SAME 2006 Forum

Session : TOOLS & METHODOLOGIES

Latency Insensitive Design : Dynamic and static scheduling with proper formal devices

Julien Boucaron, Jean-Vivien Millo and Robert de Simone : INRIA

Sophia-Antipolis, France

Abstract

The theory of latency-insensitive design (LID) was recently invented to cope with the time closure problem in otherwise synchronous circuits and programs. The idea is to allow the inception of arbitrarily fixed (integer) latencies for data/signals traveling along wires. Then mechanisms such as shell wrappers and relay-stations are introduced to "implement" the necessary back-pressure congestion control. As a result the LID is behaviourally equivalent to the synchronous specification, avoid starvation, deadlock and congestion of local synchronous IP. We first revisit the formal modeling of relay stations and shells, and provide a number of properties establishing the soundness of our models. Then we face the issue of latency and throughput "equalization" and moreover the problem of the rational solution that we solve with a "fractional register".

Table of Contents

- Introduction
- Modeling background
- Latency Insensitive Design
  - Dynamic scheduling
  - Static scheduling
- Tool development
- Conclusion and future works

Introduction

Long wire interconnect latencies induce time-closure difficulties in modern SoC designs, with propagation of signals across the dye in a single clock cycle is problematic. The theory of Latency-Insensitive Design (LID), proposed originally by L. Carloni and al. [3, 4], offers solutions for this issue.

LID starts from an ideal synchronous specification model where computation and communication are free, however those assumptions are unrealistic (timing closure, delay). Then after applying a de-synchronization stage with unbounded buffers under an ASAP firing rule, we resynchronize it using wrappers around IP (Shells), specific wire-pipelining (Relay Stations) and a latency insensitive synchronization protocol that respect the partial ordering of events of the specification. LID thus takes a synchronous specification and build an equivalent one with respect to ordering of events where we have stretch the time on a given amount of cycles. We offer a formal specification of the Shell and Relay-Station using synchronous languages (Esterel, SyncCharts) which are amenable to formal verification and proof.

The foundations of the theory of static and k-periodic scheduling for Weighted Marked Graphs is to be found in [7,8]. In [7] the authors name it as the Central Repetitive Problem (CRP). It is formulated as a generic scheduling problem : given a set of tasks, and constraints between them and between different runs of the same task, the goal is to execute this set of tasks infinitely while minimizing time spent for each execution. In [8] it is established that the scheduling of a connected graph with cycles G, is ultimately k-periodic. Indeed we can find a static scheduling of a LID circuit without any synchronization protocol.

In our attempt to design a static predictable scheduling for the whole systems, we are now describing the successive algorithmic steps involved in the process of equalizing the various loop latencies.

This last (The Fractional Register) is used to hold back specific tokens for one instant, so that rates are equalized and the tokens are presented simultaneously to the computation nodes. The desired effect is to compensate the rate difference between cycles sharing computation node(s). It consists of a (one-slot) register, used to "kidnap" the token (and its value in a real setting) for one clock cycle when a data arrive earlier. We have a proper formal design of the Fractional Register.

Modeling background

We start from a very general definition, describing what is common of all our models.

Definition 1 (*Computation Network Scheme*). We call Computation Network Scheme (CNS) a graph whose vertices are called computation nodes, and whose arcs are called data transportation links.
The intention is that nodes perform computations by consuming a data on each of its incoming links, and producing as a result a new data on each of its outgoing links.

Definition 2. *A Weighted Event Graph [6] (WEG)* is a CNS with integer-valued latency figures adorning each computation node and data link. This number indicates the time spent while performing the corresponding computation or data transportation. The corresponding firing rule is: nodes fires immediately when all input tokens are available.

WEGs can be expanded into intermediate models where links are cut into sections by introducing auxiliary transportation nodes (as many as the prescribed latencies). It can be shown that the global system preserves its essential functional properties if the buffering mechanisms in between transportation and computation nodes altogether are limited to two-place buffers. Physical implementation of these mechanisms in such case rely on relay-stations and shell wrappers, composing the core of so-called Latency-Insensitive Design (LID) theory. We recall it in the next section.

