A New System Design Methodology for Wire Pipelined SoC

Mario R. Casu
Politecnico di Torino, Italy
mario.casu@polito.it

Luca Macchiarulo
University of Hawaii, HI
lucam@hawaii.edu

Abstract

Wire Pipelining (WP) has been proposed in order to limit the impact of increasing wire delays. In general, the added pipeline elements alters the system such that architectural changes are needed to preserve functionality. We illustrate a proposal that, while allowing the use of IP blocks without modification, takes advantage of a minimal knowledge of the IP's communication profile to dramatically increase the performances. We showed the formal equivalence between WP and original system and proved the higher performance achievable through a relevant case study.

1. Theory and Implementation

A *signal* is a set of *events* e, i.e. couples $e = (v, t)$ of *values* v and *tags* t. In our sequential systems, tags can be thought as clock ticks. Processes exchange signals by means of *channels*. If we add wire pipeline elements to channels, these initially hold void data that we denote using τ symbol. The behavior is modified such that sequences of *valid* events, are interrupted by void values. A generic realization of a channel in time interval $[t_1, t_N]$ is $(v_1, t_1), \tau, \tau, (v_2, t_2), \ldots, \tau \ldots$ Let us filter out *void* symbols τ and find the maximum tag N such that every signal has a sequence of at least N values. If all sequences are identical to the original system from 1 to N, the two systems, with and without WP, are *N-equivalent*. If they are equivalent $\forall N$, they are said *equivalent*. A wrapper must enclose the processes in order to guarantee equivalence and perform the following tasks:
1) τ-filtered inputs are buffered with *semi-infinite* fifos and stored in positions progressively numbered with the tags.
2) a synchronizer keeps trace of the current tag and, a) when all identically tagged inputs are available (synchronous signals) dispatches them to the internal process, removing them from the fifos; b) if at least one input does not have the current tag, the process is *stalled*.
3) In correspondence with the stall, τ is sent to all outputs.

This wrapper could be simplified so that all valid signals ($\neq \tau$) already processed (and so eliminated from the fifo) are *counted*. The synchronizer's task is to trigger the process if all inputs with tag equal to the counter are present.

If we suppose that some processes are such that, for some of their internal states, *not all inputs are simultaneously read*, then it is possible to advance the computation even before all the corresponding inputs have reached it (relaxation of synchronicity). This is the key for an overall increase of performance. An analogous observation is in [1] without proof of equivalence. We introduce a new element in our wrapper: an **oracle** that decides which inputs are needed for the next computation. The new wrapper's features are:
1) if all inputs *required by the oracle* are present, the computation is triggered and the fifo updated.
2) The synchronizer discards all inputs whose tag is smaller than the counter value ("old" tags).

Synchronicity is guaranteed by the fact that the fifo actually discards valid non-necessary inputs which became useless due to the process' blindness to them.

The step to make the entire consideration practical is making the fifo length finite. Thus, *back-pressure* has to be implemented with a *stop* signal propagated back to the wire pipeline elements that, in order not to lose data, should contain, together with the pipeline register, at least one auxiliary register for saving a valid incoming datum when a *stop* is received. A simple FSM already described in [2] and called *relay station* (RS) suffices to guarantee correct functionality. When also the auxiliary register is full, the *stop* is propagated to the previous RS up to the source process.

Instead of counting the absolute tag, it is sufficient to use an initialized counter that records the lag between the most recent and old data received by each channel. From the practical standpoint, it is also not necessary to send the tag together with the signal, but only a bit indicating its validity [2], as the distributed counters will take care of the tag values, thanks to the ordering properties of the signals.

Wrappers with and without the additional *oracle* that uses the processing information were described in VHDL and simulated. We evaluated the wrappers' area with several synthesis experiments on a 130 nm technology. The overhead was always less than 1% with respect to an IP of

1530-1591/05 $20.00 © 2005 IEEE

100 kgates. As for the timing of the logic used within the wrappers, it was never critical in all our experiments.

2. Case Study

We considered a processor made out of five components to be enclosed within our wrappers and with pipelined connections as shown in fig. 1. The responsible of performance pitfalls are the netlist loops. Let us define the throughput Th as the average number of valid data produced in every clock cycle. A loop containing m process and n pipeline delays has Th=$\frac{m}{m+n}$ in case wrappers do not implement *oracles*. The worst loop dominates the system Th. The netlist loops are shown in fig. 1. We built the system with a minimal instruction set and used two basic programs to cover the spectrum of applications: a strictly data dependent problem, *extraction sort*, and a *matrix multiplication*. The processor was modelled with a RTL VHDL description and implemented in two different fashions, **multicycle** and **pipelined.**

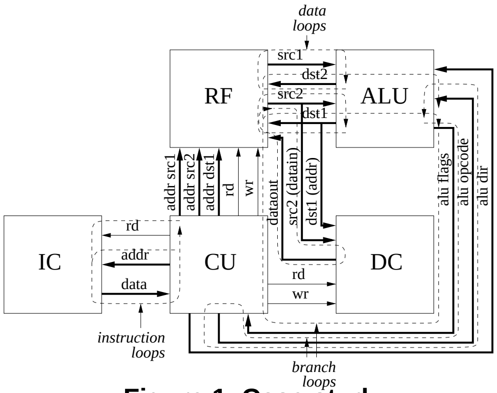

Figure 1. Case study.

