Synthesizing Synchronous Elastic Flow Networks

Greg Hoover and Forrest Brewer
{ghoover, forrest}@ece.ucsb.edu
University of California, Santa Barbara

Abstract

This paper describes an implementation language and synthesis system for automatically generating latency insensitive synchronous digital designs. These designs decouple behavioral correctness from design performance by allowing any sub-component to dynamically stall without changing correct system activity. This is accomplished by imposition of global invariants and use of local control in the form of Synchronous-Elastic Flow (SELF) networks, which are directly synthesized. This design description format reduces the complexity of implementing correct SELF networks and does not require pre-design of a correct conventional synchronous design. The design description is a specialized guarded atomic action language which is particularly suited for succinctly describing SELF designs. We present the language syntax, semantics and synthesis techniques illustrated by the design of a latency tolerant cache controller.

1 Introduction

System designers face substantial challenges when it comes to behavioral verification and component integration. IP-reuse enables ever-greater system functionality with reduced development time, but at cost to communication and control, which must facilitate sharing and synchronization between distributed components while simultaneously respecting physical and temporal constraints. Latency-insensitivity[2] has been proposed as one method for coping with this growing problem. In such a design model, components are expressed in terms of time-independent behavioral abstractions. Effectively, the designer need not specify any sequencing or scheduling information beyond behavioral causality and is assured that the design will function correctly despite dynamic latency changes in any of the architectural components. Thus the resulting designs are inherently tolerant to dynamic computation and communication delays. Composition of system components is greatly simplified since component interfaces are not bound to temporal constraints. A direct benefit of this design style is removal of a large number of critical nets that are required in conventional designs to mediate communication or operation latency changes.

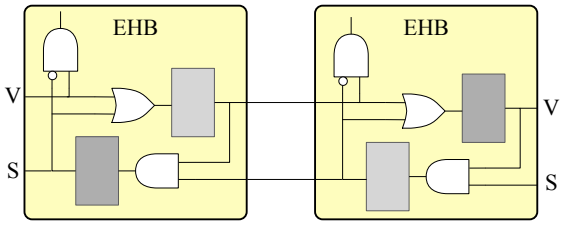

Figure 1. Elastic Buffer (EB) implemented as pair of Elastic Half-Buffers (EHB)

Automatic implementation techniques for latency-insensitive circuits, however, have largely eluded designers. The conventional preference to construct centralized controllers consistently leads to implementations that require substantive hardware and control overhead due to unknown signaling or transaction latencies, especially in large designs. Recently, the advent of Synchronous-ELastic Flow (SELF)[5, 4] provides a protocol and supporting circuit implementations for creating efficient and scalable implementations of elastic systems. Using only two signals (in its basic form) and a handful of atomic control circuits, the designer can compose complex control structures that are completely elastic and therefore immune to latency variations. In addition, these control structures are inherently distributed, providing numerous benefits over global control techniques. Finally, SELF is built on half-cycle or latch-level timing where data-path components are only clocked when necessary. Thus system implementations can achieve higher performance and, through use of naturally gated clocks and relaxed timing margins, lower power consumption. Moreover, this clocking structure is entirely synchronous, allowing use of conventional design flows for synthesis and timing analysis.

To date, realizing SELF control structures is a manual, error-prone process, where even small designs become difficult to debug. Rather, it is desirable to have a technique for synthesizing such structures from a high-level specification such that appropriate SELF circuits are inserted as a consequence of behavioral connectivity requirements. Unfortu-

978-3-9810801-3-1/DATE08 © 2008 EDAA

nately, classical HDL languages are at odds with an elastic implementation technology as they are built on execution models that assume fixed temporal relationships between statements. Nearly all hardware specification models reduce to centralized control mechanisms (due to the assumption of control signal locality), resulting in implementations whose performance is bounded by critical paths which are not apparent in the specification. These critical paths often stem from timing assumptions and lack of modularity in the control specification.

An exception are languages based on guarded atomic actions (GAA)[11, 9]. In a GAA language, functionality is encapsulated within time-independent tasks that are triggered (or guarded) by conditional expressions. By treating these expressions as token acceptors and their tasks as token producers, the resulting model becomes similar to that of the SELF protocol, where the representative dependency network is synthesizable. The designer is able to focus on functional reasoning over procedural sequencing, but at the cost of design performance predictability as the language has no intrinsic timing characteristics. In this approach, timing constraints are applied as optimization criteria, rather than as specified integral attributes of the design behavior. In this way, a design is guaranteed to be functionally correct over a range of target implementations offering the designer a landscape on which to define implementation characteristics.