Definition 3 *(Rates and critical cycles)*. Let G be a WEG, and C a loop cycle in this graph. The rate r of the loop is equal to T, where T is the L number of tokens in the loop (which is constant), and L is the sum of latencies labeling its arcs. The throughput of the graph is defined as the minimum of rates over all loops. A loop cycle is called critical if it rate is equal to the graph throughput.

Definition 4 *(Schedules)*. A schedule for a computation net is a function Sched : N -> w_N assigning an infinite word w_N {0, 1} to every computation and transportation node of the net. The intuition is that a schedule forces activity at instants where it holds a "1", and inactivity when "0". An infinite word w -> {0, 1} is called ultimately periodic if it is of the form u.(v) where u, v {0, 1}. u represents the initial part, v the periodic one. We call the length of v (noted |v|) the period of w, and the number of 1s in v, noted $|v|_1$, the periodicity of w. The rate r(w) of an ultimately periodic word w is defined as $|v|_1/|v|$. (borrow from [5])

Latency Insensitive Design : dynamic scheduling

LID theory was introduced in [3]. It relies on the fact that data links with latency, seen as physical long wires in synchronous circuits, can be segmented into sections. Specific elements are then introduced the section boundaries. Such elements are called relay-stations (RS). Instantaneous communication is possible inside a given section, but the values have to be buffered inside the RS before it can be propagated to the next section. The problem of computing realistic latencies from physical wire lengths was tackled in [2], where a physical synthesis floor-planner provides these figures. Relay stations are complemented with so-called shell wrappers (SW), which compute the firing condition for their local synchronous component (called "pearl" in LID theory). They do so from the knowledge of availability of input data and output storage slots.

Relay-Station(RS)

The signaling interface of a relay-station is depicted in figure 1. The val signals are used to propagate data/tokens, the stop signal are used for congestion control. For symmetry here stop_out is an input and stop_in an output.

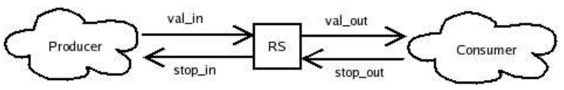

figure 1: RS block diagram

Intuitively the relay-station behaves as follows: when traffic is clear (no stop), each data (token) is propagated down at the next instant from the one it was received. When a stop_out signal is received because of downward congestion, the RS keeps its data value. But then, the previous section and the previous RS cannot be warned instantly of this congestion, and so the current RS can perfectly well receive another data at the same time it has to keep the former one. So there is a need for the RS to provide a second auxiliary register slot to store this second data value. Fortunately there is no need for a third one: in the next instant the RS cell can propagate back a stop_in control information to preserve itself from receiving yet another value. Meanwhile the first data can be sent as soon as stop_out signals are withdrawn, and the cell remains with only one value, so that in the next step it can already allow a new one and not send its congestion control signal. Note that in this scheme there is no undue gap between the data sent.

This informal description is made formal with the synchronous circuit and a FSM in [1].

Shell wrappers

The purpose of shell wrappers is to trigger the local computation node exactly when data are available from each input data link, and there is storage available for result in output data links. We do suppose that computation nodes are combinatorial elements, with time latencies expressed outside them in the relay sections.
The signal interface of SWs consists of val_in and stop_in signals indexed by the number of input data links to the SW, and of val_out and stop_out signals indexed by the number of its output data links. There is an output clock signal in addition, to fire the local component. This last signal will be scheduled at the rate of local firing thus. Note that it is here synchronous with all the val_out signals when values are abstracted into tokens.

The operational behavior of the SW is depicted as a synchronous circuit in figure 2.

The Shell works as follows:

- The internal pearl's clock and all val_out$_i$ valid output signals are generated once we have all val_in, while stop is false. The internal stop signal itself represents the disjunction of all incoming stop_out$_j$ signals from outcoming channels;
- The buffering register of a given input channel is used meanwhile as long as not all other input data are available;
- So, internal pearl's clock is set to false when ever a backward stop_out$_j$ occurs as true, or a forward val_in$_i$ is false. In such case the registers already busy hold their true value, while others may receive a valid data "just now";
- Stop_in$_i$ signals are raised towards all channels whose corresponding register was already loaded (a data was received before, and still not consumed), to warn them not to propagate any value in this clock cycle. Of course such signal cannot be sent in case the data is currently received, as it would raise a causality paradox (and a combinatorial cycle).
- Flip-flop registers are reset when the pearl's clock is raised, as it consumes the input data. Following the previous remark, the signal stop_in$_i$ holding back the traffic in channel i is raised for these channels where the data have arrived before the current instant, even in this case.

