Adaptive Latency-Insensitive Protocols

Mario R. Casu
Politecnico di Torino

Luca Macchiarulo
University of Hawaii

Editor's note
Latency-insensitive protocols (LIPs) represent a class of interblock protocols designed to overcome long multiclock interconnects. This article presents an adaptive solution to this problem, which the authors show to be more effective than earlier solutions in terms of power, area, and throughput.
—Sandeep Shukla, Virginia Tech

FOR DECADES, LOGIC designers have used the technique of breaking up slow and deep combinational networks with pipeline stages to raise clock frequencies. Breaking up long and slow on-chip interconnects in the same way is a more recent technique, at least on a pervasive basis. The reason is that wire delays increase as geometries shrink, whereas gate delays continue to decrease. The problem is particularly serious for global wires connecting blocks of a relevant complexity (a few kilogates or more). Interestingly, the 2005 *International Technology Roadmap for Semiconductors* (http://www.itrs.net/Links/2005ITRS/Home2005.htm) sets the clock period of high-performance processors starting in 2007 at about 12 FO4 delays (FO4 is the delay of a CMOS inverter loaded with four identical inverters), as if the clock period were defined only by logic levels. Wire delays are excluded from critical paths because global-interconnect pipelining will mitigate their impact.

Adding pipeline stages in logic gates as well as wires cures a design's bandwidth problems at a price: a latency increment of one clock cycle for each added pipeline stage. Latency often reduces performance, and systems with wire pipelining are no exception. In microprocessors, a pipeline stage stalls when a data dependency occurs between two instructions. In this context, loop topology connections of logic blocks can induce data dependency. Pipeline stages added to the wires forming the loop delay the arrival of the data that each logic block uses to compute new data, resulting in pipeline stalls and performance reduction. Loop topologies are more common than you might expect. If two blocks connect in such a way that the output of one is the input of the other and vice versa, the loop forms readily, as with a microprocessor-cache link.

Because latency added in wires can be harmful, researchers have tried to cope with this problem in different ways. On the one hand, it is important that SoCs, especially those using (and possibly reusing) predefined IP blocks, can tolerate an amount of latency that couldn't be predicted at design time. For this reason, the concept of latency-insensitive design emerged. On the other hand, designers can modify the classic physical design steps of floorplanning, placement, and routing to include wire latency in their optimization target. In the past, we contributed to both high-level and physical design,[1–3] but here we concentrate on the former.

Latency-insensitive design copes with excessive delays typical of global wires in current and future IC technologies. It achieves its goal via encapsulation of synchronous logic blocks in wrappers that communicate through a latency-insensitive protocol (LIP) and pipelined interconnects. Previously proposed solutions suffer from an excessive performance penalty in terms of throughput or from a lack of generality (see the ''Related work'' sidebar). This article presents an adaptive LIP that outperforms previous static implementations, as demonstrated by two relevant cases—a microprocessor and an MPEG encoder—whose components we made insensitive to the latencies of their interconnections through a newly developed wrapper. We also present an informal exposition of the theoretical basis of adaptive LIPs, as well as implementation details.

0740-7475/07/$25.00 © 2007 IEEE Copublished by the IEEE CS and the IEEE CASS

Related work

The problem of increasing wire delays with decreasing gate delays is not new, but it gained pivotal importance when clock frequencies reached such high values that signals could no longer cover the maximum on-chip distance (that is, corner to corner) in a clock period. We acknowledge the seminal work of Carloni, McMillan, and Sangiovanni-Vincentelli, who first proposed a solution to make SoCs insensitive to latencies caused by wire pipelining.[1] Researchers have proposed variants of their latency-insensitive protocol (LIP), ranging from simplified, yet correct, performance-equivalent solutions[2] to significantly modified techniques aimed at improving performance.[3,4]

Increasing wire delays and clock frequencies are also the reason why distributing a centralized clock throughout a chip with an acceptably low skew is becoming prohibitive. Globally asynchronous, locally synchronous (GALS) approaches seem appropriate in such cases. GALS approaches perform local computation in a classic synchronous way but limit it to blocks of reasonable size and perform global communication between blocks using asynchronous paradigms. Latency-insensitive design has features in common with GALS approaches. Researchers have attempted to use the best of both worlds, by pipelining asynchronous global wires through mixed-clock FIFO buffers[5] and by using synchronous-to-asynchronous interfaces at the blocks' inputs and outputs.[4]

More radical approaches aim at taking advantage of regular on-chip fabrics based on networked connections and on-chip routers (networks on chips). In such cases, point-to-point connections between routers must be latency insensitive to cope with excessive wire delays.[6] Recent research, inspired by the original idea of latency insensitivity, applies this concept to the logic design of blocks, making them elastically flexible to external latencies, whether coming from wires or logic.[7]

The process of latency desensitization of a previously working implementation (although performance limited by slow wires) must not modify the system's logic behavior. Formal, mathematically sound studies have shown how to guarantee equivalence with an appropriate LIP[1] and how to validate protocol correctness.[8] Concerning performance modeling, Lu and Koh have analyzed the upper bound achievable by Carloni, McMillan, and Sangiovanni-Vincentelli's LIP, using a netlist graph and max-plus algebra.[9]

LIP performance depends on the amount of latency in wires, and this value is known only after layout. It is important to predict the latency in advance, prior to back-end design stages, as well as to make layout tools aware of global wire latencies, as we showed for a CAD floorplanning tool.[10,11] Other works on microarchitectural floorplanning aim at reducing the performance impact of interconnect latencies evaluated in cycles per instruction.[12]