In this paper, we present a synthesis strategy for constructing SELF elastic control networks from a rule-based language. The language has been designed from a practicality standpoint, emphasizing constructs for common control operations, including speculation, multi-way decision points and out-of-order execution, while expressing much of the underlying SELF protocol semantics through use of token-based control flow. The remainder of this paper is organized as follows: Background work is presented in Section 2. A brief overview of the rule-based language is presented in 3. The execution semantics of synthesized machines are described in Section 4. And the synthesis technique is presented in Section 5. Concluding remarks are made in Section 6.

2 Background

This work draws from existing work related to the SELF control networks and guarded atomic action languages. In this section we present prior work related to our synthesis of these two technologies.

2.1 Synchronous-Elastic Flow (SELF)

Synchronous-ELastic Flow (SELF) defines a formal protocol and circuit primitives for creating elastic networks. These networks are a collection of elastic modules (circuits) and channels that conform to the SELF protocol. Channels are comprised of two control wires that implement a simple handshake between sender and receiver. Forward propagation occurs over a 'valid' (V) wire with backpressure being asserted over a 'stop' (S) wire. SELF networks are token propagating networks, where the absence of a token is a bubble and all modules agree on the following channel states:

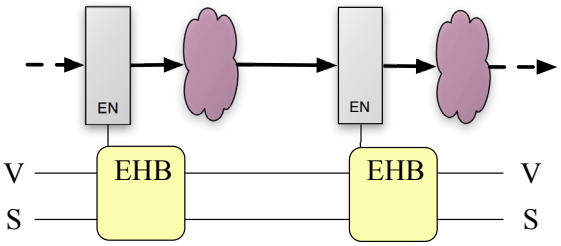

Figure 2. Execution model illustrating coupling of control and data.

- (T) Transfer, (V, ¬S): the sender provides valid data and the receiver accepts it.
- (I) Idle, (¬V): the sender does not provide valid data.
- (R) Retry, (V, S): the sender provides valid data but the receiver does not accept it.

Elastic Buffers (EB) replace traditional flip-flop-based storage nodes, dividing master and slave flip-flop phases into two distinct buffers. An EB can be implemented as a pair of Elastic Half-Buffers (EHB), where the EHBs are triggered on opposing clock phases as shown in Figure 1. Elasticity is seen when an asserted stop signal puts a channel into a retry state. While similar to traditional interlocked pipelines, functional separation of clock phases (EHB) allows segmentation of the stall path. This is possible because tokens propagate on half cycles, but enter the network only on cycle boundaries, effectively adding a half-cycle buffer between tokens. This slack between tokens allows dynamic buffering within the pipeline without additional hardware. This is a major advantage over interlock pipelining, where stall paths must span many pipeline stages resulting in potential critical paths. Finally, the use of latch-based storage offers recovery of much of the space and potential for significant power reduction by selectively clocking only those components with valid data. From a timing point of view, the network functions synchronously, with guarded clocks based only on activity of the previous cycle.

Decision points are a requirement for any practical control system. In SELF, decisions are facilitated by anti-tokens generated at non-local control points[3, 8]. Tokens and anti-tokens annihilate each other, creating a bubble in the network where they meet. Anti-tokens may take on active and passive forms enabling them to propagate or remain stationary in the control network; and enable early evaluation.

2.2 Guarded Atomic Actions

Hoe and Arvind[7] proposed an operation-centric hardware abstraction model useful for describing systems that exhibit a high degree of concurrency. In this model, operations are specified as atomic actions, permitting each operation to be formulated as if the rest of the system were frozen. As a specification medium, rule-based languages allow the designer to specify functional tasks independently of the rest of the system. This characteristic allows functional isolation which greatly reduces the effort in reasoning about composed system behavior. The syntax of a rule has the form: $antecedent \rightarrow actions$, where the antecedent is an expression used to trigger execution of the rule's actions. Machine behavior is the aggregate set of causal relationships between rules, irrespective of any temporal constraints.

Our use of the term re-writing paradigm largely exploits the evaluation freedom inherent in the semantic model. We are not currently making use of term re-writing based syntax translation or semantic matching beyond the dynamic status of tokens. Alternative uses of term re-writing for hardware and even pipeline synthesis are exemplified by the work of Marinescu [9][10] and Ayala-Rincon[1]. These systems make extensive use of term re-writing to create particular instances of algorithms or functional mappings within hardware constraints. Essentially, rules are used to provide design refinement in the form of mapped structure bound to rule execution. Our aim is to automate the process of creating a distributed token network which efficiently manages the behavior exploring architectural changes in the design. This pattern is similar to the use of these systems by Hoe[6].