One should note that the constraint demanded by the relay stations for proper functioning holds here: each output channel from the producer (is this case the shell), one has stop_out$_j$ => ¬val_out$_j$.

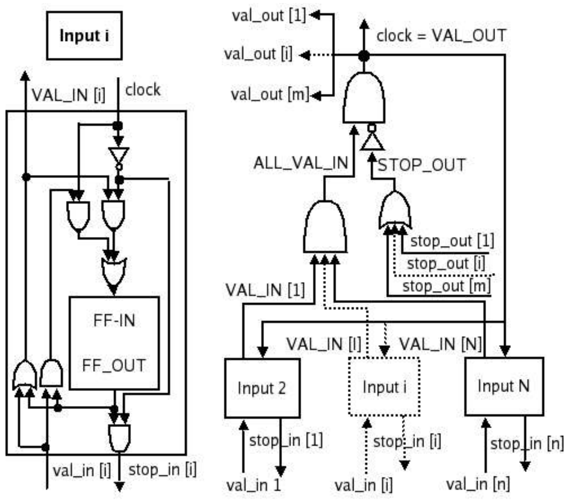

figure 2: Shell circuitry

Latency Insensitive Design : Static Scheduling

We now turn to the issue of providing static periodic schedules for LID systems. According to the previous philosophy governing the design of relay-stations, we want to provide solutions where tokens are not allowed to accumulate at places in large numbers. In fact we will attempt to equalize the flows so that tokens arrive as much as possible simultaneously at their joint computation nodes.

An optimal time schedule can be built without altering the global throughput on the topology of the LID net (see [2]). Equalization is a mean to slow-down uncritical cycles to the nearest rate of critical cycles. Indeed Equalization does not change or slow-down the rate of critical cycles, thus we can still find a periodic schedule achieving the same throughput while relaxing the rate of uncritical cycles.

We describe the successive steps:
Global throughput evaluation: We need to compute the best feasible global rate, which is the slowest rate (noted R) amongst individual loop cycles. For this we do enumerate all elementary cycles and compute their rates.

Integer latency insertion: This is solved by linear programming techniques. Linear equation systems are built to express that all elementary cycles, with possible extra variable latencies on arcs, should now be of rate R, the previously computed global throughput. The equations are also formed while enumerating the cycles in the previous phase. The particular shape of the equation system lends itself well to a direct greedy algorithm, stuffing incremental additional integer latencies into the existing systems until completion. This was confirmed by our prototype implementations.

Schedule computation (using state space construction): In order to compute the explicit schedules of the initial and stationary phases we currently need to simulate the system's behavior. We also need to store visited state, as a termination criterion for the simulation whenever an already visited state is reached. The purpose is to build the schedule patterns of computation nodes to determine where residual fractional latency elements have to be inserted.

Fractional latencies: In an ideally equalized system, the schedules of distinct computation/transportation nodes should be precisely related: the schedule of the "next" node should be that of the "previous" node shifted one slot right. After we compute the effective schedules, one can whether this is the case. If not, then extra fractional registers need to be inserted just after the regular register already set between the nodes. This FR element should delay discriminatingly some tokens (but not all).

We shall introduce a formal model of our FR elements in the next subsection. The block diagram of its interfaces are displayed in figure 3.

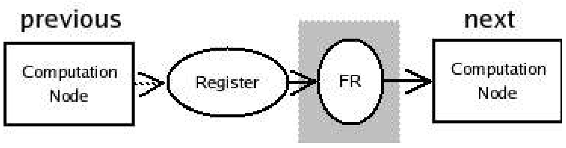

figure 3: FR insertion in the network

Fractional register element (FR)

We now formally describe the specific FR synchronous elements, both as a synchronous circuit in figure 4(b) and as a corresponding Mealy FSM in figure 4(a). The FR interface consists of two input wires TokenIn and Hold, and one output wire TokenOut. Its internal state consists of a register CatchReg. The register will be used to "kidnap" the token (and its value in a real setting) for one clock cycle whenever Hold holds. We note pre(CatchReg) the (boolean) value of the register computed at the previous clock cycle. It indicates whether the slot is currently occupied or free.

