimed graph for every possible value of ∂. In fact, since retiming cannot change the number of flip-flops on a cycle, each cycle c of any retimed correlator will present a sequence $\sigma_m(c)$ of m consecutive vertices whose connecting arcs have zero weight (i.e. the arcs do not present a storage element) where $m = \lceil \frac{|c|}{w(c)} \rceil$. In particular, while cycles $c_1 = \{v_0, v_1, v_7\}$ and $c_4 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ present sequences $\sigma_m(c_i)$ of, at most, length 2, both cycles $c_2 = \{v_0, v_1, v_2, v_6, v_7\}$ and $c_3 = \{v_0, v_1, v_2, v_3, v_5, v_6, v_7\}$ present a sequence $\sigma_m(c_i)$ of length 3. Hence, under this delay assignment, any optimally retimed circuit $\mathcal{G}$ will have a critical path of 3 vertices for a minimum clock period $\psi(\mathcal{G}) = 3 \cdot \partial$, which is 33% worse than the effective clock period $\psi_{eff}(\mathcal{G}_4)$ of the corresponding optimally recycled circuit. This 33% performance gain obtained through recycling remains constant if the length of the correlator is extended by considering any implementation with $J > 3$. In fact, any extension would create new cycles presenting a critical path of 3 vertices, while the effective clock period of the recycled circuit would continue to depend only on ∂. ■

In summary, there are cases when recycling does not produce any gain with respect to an optimally retimed circuit and cases where a relevant gain is guaranteed. This depends on the circuit topology and its delay assignment. Next I give a general model to analyze when combining recycling and retiming does pay off.

7.2.3 Benefit Analysis on the Combination of Recycling and Retiming

From the correlator circuit example it is clear that an analysis of the structure of the cycles of a circuit is the key to gauging the possible impact of combining retiming and recycling. After all, the results of Theorems 7.1 and 7.2 as well as Definition 7.7 are all related to circuit cycles. The two circuits illustrated in Figure 7.3 are important special cases to understand how to model analytically the combined application of recycling and retiming. Both circuits present a single cycle on which N vertices sit. The top circuit, $\mathcal{G}_5$, has $N-1$ registers while in the bottom circuit $\mathcal{G}_6$ there is only one register. These represent two extreme cases for this circuit structure, which, generally, can have K registers with $K \in [1, N-1]$. Notice that the case $K = 0$ is ruled out by assuming the absence of combinational loops. Also case $K \geq N$ is not interesting because the application of either retiming or recycling would not produce any benefit from the timing optimization viewpoint (the circuit would run with a clock period equal to the maximum combinational delay among all its components anyway).

For $K \in [1, N-1]$ the theoretical lower bound on the minimum clock period achievable via retiming is given by

$$\psi_{lb}(\mathcal{G}) = \left\lceil \frac{\sum_{i=1}^{N} d(v_i)}{K} \right\rceil$$

In Figure 7.3, this gives $\psi_{lb}(\mathcal{G}_5) = \lceil \frac{\sum_{i=1}^{N} d(v_i)}{N-1} \rceil$ and $\psi_{lb}(\mathcal{G}_6) = \sum_{i=1}^{N} d(v_i)$. Also, the optimally retimed graph presents a sequence $\sigma^{\star}_{m}$ of m consecutive vertices without registers in between, where $m = \lceil \frac{N}{K} \rceil$ and the '$\star$' symbol denotes that this sequence is the one with the smallest *delay sum* among the N possible ones. I write this delay sum as $d(\sigma^{\star}_{m})$ (it coincides with the circuit clock period if no single vertex has delay greater than this). Hence, $\mathcal{G}_5$ has $\psi_{ret}(\mathcal{G}_5) = \max\left\{d_{max}, (d(v_x) + d(v_y))\right\}$, where v_x, v_y are the two consecutive vertices with the smallest compound delay, Instead, $\mathcal{G}_6$ has $\psi_{ret}(\mathcal{G}_6) = \sum_{i=1}^{N} d(v_i)$. Denote with H-recycling the insertion of H relay stations on the circuit. The value of H depends on both N and K, as well as the vertex delay assignment, but, in any case, it is useless to have $H > (N-K)$. The nominal clock period of a H-recycled circuit is

$$\psi_H(\mathcal{G}) = \max\left\{d_{max}, d(\sigma^{\star}_{m'})\right\}$$

with $m' = \lceil \frac{N}{K+H} \rceil$. For instance, $\psi_1(\mathcal{G}_5) = d_{max}$. In general, the nominal clock period $\psi_H(\mathcal{G})$ is smaller than the corresponding $\psi_{ret}(\mathcal{G})$. To find the effective clock period, however, $\psi_H(\mathcal{G})$ must be divided by the throughput $\vartheta = \frac{K}{K+H}$ of the recycled circuit. Putting it all together leads to the conclusion that recycling returns a gain over optimal retiming if and only if

$$\max\left\{d_{max}, d(\sigma^{\star}_{m'})\right\} \cdot \frac{K+H}{K} < d(\sigma^{\star}_{m}) \tag{7.1}$$

The gain percentage is given by the following:

$$gain(\%) = \left(1 - \frac{\max\{d_{max}, d(\sigma^{\star}_{m'})\} \cdot \frac{K+H}{K}}{d(\sigma^{\star}_{m})}\right) \cdot 100 \tag{7.2}$$

To take into account also the clock overhead due to the shell logic it is sufficient to add the term $\psi_{shell} \cdot \frac{K+H}{K}$ to the left-hand side of the inequality.

For the special case of delay-homogeneous circuits (where $\forall i, d(v_i) = \partial$) the previous condition can be simplified as follows:

$$\left\lceil \frac{N}{K+H} \right\rceil \cdot \partial \cdot \frac{K+H}{K} < \left\lceil \frac{N}{K} \right\rceil \cdot \partial \;\Leftrightarrow\; \left\lceil \frac{N}{K+H} \right\rceil \cdot \frac{K+H}{K} < \left\lceil \frac{N}{K} \right\rceil \tag{7.3}$$

Therefore, recycling never pays off for circuit $\mathcal{G}_6$ while, as long as $N > 2$, it always does for circuit $\mathcal{G}_5$. In the latter case the gain percentage is given by $\frac{N-2}{2 \cdot N-2} \cdot 100$, which varies from 25% for $(N = 3)$ up to a maximum of 50%. Also, notice that for circuit $\mathcal{G}_5$ recycling

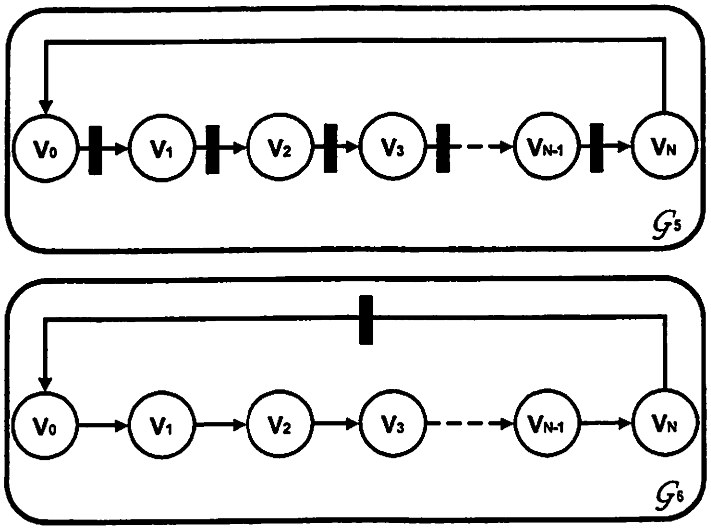

Figure 7.3: Case studies for recycling benefit analysis: N-1 FFs (top), 1 FF (bottom).

technically beats the theoretical lower bound of Theorem 7.2 as it allows to match exactly the ratio $\frac{N}{N-1} \cdot \partial$, instead of its integer ceiling.

The model presented in this section can be used to efficiently analyze more complex circuits containing many intersecting cycles because Equation (7.1) still applies for each potential critical cycle.

7.2.4 Recycling-Based Design Exploration

In this section I discuss an experimental case study on the application of recycling to the exploration of alternative gate-level implementations of a synchronous circuit.

Figure 7.4 shows a relatively simple synchronous pipelined data-path circuit that consists of four modules: two adders, one multiplier, and one divisor. The output of each module is stored in a storage element, an edge-triggered flip-flop. All flip-flops are controlled by the same clock. The data-path presents three feedback paths from the output of the second adder to the first adder, the multiplier, and the second adder itself, respectively.

I performed a design exploration for this circuit using VERILOG HDL, a commercial

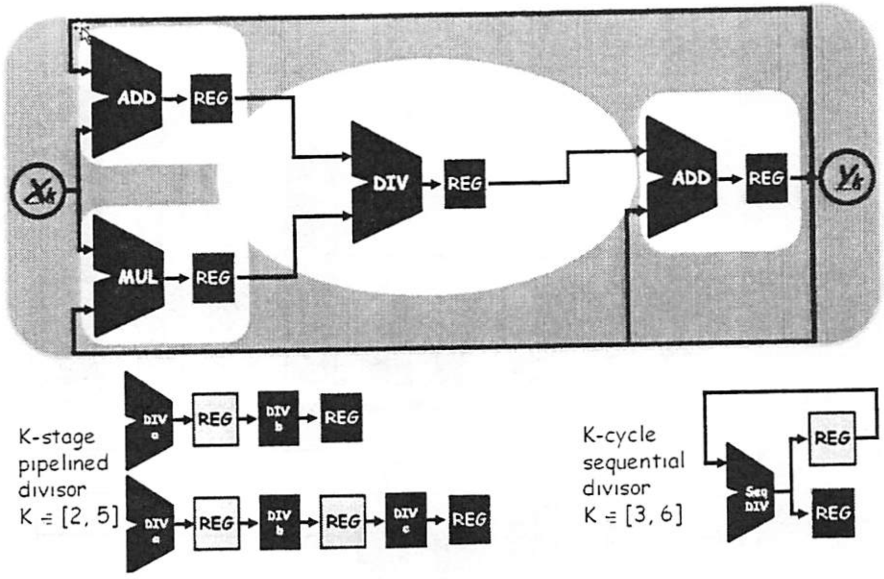
ADD
REG
MUL
REG
DIV
REG
ADD
REG
X_k
Y_k
K-stage pipelined divisor K = [2, 5]
DIV a
REG
DIV b
REG
DIV c
K-cycle sequential divisor K = [3, 6]
Seq DIV
REG

Figure 7.4: Recycling-based design exploration of a pipelined data-path.

logic synthesis tool, a commercial library of pre-designed IP cores, and two commercial technology processes (a 130*nm* and a 90*nm*) together with the corresponding gate-level standard cell libraries. I wrote the VERILOG code for the data-path circuit at the register transfer level (RTL), instancing the cores from the library. Then, I synthesized it in order to derive a standard-cell netlist with the given technologies.

Each core in the library is a *black-box* core that can be instanced directly in the VERILOG specification. The library offers several alternative designs for the functionality of each module. In particular, the choices for the divisor module are: a simple one stage divisor, four *h*-stage pipelined divisors with the number of stages *h* varying between 2 and 5, and four sequential divisors that complete the operation in *k* cycles with *k* varying between 3 and 6. The bottom of Figure 7.4 reports diagrams for the sequential divisor, a 2-stage divisor and a 3-stage divisor.