References

1. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli, ''Theory of Latency-Insensitive Design,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, Sept. 2001, pp. 1059-1076.
2. M.R. Casu and L. Macchiarulo, ''A New Approach to Latency Insensitive Design,'' *Proc. 41st Design Automation Conf.* (DAC 04), ACM Press, 2004, pp. 576-581.
3. M. Singh and M. Theobald, ''Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 04), IEEE CS Press, 2004, vol. 2, pp. 1008-1013.
4. A. Agiwal and M. Singh, ''An Architecture and a Wrapper Synthesis Approach for Multi-Clock Latency-Insensitive Systems,'' *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 05), IEEE CS Press, pp. 1006-1013.
5. T. Chelcea and S.M. Nowick, ''Robust Interfaces for Mixed-Timing Systems,'' *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 8, Aug. 04, pp. 857-873.
6. M. Dall'Osso et al., ''Xpipes: A Latency Insensitive Parameterized Network-on-Chip Architecture for Multi-Processor SoCs,'' *Proc. 21st Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 03), IEEE CS Press, 2003, pp. 536-539.
7. J. Cortadella, M. Kishinevsky, and B. Grundmann, ''Synthesis of Synchronous Elastic Architectures,'' *Proc. 43rd Design Automation Conf.* (DAC 06), ACM Press, 2006, pp. 657-662.
8. S. Suhaib et al., ''Validating Families of Latency Insensitive Protocols,'' *IEEE Trans. Computers*, vol. 55, no. 11, Nov. 2006, pp. 1391-1401.
9. R. Lu and C.-K. Koh, "Performance Analysis of Latency-Insensitive Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, Mar. 2006, pp. 469-483.
10. M.R. Casu and L. Macchiarulo, ''Throughput-Driven Floorplanning with Wire Pipelining,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, May 2005, pp. 663-675.
11. M.R. Casu and L. Macchiarulo, ''Floorplanning with Wire Pipelining in Adaptive Communication Channels,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, Dec. 2006, pp. 2996-3004.
12. M. Ekpanyapong et al., ''Profile-Guided Microarchitectural Floorplanning for Deep Submicron Processor Design,'' *Proc. 41st Design Automation Conf.* (DAC 04), ACM Press, 2004, pp. 634-639.

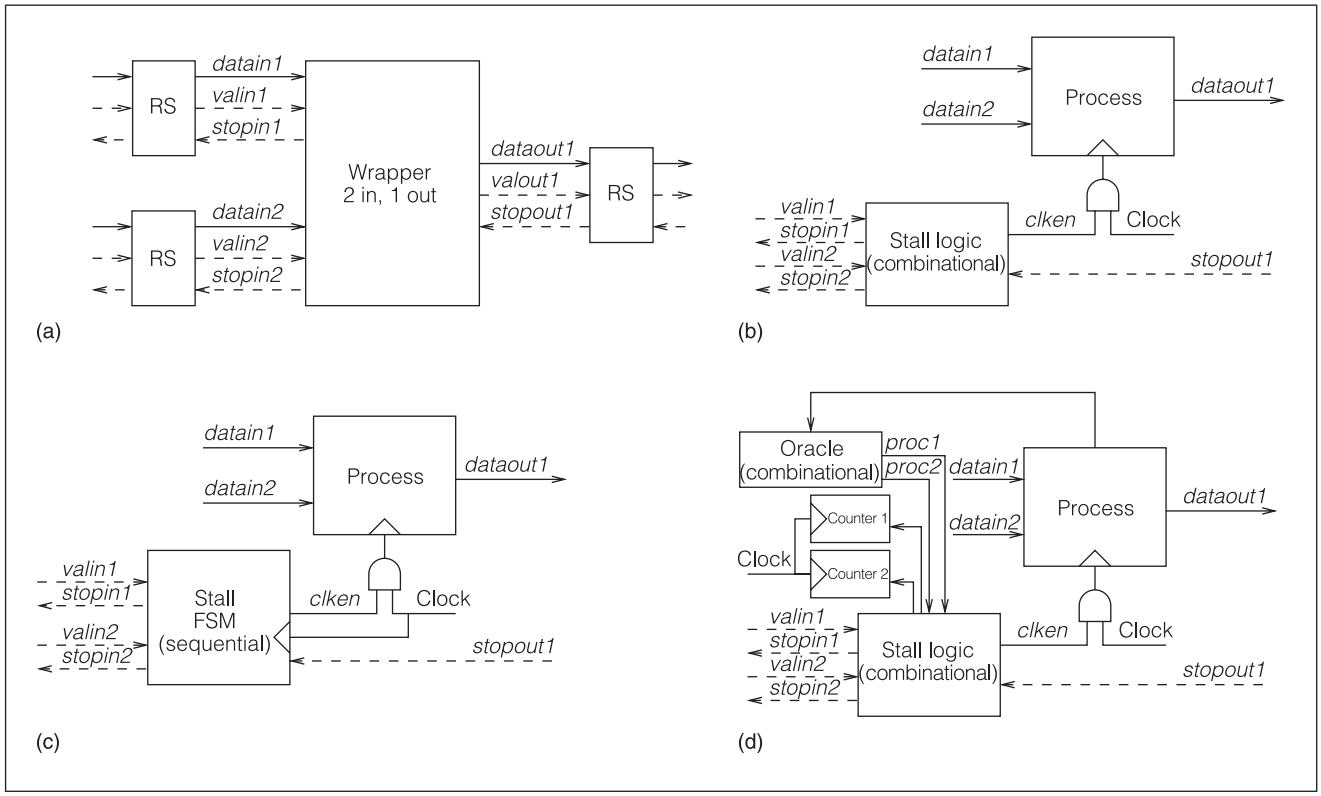

Figure 1. A 2-in, 1-out wrapper communicating with relay stations (RS) through valid and stop protocol signals (a), stall logic for static latency-insensitive protocol (LIP) wrapper (b), generalized LIP wrapper[4] (c), and block scheme of the adaptive LIP wrapper described in this article (d).