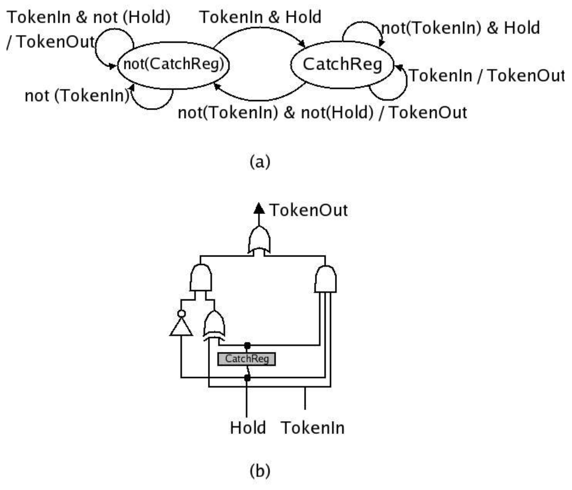

figure 4: The automaton (a) and the circuitry (b) of the Fractional Register element

Our main design problem is now to generate Hold signals exactly when needed to respect the previous constraints. In addition it should be generated from the schedules of the source and target computation or transport nodes, to bridge from the former to the latter. Consider again figure 3, we shall name w the schedule of the previous source node, and w the schedule of the next target node. After the regular register delay the tokens are produce to the FR entry on schedule 0.w (shifted one slot/instant right). The fractional buffer should hold the token exactly when the kth active step at this entry is not the kth activity step at its target node that must consume it. In other words the FR element resynchronize its input and output. Stated formally, this property becomes:
$HOLD(n) = 1$ IF F $|0.w_n|_1 != (|w'_n|_1 - |w'_0|_1)$.
It says that at a given instant n we should kidnap a value if the number of occurrences of 1 up to instant n on the previous node is different than the number of occurrences of 1 on the next computation node. Figure 5 shows a possible implementation computing Hold from signals that would explicit provide the target and source schedules as inputs.

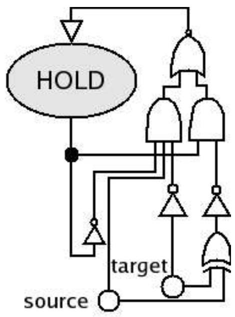

figure 5: HOLD signal circuitry

Tool development

We built a prototype tool named KPASSA (which stands for **K**-**P**eriodic **A**sap **S**chedule **S**imulation and **A**nalysis.) to simulate and analyze systems

like LID made of a combination of previous components.
Our KPASSA tool implements the various algorithmic stages described above. It computes and displays the system throughput, shows critical cycle loops and the locations of choice for extra integer latency insertions in non-critical cycles. It then computes an explicit schedule for each computation and transportation node (in the future it could be helpful to display only the important ones), and provides locations for fractional registers insertion. It also provides log information on the numbers of elements added, and whether perfect integer equalization was achieved in the early steps.

Tables 1 and 2 display benchmark results obtained with KPASSA. The first examples were obtained by assigning random latencies inside a given range to existing block diagram specifications. The last two examples are artificial large models meant to test the limits of algorithmic complexity. Table 1 records various size and characteristics relevant to the complexity of our algorithms. Table 2 reports some of the results obtained: whether perfect equalization holds; the number of fractional registers that should be required in the initial and periodic phases (note that FR elements may be needed in perfectly equalized cases, when the system is not strongly connected); the number of integer latencies added; performance in time and space consumption.