I completed two distinct designs of the data-path circuit: a traditional synchronous design and a latency-insensitive design using shell modules from a library that I had previously designed to implement the communication architecture described in Section 4.2. For both designs I started by assuming to use a simple one-stage implementation for each mod-

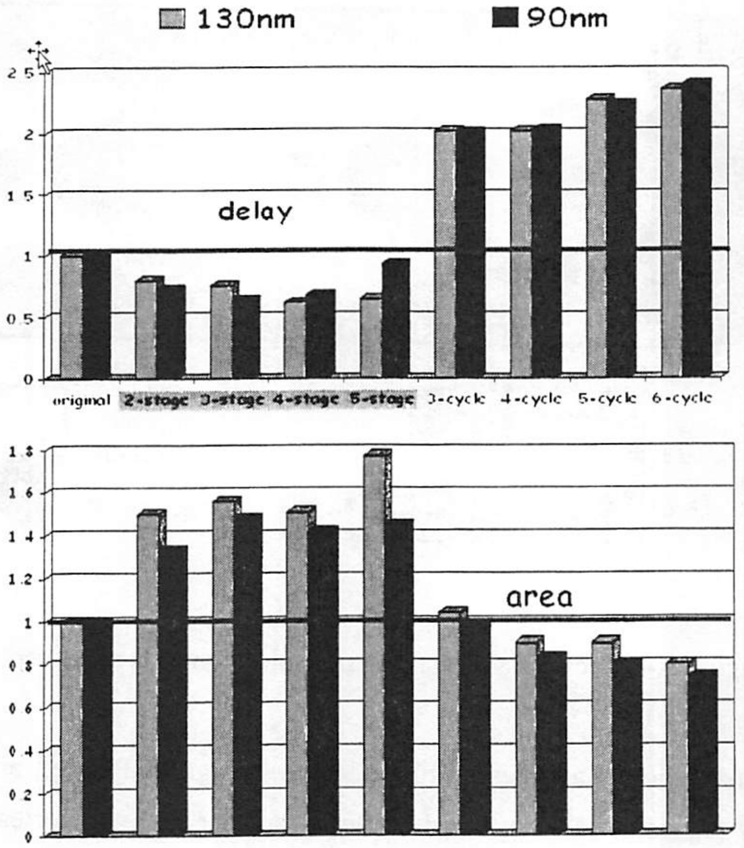

Figure 7.5: Experimental results from recycling the data-path of Figure 7.4.

ule, including the divisor. For the latency-insensitive design I simply encapsulated each module within a distinct shell. Naturally, the resulting circuit presents a bigger area occupation without providing any speed-up (the IP cores are the same). Since I did not add any relay station, the latency-insensitive design runs at the same throughput as the traditional design and produces the same output trace. I verified this by simulating the two designs with a commercial VERILOG simulator.

Then, I started the design exploration targeting the divisor module that is clearly the bottleneck for such circuit as it contains by far the longest critical path. I tried the eight different implementations of the divisor mentioned above. In each case I simply had to write few lines of VERILOG code to interface the core to the shell that encapsulates it.

These lines inform the shell when the core has completed its computation. For instance, in the case of a three-stage divisor, the shell must know that once it has provided input data to the core it has to wait three clock cycles to get the corresponding output data.

I did not have to change the internal logic of the shell, which, in fact, is an instance of the same shell for each module in the data-path since they all have two inputs and one output. Also, after modifying the divisor implementation I did not have to touch any other part of my design. By simulating the VERILOG code I validated that all latency-insensitive designs are latency equivalent to the original one.

Although no relay stations are inserted between the shells, each of the eight alternative latency-insensitive designs runs with a throughput smaller than the original design. The reason is the combination of the feedback paths in the design and the additional latency that both pipelined divisors and sequential divisors present at the first timestamp. For instance, while the flip-flop sampling the output of the two-stage pipelined divisor can be initialized with the same value as in the original design, the value of the internal stage flip-flop must be considered a void value, i.e. a τ symbol.

The bar charts of Figure 7.5 illustrate the experimental results after synthesizing the traditional design (*original*) and the eight alternative latency-insensitive designs. Specifically, the top chart reports the value of the effective clock period (i.e. accounting also for the throughput degradation) as normalized with respect to the nominal clock period of the original design. Similarly, the bottom chart reports the value of the area occupation normalized with respect to the area occupation of the original design. It is evident that the latency-insensitive designs featuring the pipelined divisors offer a better performance in exchange of a larger area occupation. Those with the sequential divisors offer better area but less speed. Specifically, the design with the four-stage divisor appears as the best choice among the pipeline divisors for a performance-oriented design with the 130*nm* technology: it is 38% faster while being 51% larger. If the goal is to minimize the area occupation, the 6-cycle sequential divisor is 18% smaller but more than twice slower than the original design. It is interesting to observe that the area/delay trade-off changes across the two technology nodes. For a performance-oriented design with the 90*nm* technology the best opportunity is represented by the 3-stage divisor which offers a 36% faster design with respect to the original design (also synthesized at 90*nm*) in exchange for being 48% larger.

7.3 Related Work

In this section I discussed the relationship between recycling and three other techniques that have been proposed to overcome the retiming invariant rule.

7.3.1 Recycling versus Slowdown

Slowdown, or *c-slow retiming*, was proposed together with retiming in [150], where the main goal is to establish formal techniques to transform a synchronous circuit into a functionally equivalent systolic circuit, i.e. a circuit that presents at least one register on every arc of its circuit graph. More recently slowdown was separately applied to FPGA by Snider [217] and Weaver *et al.* [243]. Slowdown is performed in two steps: (1) replace each register in the original circuit by a sequence of c registers, thus producing the c-slow equivalent circuit; (2) retime the c-slow circuit to minimize its clock period. While slowdown is obviously defined for c being an integer greater than one, it should be noticed that for any given circuit $\mathcal{G}$, there is an integer $k_{\mathcal{G}}$ such that c-slow retiming produces a clock period reduction if and only if $c \geq k_{\mathcal{G}}$. Furthermore, slowdown does not come for free, but, besides the extra area due to additional registers, it implies a performance overhead, somewhat similar to recycling. In particular, as suggested by its very name, a c-slow circuit, while running with a potentially higher frequency clocks, processes a single input data at the throughput of $\frac{1}{c}$. In fact, at any given clock iteration, only $\frac{1}{c}$ of the registers in the c-slow circuit contains valid data. Hence, if ψ^c is the clock period achievable via slowdown, the effective clock period is $\psi^c_{eff} = \frac{\psi^c}{c}$. It should not be a surprise, then, that slowdown was proposed with the idea of contemporaneously processing c input data streams by properly multiplexing and de-multiplexing the I/O ports of the c-slow circuit. By doing so, the processing throughput can be raised back (up to one in the optimal case, where each computation thread operates with period ψ^c_{eff}).

Besides the obvious similarities, recycling and slowdown are different and, in particular, the former subsumes the latter when one considers only the processing of single input data stream. In this case, recycling can always mimic slowdown, because the same results obtained with a c-slow retiming can be obtained via recycling by inserting relay stations (up to $c-1$ per cycle) instead of duplicating registers. In both cases, the resulting circuit throughput is given by $\frac{1}{c}$. On the other side, it is not difficult to find an example of a case where a slowdown cannot achieve the same performance that recycling offers due to the coarser granularity of the slowdown transformation. For instance, consider again circuit $\mathcal{G}_5$ of Figure 7.3 representing a feedback loop with N vertices v_i, each with a delay $d(v_i) = \partial$, and $N-1$ registers. The insertion of one single relay station produces a recycled circuit with a nominal clock period equal to ∂ and a circuit throughput equal to $\frac{N-1}{N}$ for an effective clock period $\psi_{eff} = \frac{N}{N-1} \cdot \partial$. Instead, the simplest slowdown transformation produces a 2-slow circuit that has the same clock period ∂ as the recycled circuit, but a smaller throughput equal to $\frac{1}{2}$ (which effectively cancels the benefits of the transformation because $2 \cdot \partial$ is the clock period of the original circuit). Therefore, as long as $N > 2$, the performance of the recycled circuit is better than the performance of the corresponding 2-slow circuit by a percentage equal to $\frac{N-2}{2 \cdot N-2}$, that, in the smallest case of $N = 3$, corresponds to a gain of 25%.

7.3.2 Recycling versus Timing Optimization via Software-Pipelining

A different attempt to overcome the retiming invariant rule of Theorem 7.1 is made in [22, 23], where Boyer *et. al* propose *software pipelining* techniques as a better alternative to retiming for sequential circuit optimization. The relationship between retiming and software pipelining has been studied in several works. In general it is retiming that is applied to further enhance software pipelining [33, 47]. Instead, Boyer *et. al* take the reverse approach: their goal is to derive an optimum placement of the registers such that the clock period is close (or, in the best case, equal) to the lower-bound of Theorem 7.2. Then, from the optimal schedule found with software pipelining, new registers are placed in the circuit regardless of the number and the position of the original ones. The resulting circuit is a multi-phase clocked circuit, where all clocks have the same period and the phases are automatically determined by the algorithm. Finally, edge-triggered flip-flops are used where the combinational delays exactly match that period, whereas level-sensitive latches are used elsewhere, thereby improving the circuit area. In the case of the correlator of Figure 7.1, the authors can claim to reach the theoretical lower bound of the clock period equal to 10 time units, thereby beating both the best optimally retimed circuit and the optimally recycled one.

To compare this approach with recycling, the following considerations must be done. First, the two approaches are similar in the sense that they both reduce the circuit critical path by inserting new storage elements, while making sure that all storage elements are not *sampled* at every clock iteration. This reduction is achieved *physically* by Boyer's approach as different latches are controlled with clock signals with different phases, while recycling does it *logically* by letting all flip-flops be sampled at every clock iteration in order to discard then those sampled data coinciding with a τ symbol. Secondly, Boyer's approach gives a faster circuit with simpler components (edge-triggered FFs and level-sensitive latches instead of shells and relay stations) at the price of the higher complexity of having to deal with a multi-phase clocked circuit. Finally, while the register initialization issue is not discussed in [22, 23], it is clear that, to guarantee functional equivalence, the multi-phase circuit must operate according to a pre-defined schedule which makes sure that no spurious data are ever sampled by/from a latch. Conversely, as it is also the case with respect to slowdown, the attractiveness of the recycling alternative is that the underlying latency-insensitive protocol implicitly and automatically encodes the scheduling logic.

7.3.3 Recycling versus Architectural Retiming

Hassoun and Ebeling have proposed architectural retiming as a technique that aims to decrease the clock period of a circuit by increasing the number of registers on a latency-constrained path without increasing its latency, i.e. the number of clock periods to perform the computation [104, 105]. The name architectural retiming captures the combined effect of this technique: it reschedules operations in time and modifies the structure of the circuit to preserve its functionality. This is done using the concept of a *negative register*, which can be implemented using either pre-computation or prediction. Pre-computation of a signal x can be done only if there is sufficient information in the circuit to pre-compute the value of x one cycle ahead of time. Therefore, it is possible only when x does not depend causally (and in a combinational sense) on any primary input signal. When pre-computation is not possible, the alternative is to implement an oracle that predicts the value produced by the negative register. Since a prediction can be wrong, it is necessary to implement also the logic for checking its correctness and nullifying its effects. This ultimately may require to change the interface of the circuit with its environment by making it "elastic" through the addition of a protocol that expresses void output data and back-pressure. In summary, both pre-computation and prediction require the knowledge of the internal structure of the circuit and ad-hoc design changes. This prerequisite, which substantially limits the applicability of architectural retiming, is a major difference with respect to the simple requirements that are necessary to apply recycling.

7.4 Concluding Remarks

I applied the ideas of recycling and latency-insensitive protocols, which I originally developed for system-level design, to the timing optimization of synchronous circuits at the gate-level. I showed how recycling can be combined with retiming to get circuit speed-ups that are not achievable by using stand-alone retiming or *c*-slow retiming. Recycling goes beyond what retiming offers because:

- it can be applied to any network of sequential circuits, i.e. a circuit whose computational elements are sequential circuits, while retiming can deal only with a single sequential circuit whose computational elements are combinational circuits;

- it can be applied to retimed circuits to obtain further speed-up; this is achieved by extra-pipelining the circuit through relay-station insertion, thereby circumventing the main limiting factor in retiming, namely the presence of feedback loops.

I discussed how to transform a synchronous circuit into a latency-insensitive design and how to apply recycling to break the critical paths and reduce further the clock period. Recycling does not come for free, but implies an overhead in terms of both area and latency. In fact, there are cases when this transformation may not be convenient even in terms of timing optimization. I provided an analytical formulation that, based on the circuit topology and the delay characteristics of its components, detects when recycling is advantageous from a performance viewpoint and determines the size of the reachable speed-up.

Then, I discussed an experimental case study to demonstrate that latency-insensitive design and recycling make it possible to efficiently complete significant design exploration with several alternative pre-designed IP cores also at the gate level.

Finally, I compared the proposed approach to architectural retiming as well as to the application of software pipelining techniques to sequential circuit optimization.

Chapter 8

Conclusions and Future Directions

In which past and future help sustain the illusion of the present.

ON the final page of this dissertation I list the major contributions of my research, comment on the reception of latency-insensitive design and its influence on other research projects, and outline the most promising avenues for future work.