Static versus adaptive LIPs

Latency-insensitive design builds on a reasonable assumption: We can make a synchronous process insensitive to wire latencies provided it is *stallable*. That is, it must be possible to pause the process' clock when at least one input is not available for the next computation. This assumption means that we can enclose the synchronous process within a wrapper that gates the clock, depending on input availability.

To preserve the correct operation sequence and guarantee reliable communication, we must set up a handshake mechanism called a latency-insensitive protocol. According to this protocol, every block's I/O signal is associated with a binary validity tag whose value indicates availability. An invalid tag on at least one input inhibits computation through clock gating and invalidates all of the block's output tags. The wrapper also asserts a *stop* signal associated with every input during stalled cycles to avoid loss of valid data. To avoid data overrun, a wrapper that receives on one of its outputs a stop tag generated by another wrapper stalls the controlled block.

The pipeline elements inserted in the links connecting the processes comply with the LIP and consist of simple FIFO buffers with at least two places: one to pipeline a datum and tag, and the other to store an incoming new datum and tag on a stop event. When full, the buffers propagate a back-pressure signal upward. The buffers are called relay stations, and in the original works on LIP were synchronous to both sender and receiver. Using this approach, we can derive the number of relay stations placed along a multicycle wire, because timing constraints set the maximum delay between modules. In the mixed-clock relay stations used in globally asynchronous, locally synchronous (GALS) systems, the number of stations in a wire and the internal buffer's size depend on many factors, including the need to reduce the occurrence of metastability, the difference between production and consumption rates, and the area occupation.

Figure 1a and Figure 1b show a 2-in, 1-out wrapper communicating through relay stations and the internal stall logic for the controlled process. In our implementation, we can replace the input stations with

queues (to allow a simpler pipelined interconnect) that are functionally Moore-type finite-state machines (FSMs). By breaking the direct connection of the stall logic blocks, the queues avoid combinational loops that could arise from the composition of two or more wrappers.

This informal description might erroneously imply that the sequence of stalling and firing events depends on the amount of traffic on the channels connecting the processes and on the rate of data production and consumption. However, we can show that the system's behavior is static. We calculate the system's throughput, evaluated as the average number of unstalled computations per clock cycle, as the worst ratio $m/(m+n)$ of the netlist graph, where m is the number of blocks in a graph loop, and n is the number of latencies along that loop's edges.[5] This static behavior is the key to reducing the protocol's overhead by instrumenting the wrappers to stall processes according to a pseudoperiodic, statically computed schedule.[1]

This apparently anomalous property of LIPs derives from the assumption that a single invalid input can stall the next computation, even though that particular input is not needed in that process state. As a result, many avoidable stalling events reduce the throughput to well below 1.0, the ideal value if no such events occur. Performance limitation stems from the same principle that makes the LIP attractive: The wrappers work with no knowledge of how modules use the exchanged messages. Thus, static LIPs are perhaps the most perfect and elegant archetype of complete orthogonalization between computation and communication. However, elegance and performance don't go together in this case, as the throughput formula shows. Suppose two blocks are connected in a loop ($m = 2$) with one latency per channel ($n = 2$). The system's throughput is $2/4 = 1/2$ due to valid computations alternating with stalling events caused by wire latencies. To have the same or higher data rate than the system that doesn't use wrappers and relay stations, we must at least double the clock frequency.

Fortunately, a modification of the wrappers is possible that allows performance gain while retaining most of the original LIP philosophy. As a simple example, consider a two-way multiplexer that alternatively reads one of the two inputs. If the wrapper knew the one selected, it could discard invalid data on the other to avoid useless stalls. Researchers proposed a modified protocol they called *generalized LIP* to express the fact that stalling events will no longer be associated with any possible invalid data on the input set but rather with a subset that the wrapper elects as needed for the computation.[4,6] Their wrapper includes a Mealy FSM that gates the clock only on relevant stalling events and selectively exerts back pressure. Figure 1c shows this type of wrapper, as compared with the original static version in Figure 1b. The FSM is fully specified, starting from a block's interface description in a particular hardware specification language.

In the generalized LIP, it is not possible to assign an overall system throughput based on topological features as it was in the static protocol. The instantaneous throughput, which in the previous case was coincident with the average throughput, now depends on the traffic pattern. Instead of generalized LIPs, we prefer the notion of *adaptive LIPs*, as opposed to static LIPs. We think "adaptive LIP" better captures the variability of traffic shape in channels, the fact that the sequence of stalling events changes accordingly, and the consequent adaptation of the instantaneous throughput.

The key problem of such methodologies is to guarantee system safety by not discarding relevant data. Suppose we assign each block a local time incremented by one at every valid computation, so that it records the number of enabled clock ticks since inception (or reset). If there are no stop events, the local time is the same for all system blocks and coincides with the clock cycle count (in a fully synchronous system). In case of stalling events, the blocks' local times might be misaligned. Suppose also that we associate this local time with valid computed data (we call it virtual time), so as to mark the data with the time of production (as we shall show, it is possible to get rid of both local and virtual time counters and signals in the actual implementation). The static protocol forces all data at a block's input to be coherent; in other words, computation is enabled when all inputs have the same virtual time. This is not true for the adaptive case, but it is true that *valid data on unprocessed inputs can be safely discarded if they were produced at a time equal to or earlier than the block's local time*. As a corollary, if valid data has been produced later than the local time, the wrapper has no right to refuse this data and must exert back pressure to stop the data until the local time becomes synchronized with the input virtual time.