Algorithm 1. Specification language grammar

```
module_declaration ::= module rule_declarations endmodule
rule_declarations ::= rule { rule }
rule ::= antecedent → action {, action } ;
antecedent ::= antecedent_func | token
antecedent_func ::= and | or ( token , token {, token } )
action ::= token | action_kill | function | if_stmt
        | case_stmt
action_kill ::= kill ( token {, token } )
function ::= identifier ()
if_stmt ::= if ( function ) action { action }
        { else_stmt } endif
else_stmt ::= else action { action }
case_stmt ::= case ( function ) case_option
        { case_option } endcase
case_option ::= number : action { action }
token ::= identifier
number ::= r'{[}0-9]+'
identifier ::= r'{[}a-zA-Z]{[}a-zA-Z0-9_-]{*}'
```

3 Language

Our language, TDL, is a modular, GAA language that exposes the underlying semantic execution model by using token-based control flow in the language syntax. The language grammar is shown in Algorithm 1, where the rule structure $expression \rightarrow functions, tokens$ recognizes positive expressions of tokens and triggers creation and execution of tokens and functions respectively. This structure parallels SELF elastic semantics since rule sequencing is temporally ambiguous, as are dynamic delays between SELF EHBs. Control feedback is supported through *if-else* and *case* selection constructs in the action clause.

TDL is complemented by Verilog HDL for describing the supporting datapath functionality. Function data is annotated to allow identification of shared variables and synthesis of datapaths, as well as network optimization and resource sharing. Shared variables are referenced by name and synthesized to physical register locations upon output. Verilog functions are particularly well suited for atomic (unit time) semantics since they are, by definition, stateless. In systems requiring substantial context or persistent state, it becomes inefficient to replicate data throughout the control network. A processor register file, for instance, should probably be arbitrated rather than propagated to every dependent pipe-stage. Data annotations allow unambiguous identification of such contention points; and arbitration can be automatically synthesized. Optimizations can subsequently be performed on the control network to improve the performance of target sections.

4 Execution Semantics

Unlike conventional description languages which, at best, promote data locality at the module level, TDL couples data and control locally. This tight coupling can be seen in Figure 2, where the enable output of each elastic-half-buffer (EHB) selectively enables its coupled data register. This model may appear familiar since it is commonly used in pipeline constructions, allowing the designer to divide functionality into a series of semi-autonomous stages. TDL takes this one step further by exploiting SELF elasticity, making each stage fully-autonomous and allowing functional reasoning independently of other system behaviors. A small set of connector circuits preserves the global invariants that facilitate SELF elasticity throughout the control network. These circuits provide well-defined mechanisms for control branch (fork) and synchronization (join), allowing synthesis by composition of a few circuit primitives. Latency freedom creates possibilities for non-standard components like variable delay execution units and memories, as well as low-overhead speculation and arbitration mechanisms. Furthermore, selective clocking offers opportunities for lowering power consumption, and improved design locality aids in physical layout and verification.

5 Synthesis Technique

TDL aims to provide a sound functional implementation base from which system constraints can be reliably inferred without impacting correctness. To this end, the synthesis process constructs a behavioral graph that is amenable to a

number of different optimizations, allowing exploration of a range of implementations while preserving behavioral correctness. Simple heuristics guide graph construction from a specification rule set and infer connector circuits as necessary. To clearly demonstrate our technique, we follow the design of a cache controller from specification through the stages of compilation and synthesis.

Algorithm 2. Cache controller specification.

```
module CacheController
  Reset -> MemoryArbiter,
           kill( Request, Read, ReadDone,
                 ReadFromMemory, ReadFromBus, Write,
                 WriteToMemory, WriteDone );

  Request -> drive_address(),
             case ( type () )
               0: Read;
               1: Write;
             endcase;

  Read -> if ( tag_miss() )
            if ( requires_writeback() )
              buffer_dirty(), WriteToMemory
            endif,
            ReadFromMemory
          else
            return_cache_data(), ReadDone
          endif;

  and( MemoryArbiter, ReadFromMemory ) ->
       setup_read_from_bus(), ReadFromBus;

  ReadFromBus -> MemoryArbiter, update_cache_from_bus(),
                 return_bus_data(), ReadDone;

  Write -> if ( requires_writeback() )
             buffer_dirty(), WriteToMemory
           endif,
           update_cache_from_input(), WriteDone;

   and( MemoryArbiter, WriteToMemory ) ->
     write_to_bus(), MemoryArbiter;
endmodule
```