8.1 Contributions

The major contributions of the present dissertation are summarized below:

- the *theory of latency-insensitive protocols*, which provides a robust foundation for combining the benefits of synchronous specification with the efficiency of asynchronous implementation in the design of moderately-distributed systems;
  - the definition of the properties of *latency-equivalence* and *patience* together with the formal proofs of their compositionality;
  - the identification of the property of *stallability* as the minimum requirement in order to automatically transform a synchronous process into a patient process;
  - the concept of *shell encapsulation*, thereby any stallable *core* module can be encapsulated in order to manage its communication and synchronization with the other modules in a moderately-distributed concurrent system;

  - the definition of *relay station*, the minimum-capacity *stateful repeater* used to design latency-insensitive pipelined communication channels with maximum throughput;

- the *correct-by-construction latency-insensitive design methodology for SOC design*, which targets both the timing closure problem (via automatic wire pipelining) and the productivity gap problem (via shell encapsulation of pre-designed IP components);
  - the description of a *latency-insensitive design flow* for SOC that relies on traditional CAD tools;
  - a *reference hardware implementation* for latency-insensitive communication architectures based on back-pressure, including *RTL circuit* specification for *shells* and *relay stations*;
  - the design of *distributed symmetrically pipelined channels*, also based on back-pressure, in order to handle on-chip global communication without resorting to centralized, shared global resources;
  - the application of *clock gating*, originally developed for low-power design, to the fine-grained control of the processing of a synchronous circuit;
- the development of *constructive modeling* techniques, based on *marked graphs*, for the performance analysis and optimization of latency-insensitive systems;
  - the proof that latency-insensitive systems are *live by construction* and that their performance can be computed statically based on where, and in which number, relay stations have been inserted;
  - the theoretical result that for latency-insensitive systems a *physical implementation* based on back-pressure and finite queues with length equal to two offers the same maximum sustainable throughput as a *virtual implementation* with infinite queues;
  - the definition of *recycling*, a new design optimization technique combining *channel pipelining* (via relay station insertion) and *throughput equalization* (via cycle balancing) to optimize the performance of a latency-insensitive system;

- the combination of *retiming* (a sequential module) and *recycling* (a network of sequential modules) in order to optimize the performance of sequential circuitry at the gate-level beyond what can be achieved by retiming;

8.2 Influence of Latency-Insensitive Design

I presented the first concepts of the theory of latency-insensitive protocols at the 11th International Conference on Computer-Aided Verification in July 1999 [36]. In November of the same year I presented the basic idea of the latency-insensitive design methodology for systems-on-chip at the International Conference on Computer-Aided Design [34]. Although the reaction was generally positive, there were a number of comments criticizing the work for one or more of the following reasons: (1) some researchers doubted the effectiveness of latency-insensitive design altogether because they basically perceived it as "yet another asynchronous design technique" and, therefore, inherently impractical; (2) others outright questioned the present and future need for a methodology to formally handle wire pipelining; (3) finally, other researchers were concerned with the design overhead involved to adopt latency-insensitive design in practice.

The essential differences between latency-insensitive and asynchronous design were explained in Chapter 3. Beyond that detailed discussion, here it is sufficient to say that I strongly believe that latency-insensitive design can be a vehicle to introduce the many important benefits of asynchronous circuits within a design flow that is firmly based on the well-established synchronous paradigm.

About the second criticism, I think that it will be completely dismissed in a few years, when the impact of on-chip interconnect latency will be evident also for generic ASIC and SOC design, as it already is for high-performance microprocessor design. Wire pipelining will then necessarily become a pervasive technique in integrated circuit design. Over the past few years I have already been pleased to see the publication of many papers which confirm the importance of the impact of interconnect latency and its consequence on both IC design and CAD tools in general. I discussed the most relevant of these papers in Section 2.2.

The third criticism is for me the most interesting and the debate it has generated is still ongoing. Based on the preliminary experiments discussed in Section 4.2.2, I expect that for most IP cores the design overhead of performing a shell encapsulation will be minor (less than 5% in area occupation). Still, it is true that this issue will only be definitively closed once latency-insensitive design is successfully applied to the design of a real product.

More recently, the importance of the present research work has been recognized in several ways. First, in 2003, the 1999 ICCAD paper was selected for inclusion in "The Best of ICCAD - 20 Years of Excellence in Computer-Aided Design", a collection of 42 out of over 2,200 works presented at ICCAD between 1983 and 2002 [35, 80].

Latency-insensitive design is also mentioned on page 21 in the Design Chapter of the 2003 International Technology Roadmap for Semiconductors (ITRS) [117], where it is said that *"[while] past methodologies rely on simple statistical models (e.g., wire-load models) and limited-loops iteration (e.g., one implementation pass to estimate layout, and a second pass that begins by assuming the layout estimate), current and future methodologies entail (a) restricted circuit and layout styles (e.g., two-level programmable logic fabric implementation, or doubly-shielded signal wires) to improve or even guarantee timing and noise correctness; (b) use of enforceable assumptions in the absence of good predictions (e.g., constant-delay methodology variants); and (c) removing the requirement of predictability in the first place (e.g., latency-insensitive synchronization protocols that guarantee correct behavior no matter how many clock cycles separate components)."*

More significantly, industry researchers have shown some interest in applying the latency-insensitive design methodology discussed in Chapter 4 onto real designs. Maybe it should not be surprising that, although this methodology was originally developed targeting SOC design, it was a high-performance microprocessor company that first investigated its possible applications. As of today, the jury is still out on the results of this investigation.

Finally, researchers at various institutions have applied concepts from the present work in some of their projects. I list here the most successful cases.

Latency-insensitive design has been adopted as the mode of operation for the components of "$\times$*pipes*", a scalable and high-performance network-on-chip (NOC) architecture for both homogeneous and heterogeneous multi-processor SOCs that has been co-developed by researchers at Stanford University and the University of Bologna [67, 119].

Singh and Theobald have also been working on applying concepts from latency-insensitive design to the construction of NOCs having arbitrary topologies and multiple-clock domains [215].

As reported in Section 4.5.3, Chelcea and Nowick have presented a family of low-latency high-throughput interface circuits that makes it possible to extend the idea of latency-insensitive protocols to designs with mixed-timing domains (synchronous, asynchronous, multiple clocks, ...) [49].

Hassoun and Alpert rely on the concept of relay stations as the basic synchronization elements in their approach to achieve simultaneous routing and buffer insertion for globally-asynchronous locally-synchronous SOC architectures [102, 103]

Casu and Macchiarulo have proposed an alternative implementation for the building blocks of a latency-insensitive protocol [43] which applies to the particular case when the computation of each core module can be scheduled statically (see Section 5.2). Their implementation consists of building a shell circuit that stalls its core according to a periodic scheduling sequence, which is stored in a local shift register. Hence, such shell does not need to read the values of *void* signals and *stop* signals and these, therefore, can be removed. Also, since in this particular case there is no need for back-pressure, the authors can replace relay stations with normal unit-capacity stateful repeaters such as edge-triggered flip-flops.

Building on previous work on the POLYCHRONY design environment [95, 230, 231, 236], Talpin and Le Guernic presented a process algebraic theory of behavioral type systems and applied it to the synthesis of latency-insensitive protocols. They showed that the synthesis of component wrappers can be optimized using the behavioral information carried by interface-type descriptions to yield minimized stalls and maximized throughput [229].

Recently, Edman and Svensson have proposed a design methodology to address the timing closure problem in IC design [82]. Their work shares several commonalities with latency-insensitive design: in particular, the idea of preserving the traditional synchronous design paradigm while increasing the design robustness with respect to arbitrary latency variations of on-chip interconnect and facilitating the assembly of IP blocks.

Cortadella *et al.* [63, 64] and Davare *et. al.* [72] have worked on a *desynchronization* approach targeting hardware design: the basic idea is to start from a fully synchronous synthesized (or manually designed) integrated circuit, and then replace automatically the global clock network with a set of local handshaking circuits. In other words, the goal is to derive automatically an asynchronous circuit implementation from a synchronous specification. The main advantage of this approach is the elimination of the global clock signal together with its big design overhead in terms of area occupation, power dissipation, electro-magnetic interference (EMI), and sensitivity to signal integrity and process variation issues [193, 238]. The work on desynchronization clearly parallels the idea—first described in my 1999 ICCAD paper [34]—of not to force a synchronous implementation, but, instead, to utilize well-accepted design methodologies in the specification and validation of synchronous circuits and systems.

8.3 Avenues of Future Research

In the near future the electronic systems industry will evolve through a series of major paradigm shifts. While systems-on-chip increasingly host components of various natures (digital, analog, RF, MEMS ...), the combined impact of power dissipation, signal integrity, and process variations will continue to challenge traditional design methods. It will be a daunting task to manage the complexity of a future SOC and the prospect of the expected costs of running a silicon foundry for a future technology process is anything but encouraging. At the same time, a revolution is happening at the application level where the pervasiveness of embedded systems in our daily lives continuously changes our notion of computation. General-purpose computers will still play a role, but they are unlikely to hold center stage. Moreover, *embedded software* increasingly drives the development costs of embedded systems, accounting for more than 70% in the case of automotive electronics, avionics, and communication networks. In this scenario, traditional distinctions such as "hardware vs. software" or "analog vs. digital" are likely to lose importance. The challenge becomes how to specify, implement and verify a *heterogeneous system* in an efficient and robust way.

I believe that the ideas presented in this dissertation as well as the general principles behind them can be used to guide research efforts along the following avenues:

- *Physical Modeling vs. Design Abstraction.* The physical design of an integrated circuit with nanometer technologies is affected by many phenomena that were considered second-order effects until recently. Still, running a SPICE simulation to analyze the capacitive coupling of every pair of close wires on the chip is unrealistic. Instead, the complexity of today's SOCs calls for raising the level of abstraction. The issue is how to reach a balance between seeking a more optimized design with more accurate models of the physics and making conservative choices to shield the implementation from physical effects. Similar trade-offs apply to most embedded systems that interact continuously with physical processes. For these systems the goal is to derive a reasonable abstraction of the non-idealities of the physical world in order to properly model the environment where they must operate.

- *Local Variations and Global Reliability.* How to design computing systems that are robust with respect to bounded variations from the ideal conditions of operation of their components? For instance, future SOCs will present aggressive voltage-scaling mechanisms at the component level. This may occasionally prevent a component from producing the expected results (or from doing it in the expected time frame). How then to introduce redundancy and flexibility to conjugate the benefits of low-power design with the requirement of having a correct average-case behavior for the overall system? Another example is the case of a sensor network whose nodes operate independently under a self-tuning/self-timing mechanism. How to introduce redundancy and flexibility to reach a degree of inter-node connectivity that enables the correct operations of the overall network? Communication protocols that guarantee the robustness of global properties without imposing tight constraints on local activities represent a promising research avenue for these distributed systems.

- *Design Technology for Heterogeneous Systems.* The electronic system of a modern car is a heterogeneous mix of components (sensors, actuators, ECUs ...) and networks (CAN, FlexRay, MOST ...) that are best described using different models of computation. Designers face a hard task writing embedded software for these architectures. Furthermore, their work on these cost-sensitive and safety-critical applications has important economic implications. For instance, if a software bug is found in the drive-by-wire subsystem after it has been deployed as part of a large-scale production car, the economic damage is likely to be more similar to the impact of a mistake in the hardware of a high-end microprocessor than the consequences of a bug in a desktop application. This is just one of the many commonalities between embedded software programming and integrated circuit design. Other important similarities are the distributed heterogeneous nature of both embedded systems and future SOCs and the demand for tools to assist designers in the specification, synthesis and verification of such systems. Since effective design automation tools require solid mathematical foundations, advanced research in system design must include the study of formal methods as the basis for their development.