On the basis of this observation, we derived a new adaptive wrapper that uses counters to keep track of the possible misalignment between processed and

unprocessed inputs. In addition, an oracle elaborates basic information taken from the wrapped block to select the necessary inputs in a given computation. To understand how the wrapper works, consider a block with two inputs, *in-1* and *in-2*, associated with two counters. When both inputs are processed, computation is enabled if their virtual times are the same, a fact represented by both counters' being zero. On the other hand, if the oracle selects, say, *in-1* and not *in-2*, then *in-2* can misalign. The counter of *in-1* remains at zero, while the other keeps track of the misalignment. In particular, every time new valid data on *in-1* is consumed while the unnecessary data on *in-2* is not valid, the counter of *in-2* is incremented. If valid *in-2* data is received in the absence of new valid data on *in-1*, then the counter of *in-2* is decremented and eventually brought to 0. Finally, if this happens with aligned data, the second counter is decremented to −1, and at the same time a back-pressure signal is emitted: The counters cannot get lower than −1; otherwise, the wrapper would discard data on an unprocessed input that was produced later than the local time. In this case, computation stalls until the inputs realign.

The example shows that counters don't store actual times, nor do such times need to be transmitted alongside data, because counting the number of validity bits is equivalent to keeping track of their times. Since the counters actually record the difference between virtual times, using increment and decrement signals, they don't need to store large values. We can assess their relatively small size by carefully considering the communication profiles. However, if the counters reach the maximum count because of a temporary excessive delay between inputs that could not be predicted, the back-pressure signal is asserted. Because bigger counters can reduce the number of such stall events, we can trade their area as well as their power for performance. Absolute local and virtual times are not used, so no local time counter (to compare virtual times with) is required.

Figure 1d shows that the counters (counter 1 and counter 2 in the two-input example) are controlled by the stall logic, which has the usual protocol inputs as well as binary processing signals (*proc1* and *proc2*). The latter indicate the need for input data (*datain1* and *datain2*) and are forecast by the oracle, which takes the necessary information from the logic process (oracle input in the figure).

In FSM-style implementations and in our approach, the impossibility of extracting the necessary information does not prevent the wrapper from working properly but leads instead to standard LIP behavior. By removing counters and processing signals, the wrapper's logic in Figure 1d boils down to the static logic of Figure 1b. However, the FSM approach is less general than ours in that it cannot capture data-dependent behaviors.[4,6] The only inputs to the FSM are the *valid* and *stop* protocol signals. Therefore, the wrapper can map a subset of possible states only if the input selection is perfectly known in advance. In contrast, because our wrapper takes the information directly from the process, it can closely follow the behavior of that process.

A key issue in the adaptive mechanism is the determination of effective and simple oracles: There are various ways to perform this task, such as using limited knowledge of interface semantics, communication patterns, and high- and low-level extraction. Details of these techniques are beyond the scope of this article, but we touch on this issue later in the article, when we present two case studies.

Detailed block design: static and adaptive

In describing the detailed implementation of the protocol, we concentrate on analyzing the wrapper, also called the *shell*. The shell contains a *pearl*, which consists of the original implementation's functional block, a clock-gating circuit, a simple sequential circuit that validates output data, and a combinational network that generates the back-pressure *stop* signal.

Shells and encapsulation: static protocol

Figure 2a shows the output validation, back propagation, and clock-gating circuits for a 2-in, 3-out shell. The block produces valid data if the pearl is not gated or if a previous valid datum was stopped (signal $stopout_i = 1$). Stop signals are back-propagated on input k if its input datum is valid ($valin_k = 1$) and the shell is gated ($clken = 0$). Stopping invalid data is useless. The pearl's clock is enabled only when all inputs are valid and when valid output data is not being stopped. For the case shown in Figure 2a, the enable signal is

$$clken = (valin_1 \cdot valin_2) \prod_{i=1}^{3} \overline{(valout_i \cdot stopout_i)} \quad (1)$$

where Π stands for the logic product.

Equation 1 shows that regardless of the state of the outputs, the block stops if any input contains invalid