The migration to a standard WP system, WP1 in our experiments, is straightforward: blocks are encapsulated by wrappers and wires segmented by relay stations. In the case of our new technique, WP2, the input signals are complemented, when possible, by a *processing signal* derived from the process operation. However, in our case (which we think representative of practical cases), the effort was minimal.

3. Experimental Results

We ran the following experiments for both multiprocessor and pipelined case but for space reasons only the latter case results are reported in table 1:
1. The golden system is simulated. The number of clock periods is used for the computation of the WP's performance. The throughput without WP is of course 1.0.
2. We added single relay stations (RS) for each and every connection in the system's topology (rows 2-11).
3. We built a system with 1 RS in each connection (row 12).
4. For the Matrix Multiply only, we repeated the simulations for at least 1 RS everywhere and 2 somewhere (rows 13-22) and 2 RS everywhere but in some links (rows 23-25).

We would like to draw some overall conclusions:
1. All results are in favor of the proposed WP2 system.
2. The advantage depends on the features of the communication channel at stake: In the Multicycle case, not reported in table, the CU-IC loop is excited only every 5 cycles due to the non pipelined sequence of 5 phases Instruction Fetch, Decode and contextual Operand Fetch, Execution, Memory access and Write-back. That's the reason of the best improvement of WP2 in this loop with respect to WP1 (60%) where such loop is *statically* present at all time. Other channels accessed more frequently give less advantage.
1. Even if the computations are tighter in the pipelined case (in the same clock cycle different loops are exercised at the same time) we still observe the relevant advantages of WP2.

			Th	Th	WP2 vs.
	RS Configuration	Cycles	WP1	WP2	WP1 (%)
Extraction Sort					
1	All 0 (ideal)	1559	1.0	1.0	0%
2	Only CU-RF	2078	0.75	0.75	+0%
3	Only CU-AL	2090	0.667	0.75	+13%
4	Only CU-DC	2078	0.75	0.75	+0%
5	Only CU-IC	3118	0.5	0.5	0%
6	Only RF-ALU	1886	0.667	0.83	+25%
7	Only RF-DC	1569	0.667	0.99	+49%
8	Only ALU-CU	1681	0.667	0.93	+40%
9	Only ALU-RF	1701	0.667	0.92	+38%
10	Only ALU-DC	1634	0.667	0.96	+44%
11	Only DC-RF	1624	0.667	0.96	+44%
12	All 1 (no CU-IC)	2325	0.5	0.67	+34%
13	Optimal 1 (no CU-IC)	1952	0.667	0.80	+20%
Matrix Multiply					
1	All 0 (ideal)	2778	1.0	1.0	0%
2	Only CU-RF	3704	0.75	0.75	+0%
3	Only CU-AL	3724	0.667	0.75	+13%
4	Only CU-DC	3704	0.75	0.75	+0%
5	Only CU-IC	5557	0.5	0.5	0%
6	Only RF-ALU	3596	0.667	0.77	+16%
7	Only RF-DC	2824	0.667	0.98	+47%
8	Only ALU-CU	2851	0.667	0.97	+46%
9	Only ALU-RF	3436	0.667	0.81	+22%
10	Only ALU-DC	3058	0.667	0.91	+37%
11	Only DC-RF	2994	0.667	0.93	+40%
12	All 1 (no CU-IC)	4703	0.5	0.59	+18%
13	All 1 and 2 CU-RF	4775	0.5	0.58	+16%
14	All 1 and 2 CU-AL	4703	0.4	0.59	+48%
15	All 1 and 2 CU-DC	4703	0.5	0.59	+18%
16	All 1 and 2 CU-IC	8335	0.33	0.33	0%
17	All 1 and 2 RF-ALU	5521	0.4	0.50	+25%
18	All 1 and 2 RF-DC	4703	0.4	0.59	+48%
19	All 1 and 2 ALU-CU	4776	0.4	0.58	+45%
20	All 1 and 2 ALU-RF	5235	0.4	0.53	+33%
21	All 1 and 2 ALU-DC	4919	0.4	0.56	+40%
22	All 1 and 2 DC-RF	4919	0.4	0.56	+40%
23	Optimal 2 (no CU-IC)	4919	0.4	0.56	+40%
24	All 2 (no CU-IC)	6555	0.33	0.42	+26%
25	All 2 and 1 CU-RF	6555	0.33	0.42	+26%

Table 1. Sort: Pipelined Case.

References

[1] M. Singh and M. Theobald, Generalized Latency Insensitive Systems for Single-Clock and Multi-Clock Architectures, *Proc. DATE 2004,* Paris.

[2] L.P Carloni *et al.,* A Methodology for Correct by Construction Latency Insensitive Design", ICCAD 99, pp. 309-315.

[3] L.P. Carloni *et al.* Theory of Latency-Insensitive Design, IEEE TCAD, vol. 20, No. 9, Sept. 2001, pp. 1059-1076.