

Transformations for the Synthesis and Optimization of Asynchronous Distributed Control

Abstract

Asynchronous design has been the focus of renewed interest. However, a key bottleneck is the lack of high-quality CAD tools for the synthesis of large-scale systems which also allow design-space exploration. This paper proposes a new synthesis method to address this issue, based on transformations.

The method starts with a scheduled and resource-bounded Control-Data Flow Graph (CDFG). Global transformations are first applied to the entire CDFG, unoptimized controllers are then extracted, and, finally, local transforms are applied to the individual controllers. The result is a highly-optimized set of interacting distributed controllers. The new transforms include aggressive timing- and area-oriented optimizations, several of which have not been previously supported by existing asynchronous CAD tools.

As a case study, the method is applied to the well-known differential equation solver synthesis benchmark. Results comparable to a highly-optimized manual design by Yun et al. [35] can be obtained by applying the new automated transformations. Such an implementation cannot be obtained using existing asynchronous CAD tools.

1 Introduction

Asynchronous design has been the focus of much recent interest and research activity [13]. Several commercial asynchronous chips have been produced in the last couple of years (e.g., microcontroller chips in Philips' commercial pagers [17]), and several companies have designed experimental asynchronous chips (e.g., Intel's high-speed instruction-length decoder [31] and Sun's high-speed pipelines [9]).

However, a key current limitation is the lack of high-quality CAD tools for systematic design-space exploration and optimization of large-scale asynchronous systems. Traditionally, a number of asynchronous CAD tools are limited to the design of individual controllers [14, 15, 12, 34, 26, 3, 22, 20], and thus are only useful for one step in the overall synthesis flow.

For large-scale asynchronous systems, two design approaches are now widely-used: (i) *manual design*, and (ii) use of *syntax-directed CAD tools*. Manual design allows a number of aggressive optimizations, but is cumbersome, slow and error-prone, and it does not provide systematic and automated exploration of the design space. For example, the Intel asynchronous instruction-length decoder chip [31] took over two years to complete, using a combination of manual techniques and academic synthesis tools for designing individual controllers.

Alternatively, several automated approaches have been proposed for large-scale systems which are syntax-

directed [7, 4, 5, 30]. These methods start from a high-level abstraction, such as a concurrent program, and obtain a circuit by translating each individual program construct into a corresponding sub-circuit. For example, the Philips' Tangram tool [4, 5] provides only one implementation per specification. There are few options, other than some simple peephole techniques, for design-space exploration. If the user is dissatisfied with the circuit, the original program must be manually restructured and re-compiled in the hopes of some improvement. Other CAD approaches allow only restricted and non-systematic techniques for exploration [24, 7]. Thus, a huge potential for optimization has been overlooked for a long time.

Recently, there is increasing interest in alternative approaches to the synthesis of large-scale asynchronous systems [2, 10, 21, 6, 8, 18, 23, 19, 27, 1]. Cortadella and Badia propose a synthesis style for the control unit such that each datapath block is controlled by a dedicated sub-controller [10], while Kim et al.'s approach subdivides these sub-controllers even further, assigning a sub-sub-controller to each of the processes bound to a functional unit [18]. Each of these approaches is strictly deterministic and "template-based". Only one approach [1] considers design space exploration, but at a higher level: for resource binding and allocation. Interestingly, some efficient manual designs have been presented to which none of these methods has access. Thus, there is still a serious lack of approaches providing systematic design space exploration.

The contribution of this paper is a new approach for the automated synthesis and optimization of large-scale asynchronous systems. In particular, this paper is the first to introduce, formalize and automate a wide-ranging and powerful set of transformations, which can be used for the synthesis of asynchronous distributed control. Unlike previous approaches, these new transforms can be applied in a systematic way to explore the design space and find optimal distributed controller implementations.¹

The new method starts with a given scheduled and resource-bounded Control-Data Flow Graph (CDFG) [25]. Global transforms are first applied to the entire CDFG, unoptimized controllers are then extracted, and, finally, local transforms are then applied to the individual controllers. The result is a highly-optimized set of interacting distributed controllers. The transforms include aggressive timing- and area-oriented optimizations such as: global communication channel multiplexing and symmetrization; loop parallelism; introduction of global "relative timing"-based simplification; multiplexor pre-selection; sharing of local signals; and the removal

¹These transforms, while at a much higher level of synthesis, are loosely analogous to the powerful transforms of SIS (collapse, extract, etc.) used for design-space exploration in multi-level logic synthesis.

of unnecessary handshaking wires. Several of these optimizations have not been previously formalized or provided by any other existing asynchronous CAD tool, or else in only a limited way.