5.1 Specification

Algorithm 2 presents a specification for a simple cache controller illustrating a number of control decision constructs and language features, with a sample of the supporting Verilog datapath functions shown in Algorithm 3. The entry of this spec highlights one of the advantages of our token-based model: reset is handled through the token mechanism, enabling staged behaviors necessary in practical design. This is in contrast to conventional HDL specifications where design patterns often promote a global reset functionality that is difficult and error prone to realize. Tokens are persistent and therefore require a mechanism for their destruction. The builtin *kill* function handles this, and is shown here initializing the system to a known state. The token *MemoryArbiter* is also created during reset as an arbitration mechanism for access to the memory bus.

Execution of this component begins when an enclosing component creates the *Request* token or the behavior is similarly invoked by a wire input. Associated arguments including the type, address, and data accompany the request and are propagated as necessary. Feedback functions such as *type(), tag_miss(),* and *requires_writeback()* form local control decision points by referencing coupled data registers. For example, the function *tag_miss()* checks the tag of the request address against that in the global tag memory to determine if a cache hit or miss has occurred. Action functions like *buffer_dirty()*, *return_cache_data()*, and *setup_read_from_bus()* perform datapath functionality including memory bus read and write operations and updates to shared memories. One can see that the division of functionality and data naming convention simplifies the design effort. For example, the data variable *tmp_write_buffer* is written to in *buffer_dirty()* and read from using the same name in *write_to_bus()*. While these references would likely by realized using a single register in conventional specification, here, the data is automatically moved through the control network by the compiler to maintain data locality.

Algorithm 3. Sample of supporting datapath functions for the simple cache controller design Algorithm 2 (some omitted for brevity).

```
function type;
  // read: opt_type
  type = op_type;
endfunction

function tag_miss;
  // read: tag_cache[CACHE_DEPTH][TAG_WIDTH-1:0]
  // read: op_addr[ADDR_WIDTH-1:0]
  tag_miss = tag_cache[op_addr[ADDR_WIDTH-1:TAG_WIDTH]]
               == op_addr[TAG_WIDTH-1:0];
endfunction

function buffer_dirty;
  // read: data_cache[CACHE_DEPTH][DATA_WIDTH-1:0]
  // read: op_addr[ADDR_WIDTH-1:0]
  // write: tmp_write_buffer[DATA_WIDTH-1:0]
  tmp_write_buffer =
          data_cache[op_addr[ADDR_WIDTH-1:TAG_WIDTH]];
  buffer_dirty = 1;
endfunction

function write_to_bus;
  // read: tmp_write_buffer[DATA_WIDTH-1:0]
  // read: op_addr[ADDR_WIDTH-1:0]
  // write: address_bus[ADDR_WIDTH-1]
  // write: data_bus[DATA_WIDTH-1], wr, rd
  address_bus = op_addr;
  data_bus = tmp_write_buffer;
  wr = 1;
  rd = 0;
  write_to_bus = 1;
endfunction

function update_cache_from_input;
  // read: input_data[DATA_WIDTH-1:0]
  // read: op_addr[ADDR_WIDTH-1:0]
  // write: cache_tag[CACHE_DEPTH][TAG_WIDTH-1:0]
  // write: cache_data[CACHE_DEPTH][DATA_WIDTH-1:0]
  // write: cache_status[CACHE_DEPTH][STATUS_WIDTH-1:0]
  cache_tag[op_addr[ADDR_WIDTH-1:TAG_WIDTH]] =
                op_addr[TAG_WIDTH-1:0];
  cache_status[op_addr[ADDR_WIDTH-1:TAG_WIDTH]] = 'DIRTY;
  cache_data[op_addr[ADDR_WIDTH-1:TAG_WIDTH]] = input_data;
  update_cache_from_input = 1;
endfunction
```