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. on Programming Languages and Systems*, 15(1):1–41, 1993. 70

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems*, 17(3):507–534, 1995. 65, 71

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *The 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000. 22, 23

[4] A. Allan, D. Edenfeld, W.H. Joyner Jr., A.B. Kahng, M. Rodgers, and Y. Zorian. 2001 Technology roadmap for semiconductors. *IEEE Computer*, 35(1):42–53, January 2002. 20

[5] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2002. 25, 65

[6] Francois Baccelli, Guy Cohen, Geert J. Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity*. Wiley, New York, 1992. 119, 153

[7] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, Reading,MA, 1990. 26, 107, 201

[8] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992. 66

[9] A. Benveniste. Some synchronization issues when designing embedded systems from componenents. In T.A. Henzinger and C.M. Kirsch, editors, *Embedded Software. Proceeding of the First International Workshop, EMSOFT 2001. Tahoe City, CA*, volume 2211 of *Lecture Notes in Computer Science*, pages 32–49, Berlin, October 2001. Springer Verlag. 72, 73, 74, 75

[10] A. Benveniste. Non-massive, non-high performance, distributed computing: Selected issues. In B. Monien and R. Feldmann, editors, *Proc. of the Euro-Par'2002, Parallel Processing, Paderborn, Germany*, volume 2400 of *Lecture Notes in Computer Science*, pages 29–48, Berlin, August 2002. Springer Verlag. vi, 8, 9, 10

[11] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling: Capturing causality and the correctness of loosely time-triggered architectures (LTTA). In G. Buttazzo and S. Edwards, editors, *Proc. of the Fourth ACM Intl. Conf. on Embedded Software (EMSOFT). Pisa, Italy*, September 2004. 71

[12] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99. Concurrency Theory, 10th International Conference*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177, Berlin, August 1999. Springer Verlag. 71

[13] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163:125–171, 2000. 7, 71, 72, 73

[14] A. Benveniste, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In R. Alur and I. Lee, editors, *Proc. of the Third Intl. Conf. on Embedded Software (EMSOFT). Philadelphia, PA*, volume 2855 of *Lecture Notes in Computer Science*, pages 35–50, Berlin, October 2003. Springer Verlag. 10, 71

[15] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Si-

mone. The synchronous language twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003. 7, 9, 71

[16] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control*, 5:535–546, May 1990. 7, 38, 71, 73

[17] G. Berry. *The Foundations of Esterel.* MIT Press, 2000. 7, 71

[18] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, 1996. 155

[19] M. T. Bohr. Silicon trends and limits for advanced microprocessors. *Communication of the ACM*, 41(3):80–87, March 1998. 21

[20] M.T. Bohr. Interconnect scaling - the real limiter to high performance ULSI. *IEEE International Electron Devices Meeting*, pages 241–244, December 1995. 21

[21] M. Borgatti, C. Auricchio, R. Pelliconi, R. Canegallo, C. Gazzina, A. Tosoni, and P. Rolandi. A multi-context 6.4Gb/s/channel on-chip communication network using 0.18μm flash-EEPROM switches and elastic interconnects. In *ISSCC Digest of Technical Papers*, February 2003. 111

[22] F.-R. Boyer, E. M. Aboulhamid, Y. Savaria, and I.-E. Bennour. Optimal design of synchronous circuits using software pipelining techniques. In *Proc. Intl. Conf. on Computer Design*, pages 62–67, 1998. 223, 224

[23] F.-R. Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer. Optimal design of synchronous circuits using software pipelining techniques. *ACM Trans. on Design Automation of Electronic Systems*, 6(4), 2001. 223, 224

[24] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-6(6):1062–1081, November 1987. 25

[25] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984. 25

[26] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988. 74

[27] J. A. Brzozowski and J. C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992. 66

[28] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, University of California, Berkeley, Electronics Research Laboratory, December 1993. Memorandum No. UCB/ERL 93/69. 156

[29] D. Burger and J. R. Goodman. Billion-transistor architectures: There and back again. *IEEE Computer*, 37(3):23–28, March 2004. 25

[30] S. M. Burns and A. J. Martin. Performance analysis and optimization of asynchronous circuits. In *Advanced Research in VLSI*, pages 71–86. MIT Press, 1991. 154

[31] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. Master's thesis, California Institute of Technology, 1988. 154

[32] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1991. 66, 121, 122, 153, 154

[33] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998. 223

[34] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for "correct-by-construction" latency insensitive design. In *Proc. Intl.*

Conf. on Computer-Aided Design, pages 309–315. IEEE, November 1999. 110, 229, 232

[35] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for "correct-by-construction" latency insensitive design. In Andreas Kuehlmann, editor, *The Best of ICCAD - 20 Years of Excellence in Computer-Aided Design*, chapter 12, pages 143–158. Kluwer Academic Publishers, 2003. 230

[36] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verification*, volume 1633, pages 123–133, Trento, Italy, July 1999. Springer Verlag. 31, 229

[37] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001. 31, 110

[38] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. of the Design Automation Conf.*, pages 361–367. IEEE, June 2000. 157

[39] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, Sep-Oct 2002. 10, 26

[40] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *Proc. of the 16th Symposium on Integrated Circuits and System Design, SBCCI 2003*. IEEE, September 2003. 203

[41] L. P. Carloni and A. L. Sangiovanni-Vincentelli. A formal modeling framework for deploying synchronous designs on distributed architectures. In *FMGALS 2003: First Intl. Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Architectures*, pages 11–31, September 2003. 74, 109

[42] L. P. Carloni and A. L. Sangiovanni-Vincentelli. On-chip communication design: Roadblocks and avenues. In *First IEEE/ACM/IFIP Intl. Conf. on Hardware/Software*

Codesign & System Synthesis. IEEE, October 2003. Extended abstract for invited talk. 109

[43] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proc. of the Design Automation Conf.*, pages 576–581, June 2004. 231

[44] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992. 83

[45] Anantha P. Chandrakasan. *Low Power Digital CMOS Design*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, August 1994. 83

[46] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SOC Revolution: A Guide to Platform Based Design*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1999. 25

[47] L. F. Chao and E. H. M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997. 223

[48] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984. 71, 84

[49] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proc. of the Design Automation Conf.*, 2001. 110, 231

[50] K.-T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. of the Design Automation Conf.*, pages 615–621, June 1994. 200

[51] David Chinnery and Kurt Keutzer. *Closing the Gap Between Asic & Custom: Tools and Techniques for High-Performance Asic Design*. Kluwer Academic Publishers, 2002. 25

[52] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press. 66

[53] Wesley A. Clark and Charles E. Molnar. Macromodular computer systems. In Ralph W. Stacy and Bruce D. Waxman, editors, *Computers in Biomedical Research*, volume IV, chapter 3, pages 45–85. Academic Press, 1974. 66

[54] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *4th Annual Symposium on Logic in Computer Science*, Asilomar, CA, June 1989. 65

[55] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000. 70

[56] P. Cocchini. Concurrent flip-flop and repeater insertion for high-performance integrated circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 268–273, 2002. 109

[57] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Science*, pages 511–523, 1971. 118, 119

[58] J. Cong. Challenges and opportunities for design innovations in nanometer technologies. In *SRC Design Sciences Concept Paper*, December 1997. 25, 106

[59] J. Cong, L. He, K.Y. Khoo, C.K. Koh, and Z. Pan. Interconnect design for deep submicron ICs. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 478–585. IEEE, November 1997. 106, 201

[60] J. Cong and C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. In *Proc. of the Design Automation Conf.*, pages 330–335, June 1998. 204

[61] J. Cong and X. Yuan. Multilevel global placement with retiming. In *Proc. of the Intl. Symposium on Physical Design*, pages 208–213, June 2003. 109

[62] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, McGraw-Hill, 2001. 114, 117, 131, 207

[63] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From synchronous to asynchronous: an automatic approach. In *Proc. of the Conf. on Design, Automation and Test in Europe*, March 2004. 71, 231

[64] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for de-synchronization. In *Proc. Intl. Workshop on Logic Synthesis*, Laguna Beach, CA, May 2003. 71, 231

[65] David E. Culler and Jaswinder P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1999. 102

[66] U. Cummings, A. Lines, and A. Martin. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994. 69

[67] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. ×*pipes*: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *Proc. Intl. Conf. on Computer Design*, pages 536–541, October 2003. 230

[68] W. J. Dally and A. Chang. The role of custom design in ASIC chips. In *Proc. of the Design Automation Conf.*, pages 643–647, June 2000. 25, 106

[69] W. J. Dally and S. Lacy. VLSI architecture: Past, present and future. In *Proceedings of the Advanced Research in VLSI Conference*, pages 232–241. IEEE, March 1999. 24

[70] William J. Dally and John W. Poulton. *Digital System Engineering*. Cambridge University Press, Cambridge, 1998. 80, 107

[71] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, October 1998. 121, 152, 153

[72] A. Davare, K. Lwin, A. Kondratyev, and A. Sangiovanni-Vincentelli. The best of both worlds: The efficient asynchronous implementation of synchronous specifications. In *Proc. of the Design Automation Conf.*, pages 588–591. ACM/IEEE, June 2004. 231

[73] A. Davis and S. M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995. 66

[74] J. A. Davis, R. Venkatesan, A. Kaloyeros, M. Beylansky, S.J. Souri, K. Banerjee, K.C. Saraswat, A. Rahman, R. Reif, and J.D. Meindl. Interconnect limits on gigascale integration (GSI) in the 21st century. *Proceedings of the IEEE*, 89(3):305–324, March 2001. ix, 21, 22, 23, 106

[75] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, Berlin, 1974. Springer Verlag. 155

[76] Volker Diekert and Grzegorz Rozenberg (Eds.). *The Book of Traces*. World Scientific, 1995. 73

[77] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989. 66, 67

[78] R. Drath. The VISUAL OBJECT NET++ Toolset. Developed at Dept. of Automatic Control and System Engineering. Ilmenau Univ. of Technology. Available at http://www.systemtechnik.tu-ilmenau.de/~drath/visual_E.htm. 115

[79] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991. 66

[80] Andreas Kuehlmann (Ed.). *The Best of ICCAD - 20 Years of Excellence in Computer-Aided Design*. Kluwer Academic Publishers, 2003. 230

[81] D. Edenfeld, A.B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *IEEE Computer*, 37(1):47–56, January 2004. 19

[82] A. Edman and C. Svensson. Timing closure through a globally synchronous, timing partitioned design methodology. In *Proc. of the Design Automation Conf.*, pages 71–74, June 2004. 231

[83] M. Ikeda *et al.* A Hardware/Software Concurrent Design for Real-Time SP@ML MPEG2 Video-Encoder Chip Set. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 320–326, March 1996. 173

[84] M. Ikeda *et al.* SuperENC: MPEG-2 Video Encoder Chip. *IEEE Micro*, 19(4):56–65, July 1999. 173

[85] D. Filo, D. Ku, C. Coelho, and G. De Micheli. Interface optimization for concurrent systems under timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):268–281, September 1993. 69

[86] R. W. Floyd. Algorithm 97 {SHORTEST PATH}. *Communications of the ACM*, 5(6):345, 1962. 122

[87] M.J. Flynn, P. Hung, and K.W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–13, July 1999. 21

[88] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., New York, 1979. 155

[89] B. A. Gieseke, R. L. Allmon, D. W. Bailey, B. J. Benschneider, S. M. Britton, J. D. Clouser, H. R. F. III, J. A. Farrell, M. K. Gowan, C. L. Houghton, J. B. Keller, T. H. Lee, D. Leibholz, S. C. Lowell, M. D. Matson, R. J. Matthew, V. Peng, M. D. Quinn, D. A. Priore, M. J. Smith, , and K. E. Wilcox. A 600 MHz superscalar RISC microprocessor with out-of-order execution. In *ISSCC Digest of Technical Papers*, pages 176–177, February 1997. 97, 99

[90] L. P.P.P. Van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *Proc. of ISCC*, pages 865–868, 1990. 107, 201

[91] P. Glaskowski. PENTIUM®4 (partially) previewed. *Microprocessor Report*, 14(8):10–13, August 2000. 24, 97

[92] W. Gosti, A. Narayan, R.K. Brayton, and A. Sangiovanni-Vincentelli. Wireplanning in logic synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 26–33. IEEE, November 1998. 107

[93] Wilsin Gosti. *Layout Aware Synthesis*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, December 2000. 107

[94] P. Le Guernic and T. Gautier. Data-flow to von Neumann : The SIGNAL approach. In J.L. Gaudiot and L. Bic, editors, *Advanced topics in data-flow computing*, pages 413–438. Prentice Hall, 1991. 73

[95] P. Le Guernic, J. P. Talpin, and J. C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12(3):261–303, April 2003. 231

[96] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101(2):265–288, 1992. 87, 134

[97] J. Gunawardena. Periodic behaviour in timed systems with (AND,OR) causality. part I: systems of dimension 1 and 2. Technical report, Department of Computer Science, Stanford University, Stanford, CA 94305, USA., 1993. 153

[98] J. Gunawardena. Min-max functions. *Discrete Event Dynamic Systems*, 4:377–406, 1994. 153

[99] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 7, 71

[100] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers", 1993. 7, 9, 71

[101] M. Hartmann and J. B. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567–574, 1993. 153

[102] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(11):1580–1588, November 2003. 231

[103] S. Hassoun, C. J. Alpert, and M. Thiagarajan. Optimal buffered routing path constructions for single and multiple clock domain systems. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 247–253, 2002. 231

[104] S. Hassoun and C. Ebeling. Architectural retiming: Pipeline latency-constrained circuits. In *Proc. of the Design Automation Conf.*, pages 708–713, June 1996. 224

[105] S. Hassoun and C. Ebeling. Using precomputation in architecture and logic synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 316–323, November 1998. 224

[106] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995. 66

[107] S. Hayati, A. Parker, and J. Granacki. Automatic production of controller specifications from control and timing behavioral descriptions. In *Proc. of the Design Automation Conf.*, pages 75–80, June 1989. 200