As a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [35, 25]. A highly-optimized asynchronous implementation by Yun et al. [35] was manually designed, using a number of aggressive timing- and area-based optimizations. Such an implementation cannot be obtained using existing CAD tools. We demonstrate that a very similar optimized design can be simply and automatically derived through systematic application of our new transformations.

The paper is organized as follows. Section 2 gives a basic overview of the approach, including details on the initial CDFG specification, final target architecture, and transformation method. The next three sections describe the three synthesis steps in detail: Section 3 presents the new global transformations, which operate on the CDFG itself; Section 4 explains the extraction of the individual unoptimized controllers; and Section 5 presents several local transformations, which optimize the interaction of a controller and the datapath. Section 6 presents results on a differential equation solver benchmark, and compares with a manual design of Yun. Finally, Section 7 presents conclusions and future work.

2 Overview of Approach

This section presents a basic overview of the synthesis and optimization method. The initial CDFG specification (already scheduled, resource-bound) and the target architecture are first introduced. Then, a brief summary of the synthesis flow is presented.

2.1 CDFG Specification

The synthesis method receives as an input a scheduled and resource-bounded CDFG [25] as shown in Figure 1.

In the figure, all operation nodes bound to the same functional unit are placed within the same column. For example, the three RTL statements $B := 2dx + dx$, $A := Y + M1$, and $U := U - M1$ are all bound to the ALU1 unit. In total, there are four functional units: two ALUs (ALU1 and ALU2, first and last column) and two multipliers (MUL1 and MUL2). Note that the LOOP and ENDLOOP nodes are both bound to ALU2. The START and END nodes are not bound to any functional unit. In addition to the types of nodes in the example, the approach also allows IF and ENDIF nodes.

The CDFG includes arcs that are typically present in synchronous CDFGs, as well as new types of arcs that are specific to the asynchronous case. The former includes the data-dependency arcs (dashed arcs), as well as the control arcs going from and to the LOOP, ENDLOOP, IF, ENDIF, START, and END nodes. In the synchronous case, only these arcs would be used, as well as assignments to time slices that indicate the scheduling of operations. In the asynchronous case, however, this scheduling information must be made explicit: precedence arcs are added between operations bound to the same unit to enforce the schedule (dotted arcs). Finally, the correct order of register writes and reads must be enforced (dashed arcs, like data-dependency arcs). Details will be given below. An operation node in a CDFG may “fire” if all its predecessors have “fired”.

For the proposed approach, the CDFG is assumed to be *block-structured*: the set of nodes between IF and ENDIF nodes, and LOOP and ENDLOOP nodes are considered a

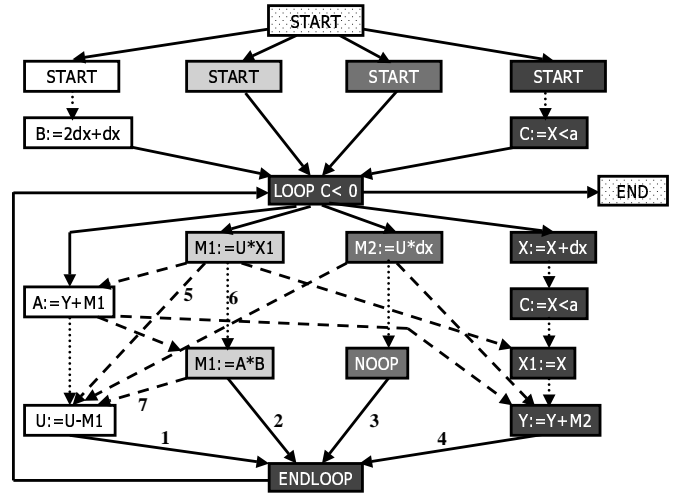


Figure 1. CDFG for DIFFEQ

block. Data dependency arcs, control flow arcs, and register allocation arcs may never cross block boundaries; these arcs can only enter or exit at the block root node (IF or LOOP). (This restriction simplifies the handling of data dependency constraints and register resources, which are used on a per-block basis.)