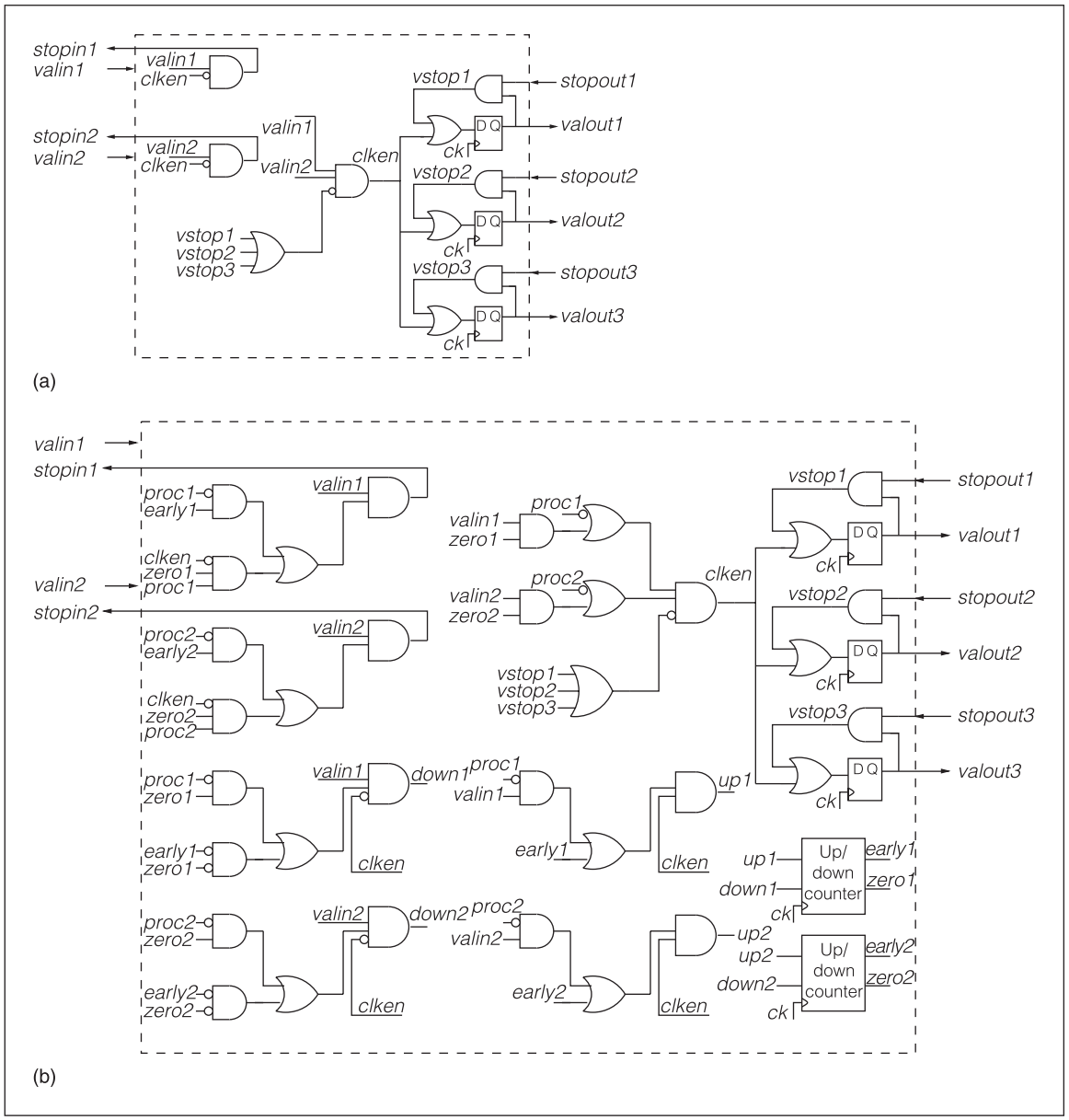

Figure 2. Static (a) and adaptive (b) shell circuit examples.

data, no matter what the actual need is for such an input. This is the key feature distinguishing static from adaptive protocols.

Implementing the adaptive protocol

To introduce adaptive behavior in the previous scheme, keeping track of time tags to enforce data coherency is important. Fortunately, as explained earlier, there's no need to communicate the entire virtual time along with the data. We simply need counters, one for each input, to record the relative distance between the most recently received inputs.

Two flags derived from the counters contain the information necessary for the protocol's functioning:

- *zero*—this flag is 1 whenever the relative input is synchronous with the shell's local time; and
- *early*—this flag is 1 whenever the relative input is one clock cycle early (which can happen only when the last computation didn't need to process that input).

When neither flag is 1, the counter stores the misalignment, in terms of virtual times, between the

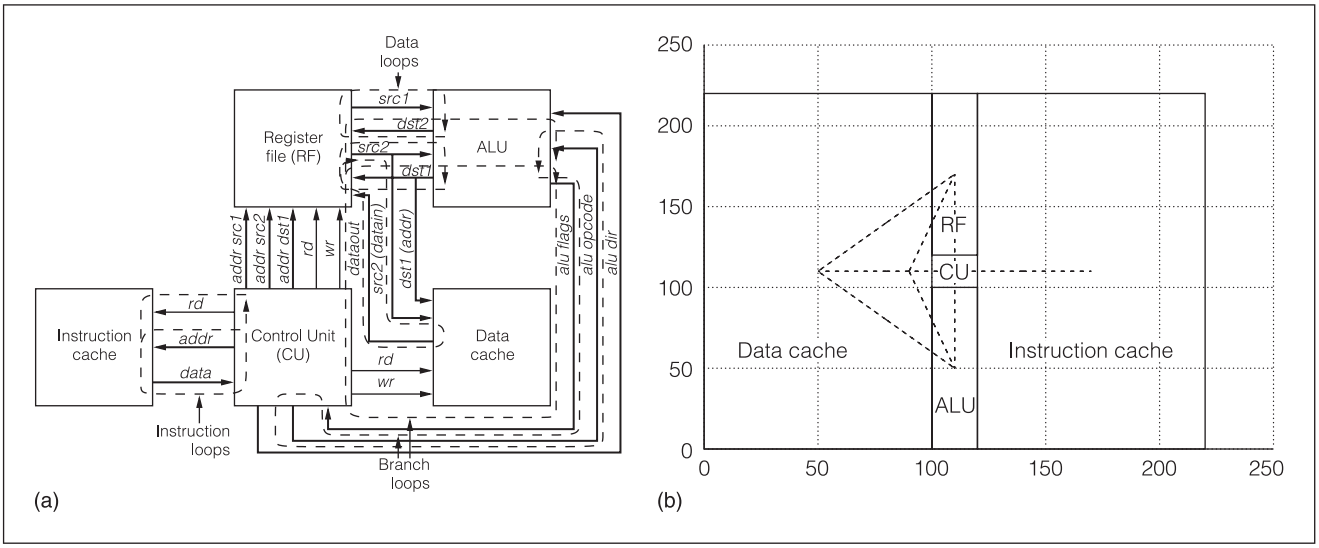

Figure 3. Case study: standard processor with constraining loops (a) and its optimized floorplan (b). Dashed lines in (b) represent connections between processor blocks. (ALU: arithmetic logic unit.)

processed inputs and the last discarded valid datum on the unprocessed input.

It is important that we allow an *early* condition on a single clock only: When the block is in the process of elaborating its *m*th output, there's no way of knowing which inputs it will need for the (*m* + 1)th output.