[108] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 2002. 78, 99, 101

[109] T.A. Henzinger, S. Qadeer, and R.K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. of the 10th Intl. Conf. on Computer-Aided Verification*, pages 440–451, Vancouver, Canada, July 1998. 65

[110] R. J. Higgins. *Digital Signal Processing in VLSI*. Analog Devices Technical Reference Books. Prentice Hall, 1990. 5

[111] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium®4. *Intel Technology Journal. Q1 Issue*, February 2001. 24, 97

[112] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001. vi, 21, 22, 23, 24, 97, 106

[113] R. Ho, K. Mai, and M. Horowitz. Managing wire scaling: A circuit perspective. In *IEEE Interconnect Technology Conference*, June 2003. 22, 106, 108

[114] S. Hojat and P. Villarrubia. An integrated placement and synthesis approach for timing closure of Power PC microprocessors. In *Proc. Intl. Conf. on Computer Design. VLSI in Computers and Processors*, pages 206–210. IEEE, October 1997. 107

[115] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading,MA, 1979. 156

[116] Accellera Organization Inc. SYSTEMVERILOG language reference manual. Available at http://www.systemverilog.org. 79

[117] ITRS. The international technology roadmap for semiconductors. *Available at http://www.public.itrs.net*, 2003. 16, 19, 20, 230

[118] 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 *8th IEEE International Symposium on Asynchronous Circuits and Systems*, April 2002. 110, 111

[119] A. Jalabert, L. Benini, S. Murali, and G. De Micheli. ×*pipesCompiler*: a tool for instantiating application-specific NoCs. In *Proc. of the Conf. on Design, Automation and Test in Europe*, February 2004. 230

[120] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4:77–84, 1975. 121

[121] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proc. Hawaii International Conf. System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993. 66

[122] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74. International Federation for Information Processing, North-Holland Publishing Company.*, pages 471–475, 1974. 200

[123] A. B. Kahng. Design technology productivity in the DSM era. In *Proc. of the Asia and South Pacific Design Automation Conference*, pages 443–448, January 2001. 18

[124] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *Synthesis of FSMs: Functional Optimization.* Kluwer Academic Publishers, 1997. 4, 60, 208

[125] L.N. Kannan, P.R. Suaris, and H. Fang. A methodology and algorithms for post-placement delay optimization. In *Proc. of the Design Automation Conf.*, pages 327–332, June 1994. 107

[126] H. Kapadia and M. Horowitz. Using partitioning to help convergence in the standard-cell design automation method. In *Proc. of the Design Automation Conf.*, pages 592–597, June 1999. 25, 106

[127] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978. 123, 153

[128] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):309–311, November 1966. 155

[129] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March 1999. 97, 99

[130] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owen, and B. Towles. Exploring the VLSI scalability of stream processors. In *Proc. of the Symposium on High Performance Computer Architecture*, pages 153–164, Anaheim, California, USA, February 2003. 24

[131] K.S. Khouri, G. Lakkshminarayana, and N.K. Jha. High-level synthesis of low-power control-flow intensive circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1715–1729, December 1999. 200

[132] F. Klass. Latches and flip-flops. In V. G. Oklobdzija, editor, *The Computer Engineering Handbook*, pages 10.35–10.70. CRC Press, New York, 2002. 27, 80, 206

[133] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Advanced Computer Science Series. Hemisphere Publishing Corporation; McGraw-Hill Book Company, 1981. 78

[134] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 56–62, June 1994. 66

[135] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. In *IEEE Computer*, June 2000. 24

[136] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Proc. of the Design Automation Conf.*, pages 59–64, June 1990. 69

[137] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press; 3rd edition, 2000(1962,1970). 1

[138] L. Lamport. What good is temporal logic? *Proc. IFIP 9th World Congress*, 5:657–658, 1983. 73

[139] L. Lamport. Composition: A way to make proofs harder. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423, Berlin, 1998. Springer Verlag. 70, 71

[140] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rhinehart and Winston, 1976. 152, 207

[141] E. A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991. 155

[142] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987. 155

[143] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. 155

[144] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998. 2, 34, 37, 48

[145] Edward A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, May 1986. 155

[146] Tak Kwan Lee. *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1995. Technical report CS-TR-95-07. 154

[147] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proc. of the Third Caltech Conf.*, pages 86–116. Computer Science Press, 1983. 204

[148] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983. 204, 207

[149] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. Technical Report SRC-RR-13, Digital-Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, August 1986. 204, 206, 207, 208

[150] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991. viii, 201, 204, 205, 206, 207, 208, 222

[151] Charles E. Leiserson. *Area-Efficient VLSI Computation.* PhD thesis, Massachusetts Institute of Technology, 1983. MIT Press. 204

[152] T. Lin and L. T. Pileggi. Throughput-driven IC communication fabric synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 274–279, 2002. 201

[153] Mikko H. Lipasti. *Value Locality and Speculative Execution.* PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, Carnagie Mellon University, Pittsburgh, Pennsylvania 15213, May 1997. 99

[154] A. Lu, H. Eisenmann, G. Stenz G., and F.M. Johannes. Combining technology mapping with post-placement resynthesis for performance optimization. In *Proc. Intl. Conf. on Computer Design. VLSI in Computers and Processors*, pages 616–621. IEEE, October 1998. 107

[155] R. Lu, G. Zhong, C.K. Koh, and J.Y. Chao. Flip-flop and repeater insertion for early interconnect planning. In *Proc. of the Conf. on Design, Automation and Test in Europe*, March 2002. 109

[156] J. Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters*, 18(1):7–13, 1984. 123

[157] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proc. Annual International Symposium on Computer Architecture*, June 2000. 24

[158] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture.* PhD thesis, California Institute of Technology, 1998. Available as Caltech technical report CS-TR-98-12. 69

[159] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990. 66

[160] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997. 69

[161] A. Mathur, A. Dasdan, and R. K. Gupta. Rate analysis for embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, 3(3):408–436, July 1998. 154, 200

[162] The MathWorks. MATLAB® & SIMULINK® Based Books. Available at http://www.mathworks.com/support/books/. 10

[163] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 8(9):37–39, September 1997. 21, 22, 106

[164] R. McInerney, K. Leeper, T. Hill, H. Chan, B. Basaran, and L. McQuiddy. Methodology for repeater insertion management in the RTL, floorplan and fullchip timing databases of the ITANIUM^{TM} microprocessor. In *Proc. of the Intl. Symposium on Physical Design*, pages 99–104, 2000. 27

[165] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. of the 9th Intl. Conf. on Computer-Aided Verification*, pages 24–35, Haifa, Israel, June 1997. 65

[166] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. of the 10th Intl. Conf. on Computer-Aided Verification*, pages 110–121, Vancouver, Canada, July 1998. 65

[167] J. D. Meindl. Interconnect opportunites for gigascale integration. *IEEE Micro*, 2003. 21, 22, 106

[168] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. Electrical and Computer Engineering Series. McGraw-Hill Book Company, 1994. 25, 173, 207

[169] M. Mizuno, W. J. Dally, and H. Onishi. Elastic interconnects: Repeater-inserted long wiring capable of compressing and decompressing data. In *ISSCC Digest of Technical Papers*, pages 346–347, February 2001. 111

[170] Mohand Mokhtari, Michel Marie, Cecile Davy, and Martine Neveu. *Engineering Applications of* MATLAB® *5.3 and Simulink® 3*. Springer-Verlag, 2000. 10

[171] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985. 66

[172] G.E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38:114–117, April 1965. 16

[173] G.E. Moore. No exponential is forever: but "Forever" can be delayed. In *ISSCC Digest of Technical Papers*, pages 20–23, February 2003. vi, 16, 17

[174] Gordon E. Moore. No exponential is forever: But "Forever" can be delayed! Keynote Address at the 2003 International Solid-State Circuits Conference (ISSCC). San Francisco, CA, February 10, 2003. Available at http://www.intel.com/pressroom/archive/speeches/moore20030210.htm, 2003. 16

[175] T. Murata. Petri Nets, marked graphs and circuit-system theory. *Circuits and Systems*, 11(2):2–12, June 1977. 119, 121

[176] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. vi, 114, 115, 118, 119

[177] Praveen K. Murthy. *Scheduling Techniques for Synchronous and Multidimensiona Synchronous Dataflow*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, December 1996. Memorandum No. UCB/ERL M96/79. 155

[178] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *34th Annual International Symposium on Microarchitecture*, 2001. 24

[179] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *Proc. ACM/IEEE Design Automation Conference*, pages 70–76, June 1994. vii, 120, 121, 122, 123

[180] Vojin G. Oklobdzija, Vladimir M. Stojanovic, Dejan M. Markovic, and Nikola M. Nedovic. *Digital System Clocking: High-Performance and Low-Power Aspects*. Wiley, New York, NY, 2003. 83

[181] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing (2nd Edition)*. Prentice Hall, 1999. 88

[182] Open SYSTEMC Initiative (OSCI). The SYSTEMC language reference manual. Available at http://www.systemc.org/. 79

[183] P. Osler, L. Sheffer, P. Saxena, D. Sylvester, and D.A. Kirkpatrick. The great interconnect buffering debate: Are you a chicken or an ostrich? In *Proc. of the Intl. Symposium on Physical Design*, pages 61–61, 2004. 21

[184] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998 (1982). 276

[185] M. C. Papaefthymiou. Understanding retiming through maximum average-weight cycles. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 338–348, 1991. 209

[186] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981. 114

[187] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik. Also, English translation, "Communication with Automata", New York: Griffiss Air Force Based, Tech. Rep. RADC-TR-65-377, Vol.1, Suppl. 1, 1966., 1962. (technical report Schriften des IIM Nr. 3). 114

[188] L. Pileggi. Achieving timing closure for giga-scale IC designs. In *1999 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1999. 25, 106

[189] F. Pogodalla, R. Hersemeule, and P. Coulomb. Fast prototyping: a system design flow for fast design, prototyping and efficient IP reuse. In *Proc. of the 7th Intl. Conf.*

on Hardware/Software Codesign(CODES'99), pages 69–73, Rome, Italy, May 1999. 19

[190] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Erratum to: A. Benveniste, B. Caillaud, P. Le Guernic. Compositionality in dataflow synchronous languages, specification and distributed code generation. Information and Computation 163, 125-171 (2000). Technical Report Available at `http://www.irisa.fr/prive/Benoit.Caillaud/erratum-ic-2003.pdf`, IRISA-INRIA, Campus de Beaulieu, 35042 Rennes cedex, France, September 2003. 73

[191] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Fourth International Conference on Application of Concurrency to System Design*, June 2004. 73, 74

[192] R. H. J. M. Otten and R. K. Brayton. Planning for performance. In *Proc. of the Design Automation Conf.*, pages 122–127, June 1998. 26, 106

[193] Jan Rabaey, Anantha Chadrakasan, and Borivoje Nikolic. *Digital Integrated Circuits: A Design Perspective - Second Edition*. Prentice Hall, 2003. 4, 27, 80, 107, 232

[194] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980. vi, 115, 119, 121

[195] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., February 1974. 117, 154

[196] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 1977. 121

[197] W. Reisig. *On the Semantics of Petri Nets*. Univ. Hamburg, Fachbereich Informatik, Bericht Nr. 100, September 1984. NewsletterInfo: 18,23. 119

[198] W. Reisig. On the semantics of Petri nets. *Formal Models in Programming, IFIP 1985*, pages 347–372, 1985. NewsletterInfo: 18,23. 119

[199] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):309–311, October 1968. 121

[200] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T. P. Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988. 66

[201] D. Sager, G. Hinton, M. Upton, T. Chappell, T.D. Fletcher, and R. Murray S. Samaan. A 0.18μ CMOS IA-32 microprocessor with a 4-GHz integer execution unit. In *ISSCC Digest of Technical Papers*, pages 324–325, February 2001. 24, 97

[202] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation: Theory & Practice*. World Scientific Publishing Company, 1999. 25

[203] K. Sakallah, T. Mudge, and O. Olukotun. Analysis and design of latch-controlled synchronous digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(3):322–333, March 1992. 153

[204] A. Salek, J. Lou, and M. Pedram. A DSM design flow: Putting floorplanning, technology mapping and gate placemente together. In *Proc. of the Design Automation Conf.*, pages 287–290, June 1998. 107

[205] K. Sato, M. Kawarabayashi, H. Emura, and N. Maeda. Post-layout optimization for deep submicron design. In *Proc. of the Design Automation Conf.*, pages 740–745, June 1996. 107