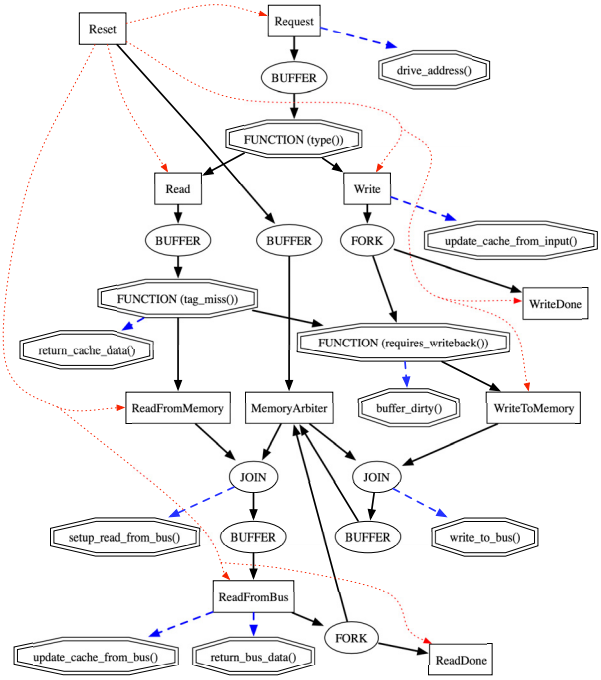

Figure 3. Pre-optimization behavior graph compiled from the simple cache control specification in Algorithm 2.

5.2 Compilation

The multi-step compilation process takes an input specification and constructs a dependency graph from token relationships. During construction, antecedent logic functions (e.g. and, or) are built, *FUNCTION* nodes are inserted for datapath feedback decision points, and SELF forks are inserted for distribution of signals within action clauses. Figure 3 illustrates the corresponding behavior graph from the specification in Algorithm 2, where control flow, output function, and kill arcs are shown solid, dashed, and dotted, respectively. The causal token relationships define machine behavior; subsequent stages merely refine the graph to preserve required token invariants and optimize the structure.

A requirement of SELF networks is that all control distribution and synchronization activities occur through defined circuit primitives that comply with the elastic channel protocol. The first refinement stage ensures that all such points are correctly handled, inserting FORK (distribution) and JOIN (synchronization) circuits as necessary. A comparison of Figures 3 and 4 highlights these changes. Inputs to the *MemoryAribter* token are shown routed through a SELECT connector, with outputs distributed through a FORK connector.

Latch-based synthesis adds additional constraints, requiring that adjacent control nodes be of opposing clock phase and shared data be accessed on a common edge. Compilation ensures this by enforcing even depth in all control loops and through data access analysis. Clock polarity conflicts are resolved by inserting a timing buffer as shown in Figure 4, where *__tdl_timing_buffer* token is necessary to enforce even weight of the loop into token *MemoryArbiter*. In a conventional specification, this action could lead to incorrect system behavior. In our case, elasticity in the design behavior ensures behavioral correctness under any such transformation. Such retimings open possibilities for optimizations that improve performance and reduce power. Finally, this stage assigns clock polarities and prepares for synthesis by locating functions, analyzing data access, and setting networks for data propagation.

5.3 Synthesis

In the final synthesis stage, the optimized behavior graph is output as synthesizable Verilog HDL that connects EHBs (or EBs) and connector circuits, along with functional datapath RTL. Synthesis of the control network is performed through a straightforward traversal of the control graph, whereby nodes are built by swapping in respective circuit implementations. Functional RTL is connected to the respective enabling outputs of EHBs (or EBs) and wrapped in sequential blocks based on latch polarity, if necessary. Using previously generated data propagation networks, registers are constructed and wired to facilitate data movement in the network, conforming to the localized SELF model. The RTL output is then synthesized to the target technology. The control logic for the above design was synthesized using Synopsys Design Compiler in TSMC 90nm (Low Power) technology. It requires 427μm and clocks at over 2.3GHz without any optimizations. These numbers omit both area and delay for the large memory elements in the design, as these would clearly dwarf any control logic costs. It is important to note that the resulting design functions correctly even under dynamic memory latency, for example if it was dependent on data locality.

Verilog output enables integration with existing design flows and synthesis tools for performing gate-level optimizations, while latency-insensitivity aids in the synthesis and layout flows since latency characteristics do not affect functional correctness. The coupled control and data model allows efficient resource location compared to the critical paths created by long wires necessary for globally accessible resources. Moreover, the inherently distributed nature of elastic control networks minimizes the possibility for creating artificial control critical paths which are common in conventional Verilog synthesis. While synthesized designs are guaranteed to be behaviorally equivalent to their specification, they provide no immediate insight into the latency or throughput of the implementation. Such bounds can be determined, in general, via model checking, providing both latency and throughput bounds given constraints on the underlying components. This information can be used to perform additional 'tuning' of the design to achieve target constraints.