Figure 2b shows a schematic diagram of the modified shell for a 2-in, 3-out block. Note that the clock-gating, back-pressure, and validation circuits simplify to the static shell if processing bits are always 1 and, consequently, *zero* signals are always 1, and *early* signals are always 0.

We control the up/down counter as follows:

- *Up count*. The block is active (*clken* = 1), and the input is either valid but not needed or early.
- *Down count*. The input is valid and the block is gated (*clken* = 0), and either the counter is positive (we are waiting for old discarded signals, neither *early* nor *zero*) or we have an unprocessed input with a *zero* count (this input can be discarded: *zero* and not *proc*).

Other than the case in which the counters reach the maximum value allowed by their finite size, the back-pressure signals are also asserted when the related input is valid, and either the signal is anticipated (*early* = 1) or is synchronous (*zero* = 1) while the pearl is gated. Finally, the clock enable signal (*clken*) lets the computation proceed whenever all outputs are not actively stopped (*vstop* = 0), and all inputs are either valid and synchronous (*zero* = 1) or unnecessary (*proc* = 0).

We implemented the model described here as an RTL VHDL block. We used it to perform all the experiments reported in the following section.

Case studies

To validate the functionality of our adaptive LIP, we implemented two systems which we chose as representatives of two extremes: a simple microprocessor (whose communication profile is extremely data dependent and related to the executed code) and an MPEG encoder (which presents a relatively uniform, burst type of communication).

Microprocessor

We described in VHDL code an extremely simplified, five-stage pipelined processor. Figure 3a shows its schematic. The five functional units—control unit (CU), register file (RF), instruction cache (IC), data cache (DC), and arithmetic logic unit (ALU)—are part of strict communication loops that are likely to reduce performance drastically in a static LIP implementation:

- *Data loops*. Operands move from the register file to the ALU and are stored back in the register file.
- *Branch loops*. Flags from the ALU move to the control unit and back to the ALU via the register file.
- *Instruction loops*. Code memory moves to the fetch unit included in the CU (not shown in the figure) in response to an instruction address.

In the adaptive implementation, each loop is logically present in different fractions of time according to different programs and input data. We considered two simple benchmarks: a data sort that exercises the data dependency of the results and the branch loop, and a matrix multiplication that extensively uses the data loops.

The adaptive implementation fairly easily isolated important conditions, such as the following, that allow the determination of status (processing or idle) of the five functional units with respect to their inputs:

- Data from the DC (*dataout*) is written in the RF only on a write cycle (the CU's *wr* signal in Figure 3a); otherwise, DC *dataout* can be ignored.
- The CU needs flags from the ALU only during execution of a conditional branch.

```
entity RF is
  port (rst : in STD_LOGIC;
  rd : in STD_LOGIC;
  wr : in STD_LOGIC;
  ...
  rf_src1 : in UNSIGNED (4 downto 0);    -- source reg 1 address and
  p_rf_src1 : out STD_LOGIC;             -- related PROCESSING bit
  rf_src2 : in UNSIGNED (4 downto 0);    -- source reg 2 address and
  p_rf_src2 : out STD_LOGIC;             -- related PROCESSING bit
  rf_des1 : in UNSIGNED (4 downto 0);    -- dest reg 1 address and
  p_rf_des1 : out STD_LOGIC;             -- related PROCESSING bit
  ...
end RF;
  ...
  process (rd, wr, from_mem)
  begin
  if( rd = '1' ) then                    -- read cycle: addresses
    p_rf_src1 <='1';                     -- of source registers
    p_rf_src2 <='1';                     -- have to be processed!
  end if;
  if( wr = '1' ) then                    -- write cycle: address
    p_rf_des1 <='1';                     -- of dest register
  ...                                    -- has to be processed
  end process;

entity ALU is
  port(op_code : in UNSIGNED (3 downto 0);
  ...
  src_1 : in UNSIGNED (15 downto 0);     -- src_1 input
  p_src_1 : out STD_LOGIC;               -- src_1 PROCESSING bit
  src_2 : in UNSIGNED (15 downto 0);     -- src_2 input
  p_src_2 : out STD_LOGIC;               -- src_2 PROCESSING bit
  ...
end ALU;
  ...
  process (op_code)
  begin
  ...
  case op_code is                        -- switch based on op_code
    when OP_IS_ADD =>                    -- when ADDITION
      p_src_1 <= '1';                    -- process both inputs
      p_src_2 <= '1';                    -- src_1 and src_2
    when OP_IS_MOVE =>                   -- when MOVE
      p_src_1 <= '1';                    -- process only src_1
  ...
  end case;
  end process;
```

Figure 4. Extract of VHDL code showing processing signals in the register file (RF) and the ALU.

Such conditions are both easy to isolate from a partial knowledge of the functional units' behaviors and simple to implement as an oracle of minimal size (which is important for both area and delay concerns).

As an example VHDL description of an oracle, Figure 4 shows an extract of the RF and ALU codes that assign the processing signals related to the RF's source and destination registers and to the ALU operands.

To assess absolute performance gain, we manually optimized the microprocessor floorplan shown in Figure 3b and then evaluated the overall data gain of the LIP systems. We estimated the areas of the various blocks on the basis of typical gate counts and memory sizes.

We generated the floorplan so as to avoid placing blocks with performance-critical communication channels (such as from the IC to the CU) too far apart. The communication between the CU and the other units is the most critical (more so for the data sort case). Therefore, we placed the CU at the center of the floorplan. The longest wires connect the DC to the ALU and the RF, and they are also less critical communication channels. The next-to-longest connection is from the ALU to the RF and is critical for the matrix multiplication but not for the sort.