[206] P. Saxena, N. Menezes, P. Cocchini, and D.A. Kirkpatrick. The scaling challenge: can correct-by-construction design help? In *Proc. of the Intl. Symposium on Physical Design*, pages 51–58, 2003. 22, 106

[207] P. Saxena, N. Menezes, P. Cocchini, and D.A. Kirkpatrick. Repeater scaling and its impact on CAD. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):451–462, April 2004. 22, 28, 106

[208] S.J. Schaffer and W.W. LaRue. BONeS DESIGNER: a graphical environment for discrete-event modeling and simulation. In *MASCOTS '94. Proc. of the 2nd. Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 371–374, Los Alamitos - CA, February 1994. IEEE. 102

[209] L. Scheffer. Methodologies and tools for pipelined on-chip interconnect. In *Proc. Intl. Conf. on Computer Design*, pages 152–157, October 2002. 27, 109

[210] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors. *http://www.semichips.org*, 2001. 18

[211] H. Shah, P. Shiu, B. Bell, M. Aldredge, N. Sopory, and J. Davis. Repeater insertion and wire sizing optimization for throughput-centric VLSI global interconnect. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 280–284, 2002. 201

[212] N. Shenoy. Retiming: Theory and practice. *Integration, the VLSI Journal*, 22:1–21, 1997. 204, 208

[213] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, November 1994. 204

[214] Naveed A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, 1999. 25

[215] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proc. of the Conf. on Design, Automation and Test in Europe*, February 2004. 231

[216] Michael J. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Publishing Company, Reading, MA, 1999. 4, 25

[217] G. Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proc. Intl. Conf. Symp. on FPGAs*, pages 177–186. ACM, February 2002. 204, 222

[218] The DLX Software. *ftp://max.stanford.edu/pub/hennessy-patterson.software*. Stanford University, 1994. 104

[219] G. S. Spirakis. Leading-edge and future design challenges - is the classical EDA ready? In *Proc. of the Design Automation Conf.*, page 416, June 2003. 17

[220] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *29th Annual International Symposium on Computer Architecture*, pages 25–36. IEEE, May 2002. 27

[221] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994. 67

[222] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. 66

[223] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 203–211, November 1998. 22, 106

[224] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron ii: A global wiring paradigm. In *Proc. of the Intl. Symposium on Physical Design*, pages 193–200, April 1999. 22, 106

[225] D. Sylvester and K. Keutzer. Rethinking deep-submicron circuit design. *IEEE Computer*, 32(11):25–33, November 1999. 22, 106

[226] D. Sylvester and K. Keutzer. Impact of small process geometries on microarchitecture in system on a chip. *Proceedings of the IEEE*, 89(4):467–489, April 2001. 22, 106

[227] T. G. Szymanski and N. Shenoy. Verifying clock schedules. In *Proc. Intl. Conf. on Computer-Aided Design*, 1992. 153

[228] J. P. Talpin, A. Benveniste, B. Caillaud, and P. Le Guernic. Hierachical normal form for desynchronization. Technical Report 3822, IRISA, 1999. 71

[229] J. P. Talpin and P. Le Guernic. Process algebraic theory of a behavioral type system for protocol synthesis in system design. *Journal of Formal Methods in System Design*, submitted. 231

[230] J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. A polychronous model for high-level component-based system design. In *Proc. of the Conf. on Design, Automation and Test in Europe*, 2003. 231

[231] J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Formal refinement-checking in a system-level design methodology. *Fundamenta Informaticae*, 2004. 231

[232] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972. 131

[233] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2), Mar-Apr 2002. 24

[234] Donald E. Thomas and Philip E. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1996. 25, 65

[235] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal Research and Development*, 11:25–33, January 1967. 99

[236] The POLYCHRONY Toolset. Developed at IRISA. Available at http://www.irisa.fr/espresso/Polychrony/. 231

[237] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986. 66

[238] J. P. Uyemura. VLSI clocking and system design. In *Introduction to VLSI Circuits and Systems*. Wiley, New York, 2002. 232

[239] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1985. 67

[240] Tiziano Villa, Timothy Kam, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *Synthesis of FSMs: Logic Optimization*. Kluwer Academic Publishers, 1997. 4, 60, 208

[241] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997. 24

[242] Jacun Wang. *Timed Petri Nets: Theory and Application*. Kluwer Academic Publishers, 1998. 117

[243] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proc. Intl. Conf. Symp. on FPGAs*, pages 177–186. ACM, February 2003. 204, 222

[244] Ted E. Williams. Latency and throughput tradeoffs in self-timed asynchronous pipelines and rings. Technical Report CSL-TR-90-431, Stanford University, August 1990. 154

[245] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75. IEEE Computer Society Press, April 1997. 154

[246] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1225–1230, October 2000. 131

[247] A. Xie and P. A. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In A. Yakovlev, L. Gomes, and L. Lavagno,

editors, *Hardware Design and Petri Nets*, pages 239–268. Kluwer Academic Publishers, March 2000. 68, 154

[248] Aiguo Xie. *Performance Analysis of Asynchronous Circuits and Systems*. PhD thesis, University of Southern California, August 1999. 68

[249] Alex Yakovlev, Luis Gomes, and Luciano Lavagno (Eds.). *Hardware Design and Petri Nets*. Kluwer Academic Publishers, 2000. 154

[250] T. Yamada and S. Kataoka. On some LP problems for performance evaluation of marked graphs. *IEEE Transactions on Automatic Control*, 39(3):696–698, 1994. 123

Appendix A

Rate Equalization and Cycle Balancing Problems

A.1 Rate Equalization Problems

In the most general case the rate equalization problem is defined as follows:

Problem A.1 (Rate Equalization Problem (RE)) ***Given:*** *K pairs of nonnegative integers* $(a_1, b_1), \ldots, (a_K, b_K) \in \mathbf{Z}^+ \times \mathbf{Z}^*$.

Minimize: *The cost*

$$g = \sum_{k=1}^{k=K} |m'_k| + |n'_k|$$

over all tuples of pairs of integer variables $((m'_1, n'_1), \ldots, (m'_K, n'_K))$ *such that*

$$\forall k, (m'_k, n'_k) \in \Big([(1-a_k), \infty[\; \times \; [-(b_k - a_k), \infty[\Big)$$

Subject to:

$$\exists r \in \mathbf{Q}^+, r \leq 1, \;\; \forall k \in [1, K], \;\; \Big(\frac{a_k + m'_k}{b_k + m'_k + n'_k} = r \Big) \tag{A.1}$$

□

It is easy to see that this problem presents always a solution. In fact, for any instance of the problem there is always at least one distinct tuple $(\hat{m}_1, \hat{n}_1), \ldots, (\hat{m}_K, \hat{n}_K)$ that satisfies constraint (A.1). This is the *trivial tuple* that is obtained by setting $\forall k, (\hat{m}_k, \hat{n}_k) = (0, -(b_k - a_k))$. The trivial tuple gives cost $g_{triv} = \sum_{k=1}^{k=K} |(b_k - a_k)|$ with $r = 1$. Naturally, if the trivial tuple is the one with the minimum cost among all the tuples satisfying constraint (A.1), then it is also the solution of the problem. It is also possible to consider other cost functions: for instance, the $|m'_k|$ and $|n'_k|$ contributions can be weighted with two constants w_1 and w_2:

$$g = \sum_{k=1}^{k=K} w_1 \cdot |m'_k| + w_2 \cdot |n'_k|$$

In the next sections, I first focus on three simplified versions of the rate equalization problem. Differently from the general case of Problem A.1, these problems have in common the fact that integer variables $((m'_1, n'_1), \ldots, (m'_K, n'_K))$ are forced to be nonnegative. First, I consider the 2-pair rate equalization problem that simply equalizes two ratios. Then I consider two cases of multiple pairs and enforce subsequently dual constraints on the variables: the first case requires that $\forall k (m'_k = 0 \ \wedge \ n'_k \geq 0)$ while the second case requires that $\forall k (n'_k = 0 \ \wedge \ m'_k \geq 0)$.

A.1.1 The 2-Pair Decreasing Rate Equalization Problem

For all $a, b \in \mathbf{Z}^+, (0 < a \leq b)$, let $\sigma_{a,b} : \mathbf{Z}^+ \to \mathbf{Q}^+$ be the infinite sequence whose n-term is $\sigma_{a,b}(n) = \frac{a}{b+n}$. Clearly, $\sigma_{a,b}$ is decreasing monotone and converging to 0. The following lemma shows that for all pairs (a, b), there exists a positive integer R such that one term of sequence $\sigma_{a,b}(n)$ is equal to $\frac{1}{R}$ and, furthermore, that the sequence touches all the values $\frac{1}{R+j}$ for $j = 1, 2, \ldots$, while converging to 0.

Lemma A.1 $\forall a, b \in \mathbf{Z}^+, (0 < a \leq b)$, $\exists R \in \mathbf{Z}^+, (R \geq \lceil \frac{b}{a} \rceil)$, *such that*

$$\forall j \in \mathbf{Z}^* \ \exists \overline{n} \in \mathbf{Z}^* \left(\sigma_{a,b}(\overline{n}) = \frac{1}{R+j} \right)$$

Proof. For all $a, b \in \mathbf{Z}^+, (0 < a \leq b)$, and for all $j \in \mathbf{Z}^*$ suppose $\sigma_{a,b}(\overline{n}) = \frac{1}{R+j}$. Then, $\overline{n} = a \cdot (R + j) - b$. Now, let $R = \lceil \frac{b}{a} \rceil$, then $a \cdot R$ is the smallest integer greater than (or equal to) b and for all $j \in \mathbf{Z}^*$, $\overline{n}$ is a nonnegative integer. □

Any two sequences $\sigma_{a,b}$ and $\sigma_{a',b'}$ share an infinite set of common terms.

```
TwoPairEqualizer $(a, b, d', b')$
{Find $(n, n') \in \mathbf{Z}^{*2}$ s.t. $\frac{a}{b+n} = \frac{d'}{b'+n'}$...}
{...and the cost $C = n + n'$ is minimum.}
$n \Leftarrow 0$;  $n' \Leftarrow 0$;  $d \Leftarrow b$;  $d' \Leftarrow b'$;
{Preprocessing}
if $(\frac{a}{d} < \frac{d'}{d'})$ then
    $n \Leftarrow 0$
    $n' \Leftarrow 1$
else if $(\frac{a}{d} > \frac{d'}{d'})$ then
    $n \Leftarrow 1$
    $n' \Leftarrow 0$
end if
{Main Loop}
while $(\frac{a}{d} \neq \frac{d'}{d'})$ do
    while $(\frac{a}{d} < \frac{d'}{d'})$ do
        $d' \Leftarrow b' + n'$
        $n \Leftarrow \lceil \frac{a \cdot d' - d' \cdot b}{d'} \rceil$
        $n' \Leftarrow n' + 1$
    end while
    while $(\frac{a}{d} > \frac{d'}{d'})$ do
        $d \Leftarrow b + n$
        $n' \Leftarrow \lceil \frac{d' \cdot d - a \cdot b'}{a} \rceil$
        $n \Leftarrow n + 1$
    end while
end while
{Postprocessing}
$n \Leftarrow d - b$
$n' \Leftarrow d' - b'$
return $(n, n')$
```

Figure A.1: Algorithm to solve the 2-Pair Decreasing Rate Equalization Problem.

```
1: DECREASINGPAIREQUALIZER $(a_1, b_1, \ldots, a_K, b_K)$
2: {Find $(n_1, \ldots, n_K) \in \mathbf{Z}^{*K}$ s.t. $\exists r \in \mathbf{Q}^+, r \leq 1,\ \forall k \in [1,K],\ (\frac{a_k}{b_k+n_k} = r)$...}
3: {...and the cost $g = \sum_{k=1}^{K} n_k$ is minimum.}
4: {Preprocessing}
5: for $(k = 1; k \leq K; k{+}{+})$ do
6:     $n_k \Leftarrow 0$
7: end for
8: {Main Loop}
9: while ! $(\frac{a_1}{b_1+n_1} = \frac{a_2}{b_2+n_2} = \cdots = \frac{a_K}{b_K+n_K})$ do
10:     {Find indexes $max$ and $min$ associated...}
11:     {...respectively to the pairs with the largest and smallest ratio}
12:     $max \Leftarrow 0$; $min \Leftarrow 0$; $maxRatio \Leftarrow 0$; $minRatio \Leftarrow \infty$;
13:     for $(k = 1; k \leq K; k{+}{+})$ do
14:         if $(\frac{a_k}{b_k+n_k} > maxRatio)$ then
15:             $max \Leftarrow k$
16:             $maxRatio \Leftarrow \frac{a_k}{b_k+n_k}$
17:         end if
18:         if $(\frac{a_k}{b_k+n_k} < minRatio)$ then
19:             $min \Leftarrow k$
20:             $minRatio \Leftarrow \frac{a_k}{b_k+n_k}$
21:         end if
22:     end for
23:     {Compute increment step for the denominator of the largest pair}
24:     $tmp \Leftarrow \frac{a_{max} \cdot (b_{min}+n_{min})}{a_{min}} - b_{max}$
25:     $incrementStep \Leftarrow \lceil tmp - n_{max} \rceil$
26:     $n_{max} \Leftarrow n_{max} + incrementStep$
27: end while
28: return $(n_1, \ldots, n_K)$
```

Figure A.2: Algorithm to solve the Decreasing Rate Equalization Problem.