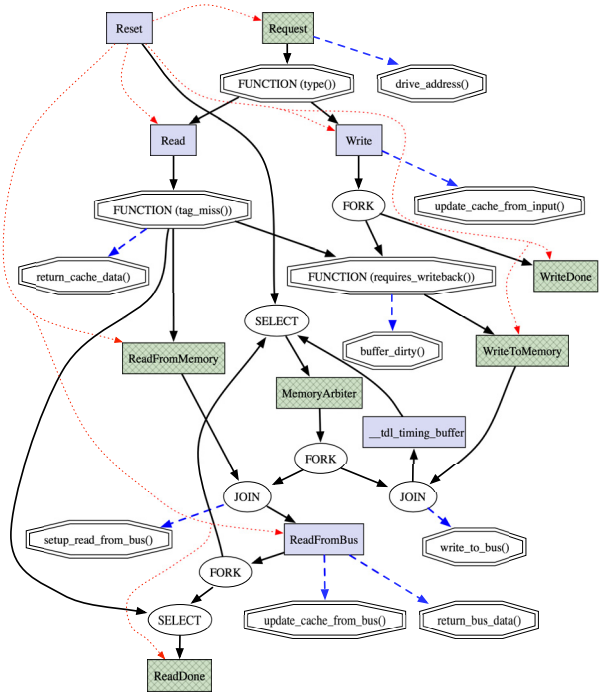

Figure 4. Compiled behavior graph from the simple cache controller specification in Algorithm 2.

Synthesizing to SELF networks required several additions to the fundamental SELF circuit primitives. For instance, SELF lacks a mechanism for creating decision points with or-causality, as this would negate the ability to guarantee ordering of tokens. While this invariant can be useful in guaranteeing system correctness, it is often the case that control specifications cannot be succinctly expressed without it. Moreover, fixed token ordering prevents possibilities for speculative execution. As previously mentioned, implementation efficiency may dictate that migrating large memories throughout the control network is impractical. In such cases, arbitration is required. Prioritized selection logic is used to implement these data-derived arbitration points. Because system behavior is latency-insensitive, any prioritization of access is behaviorally correct. Furthermore, optimization offers the ability to lower power and improve performance through re-prioritization of these points.

6 Conclusions

Latency-insensitive specification is a viable method for curbing the growing complexity of modern designs. Through synthesis of truly distributed control using Synchronous-ELastic Flow (SELF) networks, we believe it is possible to realize scalable systems that achieve low power and high performance while reducing design effort. Much of this is predicated on the ability to construct latch-based systems that only clock active components and allow relative synchrony between component clocks to be more loosely defined. Furthermore, the structure of synthesized elastic networks is amenable to a wide range of optimizations without sacrificing correct system behavior. Such optimizations include locality-directed design restructuring to improve performance and lower resource overhead, as well as, resource use optimization through analysis of data access patterns. Our language offers a practical solution to constructing SELF control networks, enabling manageable and scalable system design.

References

[1] M. Ayala-Rincon, C. H. Llanos, R. P. Jacobi, and R. W. Hartenstein. Prototyping time- and space-efficient computations of algebraic operations over dynamically reconfigurable systems modeled by rewriting-logic. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):251–281, 2006.

[2] L. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. pages 1059 – 1076, 2001.

[3] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, 2007.

[4] J. Cortadella, M. Kishinevsky, and B. Grundmann. Self: Specification and design of synchronous elastic circuits. In *TAU '06: Proceedings of the ACM/IEEE International Workshop on Timing Issues 2006*, 2006.

[5] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 657–662, New York, NY, USA, 2006. ACM Press.

[6] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI '99: Proceedings of X IFIP International Conference on VLSI*, 1999.

[7] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[8] J. Julvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006.

[9] M.-C. Marinescu and M. Rinard. High-level specification and efficient implementation of pipelined circuits. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 655–661, New York, NY, USA, 2001. ACM Press.

[10] M.-C. V. Marinescu and M. Rinard. High-level automatic pipelining for sequential circuits. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 215–220, New York, NY, USA, 2001. ACM Press.

[11] D. Rosenband and Arvind. Modular scheduling of guarded atomic actions. In *DAC '04: Proceedings of the 41st Design Automation Conference (DAC)*, 2004.