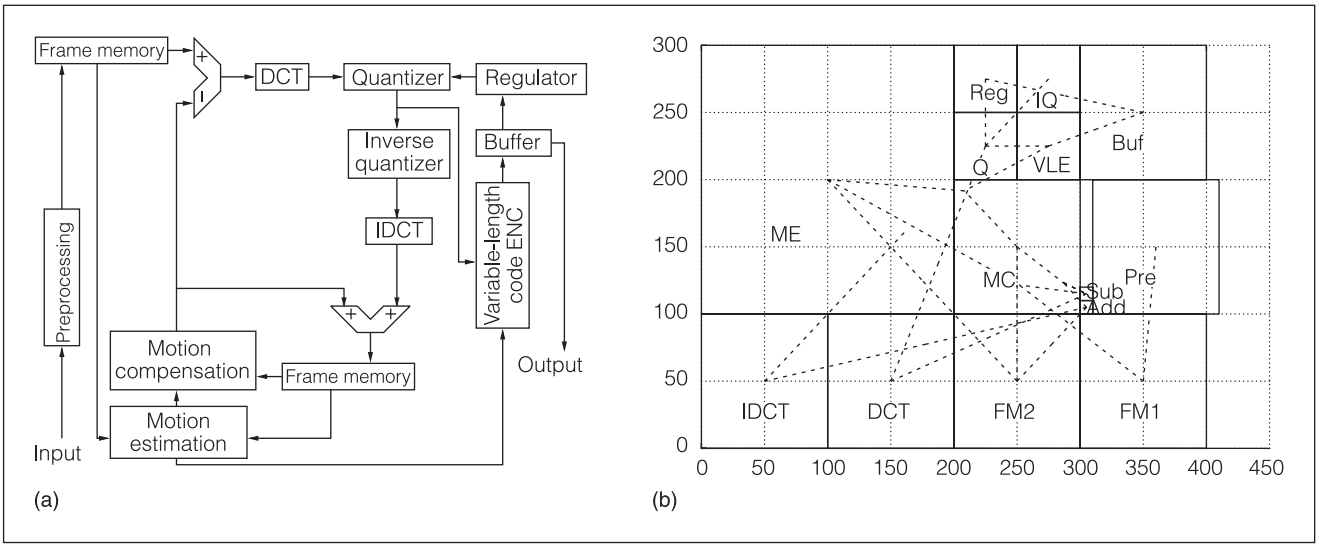

Figure 5. MPEG encoder block scheme (a) and floorplan (b). Dashed lines in (b) represent connections between processor blocks. (Add: addition; Buf: buffer; DCT: discrete cosine transform; IDCT: inverse discrete cosine transform; Enc: encoder; FM: frame memory; MC: motion compensation; ME: motion estimation; Pre: preprocessing; Q: quantizer; IQ: inverse quantizer; Reg: regulator; Sub: subtraction; VLE: variable-length encoder.)

After computing such distances, we could estimate the wire latencies and evaluate the system *data rate* (the product of throughput and frequency) as a function of the *critical length* (the maximum admitted distance between two relay stations, given a frequency constraint—shorter critical lengths correspond to higher frequencies).

MPEG encoder

The fixed MPEG communication pattern is extremely bursty, so there are always periods of nonmutual communication. This behavior suggests that the adaptive LIP should be more suited to this case than to the microprocessor. Following the description by Ikeda et al.,[7] already used in the context of LIPs,[5] we implemented the skeleton of an encoder that respects all the MPEG communication patterns.

As Figure 5a shows, the tightest loop, involving three blocks, is less strict than in the microprocessor in Figure 3a, in which two-block loops were present. Besides a single four-block loop, all other blocks belong to larger loops. Through a clever floorplan, we kept the blocks of short loops close to one another so that we could limit the insertion of relay stations to branches that appear only in loose loops. As a result, the small throughput reduction and larger clock frequencies guaranteed by wire pipelining could significantly speed up both the static and the adaptive cases.

We evaluated the throughput of the automatically generated floorplan in Figure 5b (we give further details elsewhere[3]). Again, we estimated the areas of the blocks on the basis of typical gate counts. Figure 5b shows the placement of the short loops. The shortest loop consists of motion compensation (MC), second frame memory (FM2), and adder (Add). The same is true for the loop involving the buffer (Buf), the regulator (Reg), the quantizer (Q), and the variable-length encoder (VLE). The longest connection, and so the most likely candidate for wire pipelining, is between the inverse quantization (IQ) and the inverse discrete cosine transform (IDCT), which are members of an eight-block loop. The floorplan confirms our intuition concerning potential causes of throughput reduction.

In this case, the approach we used to derive the oracles was different. From the system description, we knew a priori the communication pattern between units and could statically define the correct activation sequence for all processing signals. Moreover, we could do this without a deep knowledge of the blocks or any form of reverse engineering. We think that this approach can be adapted to other cases in which data computation occurs through streaming between functional units, as in many DSP applications.

We ran VHDL simulations after floorplanning, with and without the pipelining elements calculated from the block-to-block distances in the layout.