Lemma A.2 $\forall a,b,a',b' \in \mathbf{Z}^+, (0 < a \leq b), (0 < a' \leq b')$, *there exists an infinite number of pairs* $(\overline{n},\overline{n}')$ *of non-negative integers s.t.* $\sigma_{a,b}(\overline{n}) = \sigma_{a',b'}(\overline{n}')$.

Proof. Apply Lemma A.1 for the two sequences $\sigma_{a,b}, \sigma_{a',b'}$ with $R = \lceil \max\{\frac{b}{a}, \frac{b'}{a'}\} \rceil$. Therefore, $\forall j \in \mathbf{Z}^*, \exists \overline{n}, \overline{n}' \in \mathbf{Z}^*$ s.t. $\sigma_{a,b}(\overline{n}) = \frac{1}{R+j} = \sigma_{a',b'}(\overline{n}')$. □

The goal, however, is to find the first common term between two sequences $\sigma_{a,b}$ and $\sigma_{a',b'}$. Since the two sequences are decreasing monotone this translates into searching the pair of positive integers n, n' such that $\sigma_{a,b}(\overline{n}) = \sigma_{a',b'}(\overline{n}')$ and $n + n'$ is minimum. This problem can be casted as the following instance of Problem A.1.

Problem A.2 (Two-Pair Decreasing Rate Equalization Problem (2n-DRE))

Given: *Two pairs of positive integers* $(a,b),(a',b') \in (\mathbf{Z}^+)^2$ *such that* $a \leq b$, $a' \leq b'$,

Minimize: *The cost* $g = n + n'$ *over all nonnegative integers* $n, n' \in \mathbf{Z}^*$.

Subject to:

$$\frac{a}{b+n} = \frac{a'}{b'+n'} \tag{A.2}$$

□

Lemma A.2 guarantees that this problem has solution, since there exists an infinite number of pairs (n,n') which satisfy constraint (A.2). Fig. A.1 reports a simple algorithm to solve this problem.

A.1.2 The Decreasing Rate Equalization Problem

Lemma A.2 can be easily extended to the case of K sequences $\sigma_{a,b}, \ldots, \sigma_{a_K,b_K}$, that together share an infinite set of common terms.

Lemma A.3 $\forall a_1, b_1, a_2, b_2, \ldots, a_K, b_K \in \mathbf{Z}^+$, *with* $\forall k (0 < a_k \leq b_k)$. *there exists an infinite number of tuples of non-negative integers* $(\overline{n}_1, \ldots, \overline{n}_K) \in (\mathbf{Z}^+)^K$, *s.t.* $\sigma_{a_1,b_1}(\overline{n}_1) = \sigma_{a_2,b_2}(\overline{n}_2) = \cdots = \sigma_{a_K,b_K}(\overline{n}_K)$.

Proof. Apply Lemma A.1 for the K sequences $\sigma_{a_1,b_1}, \sigma_{a_2,b_2}, \ldots, \sigma_{a_K,b_K}$ with

$$R = \left\lceil \max_{k \in [1,K]} \left\{ \frac{b_k}{a_k} \right\} \right\rceil$$

Therefore,

$$\forall j \in \mathbf{Z}^*, \exists(\bar{n}_1,\ldots,\bar{n}_K) \in (\mathbf{Z}^*)^K, \left(\sigma_{a_1,b_1}(\bar{n}_1) = \sigma_{a_2,b_2}(\bar{n}_2) = \cdots = \sigma_{a_K,b_K}(\bar{n}_K) = \frac{1}{R+j}\right)$$

□

Problem A.2 can be extended to the case of K pairs of nonnegative integers as follows.

Problem A.3 (Decreasing Rate Equalization Problem (DRE))

Given: *K pairs of positive integers $(a_1,b_1),(a_2,b_2),\ldots,(a_K,b_K) \in (\mathbf{Z}^+)^2$ such that*

$$\forall k \in K, \left(a_k \leq b_k\right)$$

Minimize: *The cost $g = \sum_{k=1}^{K} n_k$ over all nonnegative integers $n_k \in \mathbf{Z}^*$.*

Subject to:

$$\exists r \in \mathbf{Q}^+, r \leq 1 \;\; \forall k \in [1,K], \;\; \left(\frac{a_k}{b_k+n_k} = r\right) \tag{A.3}$$

□

The solution of this problem can be obtained by applying recursively the procedure of Figure A.1 on a binary tree computational structure, where each node corresponds to one call of the procedure for any two fractions that have not been equalized yet. In fact, the final solution is independent from the order adopted in subsequently selecting pairs of fractions to equalize, as the following lemma proves.

Lemma A.4 *The solution of problem A.3 is unique.*

Proof (by contradiction). Let P_K be an instance of problem A.3 with $(a_1,b_1),(a_2,b_2),\ldots,$ $(a_K,b_K) \in (\mathbf{Z}^*)^2$ being the K pairs of nonnegative integers. Let $S = (n_1,\ldots,n_K)$ be the solution tuple with cost $g(S) = \sum_{k=1}^{K} n_k$ and ratio r. Assume that $S' = (n'_1,\ldots,n'_K)$ is another solution for P_K, with differs from S for at least an element $n'_i \neq n_i$, with $i \in [1,K]$. Without loss of generality, assume that $n'_i < n_i$. Denote with r' the ratio associated to S'. Naturally, $n'_i < n_i$ implies that $r' = \frac{a_i}{b_i+n'_i} > r$. On the other side, since both S and S' are solutions of P_k then, necessarily, $g(S) = g(S')$. Therefore, to compensate the difference in the cost sum, there must be at least one index $j \in [1,K]$ with $j \neq i$ and such that $n'_j > n_j$. But this would

imply that $\frac{a_i}{b_i+n'_i} < q$ and, therefore, $\frac{a_i}{b_i+n'_i} \neq r'$, thus contradicting that S' is a solution of P_k.
□

The algorithm reported in Figure A.2 represents another method, in general more efficient, to solve problem A.3.

A.1.3 The Increasing Rate Equalization Problem

For all $c \in \mathbf{Z}^*$ and $b \in \mathbf{Z}^+$, with $(c \leq b)$, let $\sigma_{c,b} : \mathbf{Z}^+ \longrightarrow \mathbf{Q}^+$ be the infinite sequence whose n-term is $\sigma_{c,b}(n) = \frac{c+n}{b+n}$. Clearly, $\sigma_{c,b}$ is increasing monotone and converging to 1. If $\sigma_{a,b}(n) = \frac{a}{b+n}$ and $c = b - a$, then $\sigma_{c,b}(n) = 1 - \sigma_{a,b}(n)$. As a consequence, results similar to the ones obtained above for sequence $\sigma_{a,b}(n)$ can now be derived for sequence $\sigma_{c,b}(n)$. In particular, it is easy to prove that any two sequences $\sigma_{c,b}$ and $\sigma_{c',b'}$ share an infinite set of common terms.

Lemma A.5 *For all $c_1, b_1, c_2, b_2, \ldots, c_K, b_K \in \mathbf{Z}^+$, with $\forall k(c_k \leq b_k)$, there exists an infinite number of tuples of non-negative integers $(\overline{n}_1, \ldots, \overline{n}_K) \in \mathbf{Z}^{+K}$, s.t. $\sigma_{c_1,b_1}(\overline{n}_1) = \sigma_{c_2,b_2}(\overline{n}_2) = \cdots = \sigma_{c_K,b_K}(\overline{n}_K)$.*

Proof. It can be proven by applying Lemma A.3 to the sequences $\sigma_{a_1,b_1}(\overline{n}_1), \ldots \sigma_{a_K,b_K}(\overline{n}_K)$, after setting $\forall k(a_k = b_k - c_k)$. First, notice that since $\forall k(c_k \leq b_k)$, then also $\forall k(0 < a_k \leq b_k)$. Then, define $R = \lceil \max_{k \in [1,K]} \{\frac{b_k}{a_k}\} \rceil$. By Lemma A.3,

$$\forall j \in \mathbf{Z}^*,\ \exists(\overline{n}_1, \ldots, \overline{n}_K) \in (\mathbf{Z}^*)^K, \left(\sigma_{a_1,b_1}(\overline{n}_1) = \sigma_{a_2,b_2}(\overline{n}_2) = \cdots = \sigma_{a_K,b_K}(\overline{n}_K) = \frac{1}{R+j}\right)$$

Hence, multipling each term of these relation by -1 and then adding 1 to it, gives

$$\forall j \in \mathbf{Z}^*,\ \exists(\overline{n}_1, \ldots, \overline{n}_K) \in (\mathbf{Z}^*)^K,$$
$$1 - \sigma_{a_1,b_1}(\overline{n}_1) = 1 - \sigma_{a_2,b_2}(\overline{n}_2) = \cdots = 1 - \sigma_{a_K,b_K}(\overline{n}_K) = 1 - \frac{1}{R+j}$$

And finally,

$$\forall j \in \mathbf{Z}^*,\ \exists(\overline{n}_1, \ldots, \overline{n}_K) \in (\mathbf{Z}^*)^K, \left(\sigma_{c_1,b_1}(\overline{n}_1) = \sigma_{c_2,b_2}(\overline{n}_2) = \cdots = \sigma_{c_K,b_K}(\overline{n}_K) = \frac{R+j-1}{R+j}\right)$$

□

A dual problem of Problem A.3 can be defined for sequences of the type $\sigma_{c,b}(n)$.

```
RATEEQUALIZER ( $(a_1,b_1),\ldots,(a_K,b_K), justTheSeeds, constraints$ )
{Find $(m'_1 n'_1),\ldots,(m'_K,n'_K) \in [(1-a_k),\infty[ \times [-(b_k-a_k),\infty[$ s.t.}
{$\exists r \in \mathbf{Q}^+, r \leq 1\ \forall k \in [1,K],\ (\frac{a_k-m'_k}{b_k+m'_k+n'_k} = r)$...}
{...and the cost $g = \sum_{k=1}^{K} |m'_k| + |n'_k|$ is minimum.}
$C_{ub} \Leftarrow 0$
for $(k = 1; k \leq K; k++)$ do
    $C_{ub} \Leftarrow C_{ub} + |n_k|$
end for
{Sort input pairs $(a_k, b_k)$ in ascending order by the ratio $\frac{a_k}{b_k}$}
$\mathcal{P} \Leftarrow sortInputPairsByRatio(\ (m_1,n_1),\ldots,(m_K,n_K)\ )$
{Find the candidate equalization seeds}
$CES \Leftarrow$ FINDCANDIDATEEQUALIZATIONSEEDS $(\mathcal{P}, C_{ub})$
{Sort the candidate equalization seeds by their optimality}
$CES_{sort} \Leftarrow sortCandidateEqualizationSeeds(CES)$
if $(justTheSeeds)$ then
    return $CES_{sort}$
end if
{Generate feasible solution of optimization problem}
for all $i \in CES_{sort}$ do
    {Get next best equalization seed (smallest cost)}
    $c \Leftarrow CES_{sort}[i]$
    $(m'_1 n'_1),\ldots,(m'_K,n'_K) \Leftarrow deriveEqualizingPairs(c)$
    {Check whether the solution is within the input constraints}
    if $(isFeasible(\ (m'_1 n'_1),\ldots,(m'_K,n'_K), constraints\ ))$ then
        return $(\ (m'_1 n'_1),\ldots,(m'_K,n'_K)\ )$
    end if
end for
```