Conclusion and Further Topics

Concerning the static scheduling, a number of important topics are left open for further theoretical developments:

- Relaxing the firing rule: So far the theory developed here only consider the case where local synchronous components all consume and produce data on all input and output channels in each computation step, and where they all run on the same clock. So it can be proved that the relaxed-synchronous version produces the same output streams from the same input streams as the fully synchronous specification Several papers considered extensions in the context of GALS systems, but then ignored the issue of functional correspondence with an initial well-clocked specification, which is our important correctness criterion. This relaxation may help to minimize some metrics :
    - Discovering short and efficient (minimizing number of FR elements) initial phases is also an important issue here.
    - The distribution of integer latencies over the arcs could attempt to minimize (on average) the number of computation nodes that are active altogether. In other words transportation latencies should be balanced so that computations alternate in time whenever possible. The goal is here to avoid "hot spots" that is to say flatten the power peaks. It could be achieved by some sort of retiming/ recycling techniques and schedules

	#Nodes	#Cycles	#Critical Cycles	Max Cycle Latency	Throughput
MPEG2 Video Encoder	16	7	3	21	3/7
Encoder MultiStandard ADPCM	12	23	23	14	1/2
H264/AVC Encoder	20	12	3	27	4/9
29116a 16bits CAST MicroCPU	11	7	3	35	3/35
Abstract Stress Cycles	40	2295	1	1054	4/29
Abstract Stress Nodes	175	3784	1	1902	4/29

Table 1: Example sizes before equalization

	Perfect Eqn.	#FR init/periodic	#Added latencies	Time	Memory
MPEG2 Video Encoder	N	9/5	18	<1 sec	˜11MB
Encoder MultiStandard ADPCM	Y	24/0	91	<1 sec	˜11MB
H264/AVC Encoder	N	18/11	0	˜ 1sec	˜11MB
29116a 16bits CAST MicroCPU	Y	0/0	0	˜ 1sec	˜11MB
Abstract Stress Cycles	N	55/24	1577	˜17 sec	˜16MB
Abstract Stress Nodes	N	59/23	2688	˜4 min	˜43MB

Table 2: Equalization performances and results (Run on P4 3.4Ghz, 1GB RAM , Linux 2.6 and JDK 1.5)

exploration still using a relaxed firing rule;

- Marked graphs do not allow for control-flow alternatives and control modes. One reason is that, in a generalized setting such as full Petri Nets, it can no longer be asserted that token are consumed and produced at the same rate. But explicit "branch schedules" could maybe help regulate the branching control parts similarly to the way they control the flow rate;

Finally, the goal would be to define a general GALS modeling framework, where GALS components could be put in GALS networks (to this day the framework is not compositional in the sense that local components need to be synchronous). A system would consists again of computation and interconnect communication blocks, this time each with appropriate triggering clocks, and of a scheduler providing the subclocks computation mechanism, based on their outer main clock and several signals carrying information on control flow.

References

[1] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone. Another glance at relay stations in latency-insensitive designs. In FM-GALS'05, 2005.

[2] Mario R. Casu and Luca Macchiarulo. Floor-planning for throughput. In ISPD '04: Proceedings of the 2004 international symposium on Physical design, pages 62-69, New York, NY, USA, 2004. ACM Press.

[3] Luca P.Carloni, Kenneth L.McMillan, and Alberto L.Sangiovanni-Vincentelli. Theory of latency-insensitive design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2001.

[4] Luca P.Carloni, Kenneth L.McMillan, Alexander Saldanha, and Alberto L.Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In THE BEST OF ICAD, 200x.

[5] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks. In POPL 2006 Proceedings, January 2006.

[6] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked directed graph. Journal of Computer and System Sciences, 5:511 523, october 1971.

[7] J. Carlier Ph. Chretienne. Probleme d'ordonnancement: modelisation, complexite algorithmes. Masson, Paris, 1988.

[8] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. Synchronization and Linearity. Wiley, 1992.

About the Authors

Julien Boucaron is a second-year PhD student in the Aoste team at INRIA on a scholarship with STMicroelectronics Project ForComent. He holds a master degree from the university of Nice/Sophia-Antipolis.

Jean-Vivien Millo is also (first-year) PhD student in the Aoste team at INRIA, on a scholarship with STMicroelectronics in the framework of the CIM PACA Design Platform regional collaboration initiative. He holds an engineering degree from ESIGETEL and a master degree from university of Marne la Vallee.

Robert de Simone is the head of INRIA Aoste team, and a researcher at INRIA since 1985. His former research interests include Concurrency Theory and automatic Model-Checking, as well as synchronous languages such as the Esterel/SyncCharts formalisms; he is currently involved in Globally-Asynchronous/Locally synchronous Models of Computation.