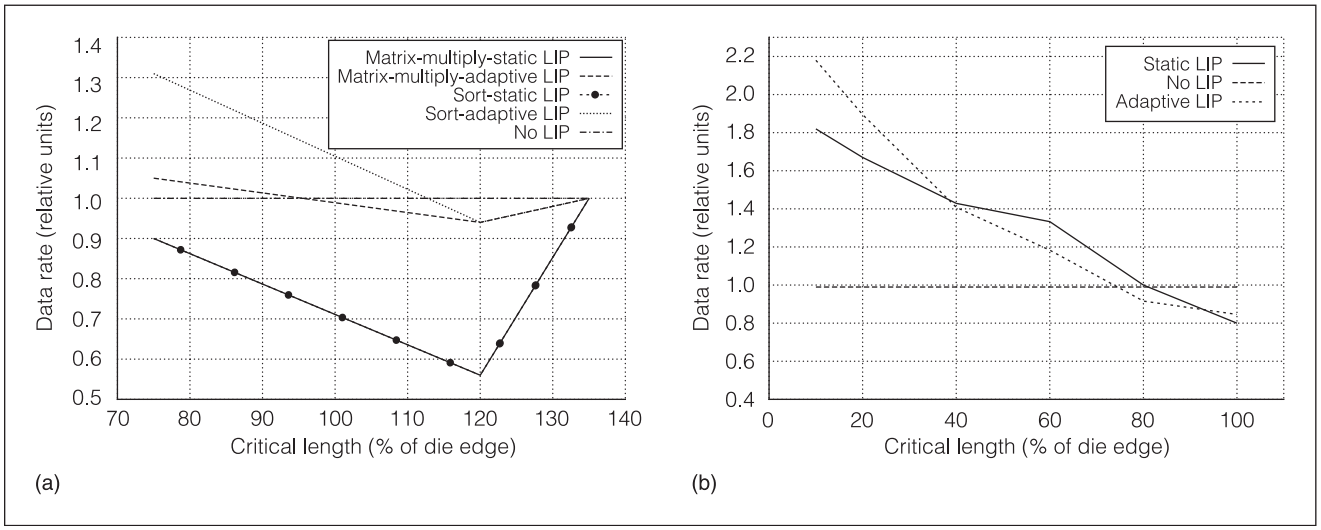

Figure 6. Data rate comparisons: microprocessor (a) and MPEG encoder (b).

Results

Figure 6a and Figure 6b show the results for the microprocessor and MPEG experiments. The no-LIP systems are assigned the nominal data rate of 1 (frequency 1, throughput 1). The LIP systems have a data rate resulting from the product of the increased frequency due to shorter wires and the possibly reduced throughput.

The microprocessor case results show that the adaptive system manages to increase performance, but the actual advantage depends on the benchmark. As expected, the matrix multiplication was not as favorable, because of the criticality of the ALU-RF communication. The code had no effect on the static system; the two programs had the same throughput, so the two curves perfectly overlap.

The nonadaptive methodology fails to provide any advantage, because of the tightly interconnected blocks: Introducing pipelines in every branch immediately halves the throughput, wiping out any frequency advantage. This explains the slope change of the data rate curves: A single latency due to a frequency constraint even slightly stricter than the minimum necessary to avoid wire pipelines immediately reduces throughput and is not compensated for by the small frequency increase.

In contrast, the results in Figure 6b show an enormous advantage of MPEG LIP systems over the microprocessor case. The existence of a range of critical lengths for which the static implementation outperformed the adaptive one is attributable to the fact that the floorplans were slightly different because the physical design tool used different cost functions for the two optimizations.[3] The best results for the adaptive case are for short lengths (and thus high frequencies). We must weigh the data rate's more than doubling in the no-LIP case against the simplifying assumptions we made here and particularly clock tree synthesis, skew control, and the absence of logic limitations. Nonetheless, the graph shows that there is abundant design space in which to proceed toward faster systems.

We are presently working toward defining automated techniques for extracting processing signals from a synthesizable code or a gate-level netlist. In the latter case, logical and testing techniques (observability, controllability, and don't-care extraction) can help. Should the proposed or a similar LIP methodology become standardized, the designers would already provide processing signals as part of their blocks' regular output, thus making the extraction technique unnecessary.

We envision the need for far more research on latency insensitivity in GALS systems, mixed-clock relay station sizing, and physical design aspects such as latency-aware floorplanning, placement, and routing. This research is particularly important because the hypothesis of full synchronicity in future high-performance and large systems is doubtful, but the increase of wire delays over gate delays is a fact. We advocate an advancement of such research and hope to contribute to it in forthcoming works. ■

References

1. M.R. Casu and L. Macchiarulo, "A New Approach to Latency Insensitive Design," *Proc. 41st Design Automation Conf.* (DAC 04), ACM Press, 2004, pp. 576-581.
2. M.R. Casu and L. Macchiarulo, "Throughput-Driven Floorplanning with Wire Pipelining," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, May 2005, pp. 663-675.
3. M.R. Casu and L. Macchiarulo, "Floorplanning with Wire Pipelining in Adaptive Communication Channels," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, Dec. 2006, pp. 2996-3004.
4. M. Singh and M. Theobald, "Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures," *Proc. Design, Automation and Test in Europe Conf.* (DATE 04), IEEE CS Press, 2004, vol. 2, pp. 1008-1013.
5. L.P. Carloni and A. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency Insensitive Systems," *Proc. 37th Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 361-367.
6. A. Agiwal and M. Singh, "An Architecture and a Wrapper Synthesis Approach for Multi-Clock Latency-Insensitive Systems," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 05), IEEE CS Press, pp. 1006-1013.
7. M. Ikeda et al., "SuperENC: MPEG-2 Video Encoder Chip," *IEEE Micro*, vol. 19, no. 4, Jul./Aug. 1999, pp. 56-65.

Mario R. Casu is an assistant professor in the Department of Electronics at Politecnico di Torino, Italy. His research interests include circuits, technology, and architectures for ultra deep-submicron SoCs. Casu has an MSc and a PhD in electronics engineering from Politecnico di Torino. He is a member of the IEEE.

Luca Macchiarulo is an assistant professor in the Department of Electrical Engineering of the University of Hawaii at Manoa. His research interests include interactions of physical design and logic synthesis, on-chip communication, and biomedical applications. Macchiarulo has an MSc and a PhD in electronics engineering from Politecnico di Torino.

■ Direct questions and comments about this article to Mario R. Casu, Politecnico di Torino, Dipartimento di Elettronica, Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy; mario.casu@polito.it.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.