Figure A.3: Algorithm to solve the Rate Equalization Problem.

```
1: FINDCANDIDATEEQUALIZATIONSEEDS( $\mathcal{P}, C_{ub}$ )
2: {Compute Starting Equalization Seed Set}
3: $p \Leftarrow \mathcal{P}[0]$;  $M_{max} \Leftarrow C_{ub}$;
4: if $(|1 - a_p| > M_{max})$ then $M_{min} \Leftarrow -M_{max}$ else $M_{min} \Leftarrow 1 - a_p$ end if
5: for $(i = M_{min}; i \leq M_{max}; i++)$ do
6:   $N_{max} \Leftarrow C_{ub} - |i|$
7:   if $(|1 - (b_p - a_p)| > N_{max})$ then $N_{min} \Leftarrow -N_{max}$ else $N_{min} \Leftarrow 0 - (b_p - a_p)$ end if
8:   for $(j = N_{min}; j \leq N_{max}; j++)$ do
9:     {Save seed tuple}
10:    $\sigma = (i, j, \frac{a_p+i}{b_p+i+j}, (i+j))$
11:    $CES \Leftarrow CES \cup \sigma$
12:  end for
13: end for
14: {Filter feasible seeds}
15: $I \Leftarrow createIndexInterleavedArray(\mathcal{P})$
16: for $(k = 1; k > |I|; k++)$ do
17:   $z \Leftarrow \mathcal{P}[I[k]]$
18:   for $(\sigma \in CES)$ do
19:     $matched = FALSE$;  $minMatch = \infty$;  $M_{max} \Leftarrow C_{ub} - \sigma[4]$
20:     if $(|1 - a_z| > M_{max})$ then $M_{min} \Leftarrow -M_{max}$ else $M_{min} \Leftarrow 1 - z_p$ end if
21:     for $(i = M_{min}; i \leq M_{max}; i++)$ do
22:       $N_{max} \Leftarrow C_{ub} - |i|$
23:       if $(|1 - (b_z - a_p)| > N_{max})$ then $N_{min} \Leftarrow -N_{max}$ else $N_{min} \Leftarrow 0 - (b_p - a_p)$ end if
24:       for $(j = N_{min}; j \leq N_{max}; j++)$ do
25:         if $\frac{a_z+i}{b_z+i+j} == \sigma[3]$ then
26:           $matched \Leftarrow true$
27:           if $(C_{ub} - \sigma[4]) + (|i| + |j|) < minMatch$ then $minMatch \Leftarrow (C_{ub} - \sigma[4]) + (|i| + |j|)$ end if
28:         end if
29:       end for
30:     end for
31:     if $(matched)$ then $\sigma[4] \Leftarrow minMatch$ else $CES \Leftarrow CES \setminus \{\sigma\}$ end if
32:   end for
33: end for
```

Figure A.4: Algorithm to find the candidate equalization seeds.

Problem A.4 (Increasing Rate Equalization Problem (IRE))

Given: *K pairs of positive integers* $(c_1, b_1), (c_2, b_2), \ldots, (c_K, b_K) \in (\mathbf{Z}^+)^2$ *such that* $\forall k \in K, (c_k \leq b_k)$,

Minimize: *The cost* $g = \sum_{k=1}^{K} m_k$ *over all nonnegative integers* $m_k \in \mathbf{Z}^*$.

Subject to:

$$\exists z \in \mathbf{Q}^+, z \leq 1 \ \forall k \in [1, K], \ \left(\frac{c_k + m_k}{b_k + m_k} = z \right) \tag{A.4}$$

□

A possible way to find the solution of this problem is to solve the corresponding instance of the dual Problem A.3. This instance is obtained by considering the K pairs of nonnegative integers $(a_1, b_1), (a_2, b_2), \ldots, (a_K, b_K) \in (\mathbf{Z}^*)^2$, where $\forall i \ (a_i = b_i - c_i)$. The tuple $(n_1, \ldots, n_K)$ that represents the optimum solution of Problem A.3 corresponds exactly to the tuple $(m_1, \ldots, m_K)$ representing the optimum solution of Problem A.4. Naturally, the rational numbers r, z in the two problems are related by the fact that $r = 1 - z$.

A.1.4 Solving the Rate Equalization Problem

Algorithm RATEEQUALIZER of Figure A.3 solves the general Rate Equalization Problem (RE). The algorithm start by setting an upper bound C_{ub} for the cost of the solution. This is given by the cost of the trivial tuple discussed in Section A.1. Then, all input pairs are sorted in ascending order of their ratios $\frac{a_k}{b_k}$. This information is passed to routine FIND-CANDIDATEEQUALIZATIONSEEDS, reported in Figure A.4, which does the bulk of the work. The routine takes the pair (a_p, b_p) with the smallest ratio and, by varying the values of m_k, n_k into the corresponding integer intervals such that the cost $|m_k| + |n_k| \leq C_{ub}$, it considers all the possible ratios that can be obtained from $\frac{a_k + m_k}{b_k + m_k + n_k}$. Each of these value is a *candidate equalization seed* with its own ratio and cost. Then, each seed is filter iteratively against all the other pairs (a_k, b_k). At each iteration a new pair (a_z, b_z) is considered. This is chosen based on the interleaving order of the pair ratios. At each iteration, the goal is to verify whether with the "remaining upper bound" available pair (a_p, b_p) can match the ratio of the cost. As soon as a pair cannot match this ratio the seed is discarded. Finally, the set CES_{sort} of candidate equalization seeds is returned to algorithm RATEEQUALIZER which simply sorts it in increasing order of seed cost. For each seed the solution is quickly computed equalizing the original pair to the ratio of the seed. Algorithm RATEEQUALIZER accepts constrains in terms of ranges for the ratio r as well as for the cost g. Only those solutions that are inside these ranges are returned.

A.2 Solving the Cycle Balancing Problem

Given a marked graph $\mathcal{MG}_S$ modeling a latency-insensitive system implementation S, the cycle balancing problem is defined as follows:

Problem A.5 (Cycle Balancing Problem (CPB))

Given: *A marked graph $\mathcal{MG}_S = (P,T,F,W,M_0)$ with a set of cycles $C(\mathcal{MG}_S)$.*

Minimize: *The cost*

$$g = \sum_{t_i \in T} |m'(t_i)| + \sum_{p_j \in P} |n'(p_j)|$$

over all integer variables $n'(t_1),\dots,n'(t_{|T|})$ and $m'(p_1),\dots,m'(p_{|P|})$, with

$$\forall i \in [1,|T|], (n'(t_i) \in [-1,1] \;\wedge\; \forall j \in [1,|P|], (m'(p_j) \in [-1,1]$$

Subject to: $\exists q \in \mathbf{Q}^+, q \geq 1,\ \forall k \in [1,|C(\mathcal{MG}_S)|]$ *:*

$$\frac{\sum_{t_i \in T} x(t_i,c_k) \cdot \left[n(t_i) + n'(t_i)\right]}{\sum_{p_j \in P} y(p_j,c_k) \cdot \left[m(p_j) + m'(p_j)\right]} = q$$

where $P_0 = \{p \in P | M_0(p) = 0\}$ is the set of initially empty places and $\forall t_i \in T$, $\forall p_j \in P$

- $n(t_i) = 1$
- $m(p_j) = \begin{cases} 0 & \textit{if } p_j \in P_0 \\ 1 & \textit{otherwise} \end{cases}$
- $x(t_i,c_k) = \begin{cases} 1 & \textit{if } t_i \in c_k \\ 0 & \textit{otherwise} \end{cases}$

```
1: CycleBalancer $(n(t_1),\ldots,n(t_{|T|}),m(p_1),\ldots,m(p_{|P|}))$,
2: for all $k \in [1,K]$ do
3:     $b_k \Leftarrow \sum_{t_i \in T} x(t_i,C_k) \cdot n(t_i)$
4:     $a_k \Leftarrow \sum_{p_j \in P} y(p_j,C_k) \cdot m(p_j)$
5: end for
6: {Invoke algorithm RateEqualizer until step 16}
7: $CES_{sort} \Leftarrow$ RateEqualizer$\left((a_1,b_1),\ldots,(a_K,b_K),true,\emptyset\right)$
8: for all candidate $c \in CES_{sort}$ do
9:     {Reset set of auxiliary variables}
10:    for all $t_i \in T, p_j \in P$ do
11:        $m'(t_i) \Leftarrow 0$; $E(t_i) \Leftarrow false$
12:        $n'(p_j) \Leftarrow 0$; $E(p_j) \Leftarrow false$
13:    end for
14:    $(M,N) \Leftarrow computeTargetValues(c)$
15:    {Sort cycle row indexes in increasing order of their targets}
16:    $\mathcal{H} \Leftarrow sortCycleRowIndexes(M,N)$
17:    {Fire recursive step to seek new equalization solution}
18:    $s \Leftarrow$ RecursiveStep $\left(K,M,N,\mathcal{H},E,n'(t_i),m'(p_j),0,\emptyset\right)$
19:    if $(s \neq \emptyset)$ then
20:        $\mathcal{S} \Leftarrow \mathcal{S} \cup s$
21:    end if
22: end for
23: return $\mathcal{S}$
```

Figure A.5: Algorithm to solve the Cycle Balancing Problem

```
1: RECURSIVESTEP $\left(K,M,N,\mathcal{H},E,n'(t_i),m'(p_j),h,s\right)$
2: if $(h = K)$ then
3:     {No more rows to equalize. Exit recursion successfully.}
4:     return $s$
5: end if
6: $k \Leftarrow \mathcal{H}[h]$
7: $NN[k] \Leftarrow N[k] - \sum_{t_i \in T} [x(t_i,C_k) \cdot n'(t_i)]$
8: $MM[k] \Leftarrow M[k] - \sum_{p_i \in P} [y(p_j,C_k) \cdot m'(p_j)]$
9: $\mathcal{R} \Leftarrow findSetOfConfigurations(MM[k],NN[k],E)$
10: for all $r \in \mathcal{R}$ do
11:     {Pick configuration r as a possible cycle solution}
12:     for all $t_i, p_j \in C_k$ do
13:         $(n'_s(t_i),m'_s(p_j)) \Leftarrow (n'(t_i),m'(p_j))$
14:         $(n'(t_i),m'(p_j)) \Leftarrow setVariablesFromConfiguration(r,s)$
15:         $E_s(t_i) \Leftarrow E(t_i)$; $E_s(p_j) \Leftarrow E(p_j)$;
16:     end for
17:     if (RECURSIVESTEP $\left(K,M,N,\mathcal{H},E,n'(t_i),m'(p_j),(h+1),s\right) \neq emptyset$) then
18:         $s \Leftarrow (n'(t_1),\ldots,n'(t_{|T|}),m'(p_1),\ldots,m'(p_{|P|}))$
19:         return $s$
20:     else
21:         for all $t_i, p_j \in C_k$ do
22:             $(n'(t_i),m'(p_j)) \Leftarrow (n'_s(t_i),m'_s(p_j))$
23:             $E(t_i) \Leftarrow E_s(t_i)$; $E(p_j) \Leftarrow E_s(p_j)$;
24:         end for
25:     end if
26: end for
27: {No possible equalization found. Exit recursion unsuccessfully.}
28: return $\emptyset$
```

Figure A.6: Recursive step inside algorithm CYCLEBALANCER of Figure A.5

- $y(p_j, c_k) = \begin{cases} 1 & \textit{if } p_j \in c_k \\ 0 & \textit{otherwise} \end{cases}$

□

It is easy to see that the cycle balancing problem has always at least a solution. In fact, the trivial variable configuration that is obtained by setting $\forall t_i$, $(n'(t_i) = 0)$ and $\forall p_j$, $(m'(p_j) = 1 \Leftrightarrow m(p_j) = 0)$ satisfies the problem constraints with $q = 1$. In general several solutions are possible. These may have the same or different values of q.

Algorithm CYCLEBALANCER illustrated in Figure A.5 solves Problem A.5. The algorithm is organized on three basic routines. The main routine invokes routine RATEEQUALIZER to find all the candidate solutions of a relaxed version of Problem A.5. The relaxation consists in the fact that routine RATEEQUALIZER assumes that all the cycles have pairwise empty place/transition intersection. In other words, it gives the exact solution if the marked graph is a connection of distinct strongly connected components (SCCs) each having at most one cycle. Notice that RATEEQUALIZER searches the solution space in a very efficient way because the independence among variables $(m'_1, n'_1), (m'_2, n'_2), \dots, (m'_K, n'_K)$ does not require the use of any branch-and-bound technique [184]. Once all the candidate solutions are returned to CYCLEBALANCER, the core routine RECURSIVESTEP is invoked for each candidate solution. Routine RECURSIVESTEP attempts to *justify* the candidate solution by finding an assignment for the integer variables $n'(t_1), \dots, n'(t_{|T|})$ and $m'(p_1), \dots, m'(p_{|P|})$. This assignment must satisfy the ratio constraint found during the solution of the relaxed problem without violating the reciprocal constraints which cycles that share common variables impose on each other. The search is limited by the maximum number of transformations allowed. All the valid solutions are returned in increasing order of cost g and, among those having the same cost, increasing order of cycle time.

Index