In the asynchronous case, where operations may take non-fixed (i.e., variable) amounts of time, a legal schedule of operations is obtained by a direct implementation of the constraint arcs. That is, each constraint arc is implemented (in the basic case) by a single wire or *global channel*, which is used to indicate when the receiving CDFG node is allowed to execute.

An operation node $R_1 := R_2 \text{ op } R_3$ has the following constraint arcs that indicate when the operation node may fire and which consequences the firing on other operation nodes has:

1. **Control flow (solid arcs):** control arcs from and to START, END, IF, ENDIF, LOOP and ENDLOOP nodes
2. **Scheduling within a FU (dotted arcs):** scheduling arcs to order the operations assigned to a functional unit
3. **Data dependency (dashed arcs):**
 - incoming arcs from operations that provide its operands R_2 and R_3
 - outgoing arcs to operations that use the result R_1
4. **Register allocation (dashed arcs):**
 - incoming arcs from all operations that use the old register value of R_1 , to avoid early overwriting of R_1
 - outgoing arcs to the next writes to R_2 and R_3 to avoid early writes of R_2 and R_3

Example: A constraint arc that has source node a and destination node b is denoted: (a, b) . In Figure 1, the arc $(\text{LOOP}, A := Y + M1)$ is a control arc, and $(A := Y + M1, U := U - M1)$ is a scheduling arc for ALU1. The arcs $(M1 := U * X1, A := Y + M1)$ and $(A := Y + M1, M1 := A * B)$ illustrate the data dependencies incident to the node $A := Y + M1$. The arc $(M1 := U * X1, U := U - M1)$ is a register allocation constraint arc with respect to U , and it is a data dependency arc with respect to $M1$.

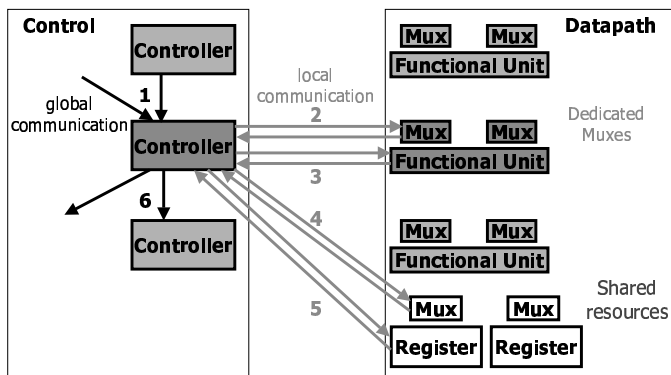


Figure 2. Target architecture

2.2 Target Architecture

The target architecture for the proposed approach is shown in Figure 2. The datapath consists of functional units (each with associated dedicated input muxes), as well as registers (each with an associated input mux).

The distributed control consists of one controller per functional unit. Each controller interacts with other controllers, with its dedicated functional unit and input muxes, and with registers and their input muxes. Note that the registers and their input muxes may be shared by other controllers.

The basic operating protocol is as follows. A functional unit controller waits for a set of “ready” signals from other controllers. These signals indicate that the controller may execute the next RTL statement bound to the corresponding functional unit. Once enabled, the controller then interacts with the datapath according to the figure, i.e. by selecting the appropriate source input muxes, then activating its functional unit, then selecting the appropriate destination register mux, and finally latching the result. As a last step, the functional unit, in turn, signals to other controllers with “ready” signals that it has completed execution of the RTL statement.

Controller-controller communication (using “ready” signals) is implemented using a form of “transition signaling”. Unlike in standard 2-phase transition-signaling protocol [7], where a transition pair on two wires ($req+$ / $ack+$ or $req-$ / $ack-$) completes a communication between sender and receiver, the proposed scheme is even simpler: no acknowledgment wire is used. Thus, controllers communicate with each other by a single transition ($req+$ or $req-$) on one wire. (This scheme is based on the observation that such a ready signal is typically the last event in executing an RTL statement, and no acknowledgment is required.) “Ready” signals serve thus two purposes: incoming signals to a functional unit are “request” signals, and outgoing signals are “done” signals. In contrast, the controller-datapath communication uses a standard 4-phase protocol. In a 4-phase protocol, a standard return-to-zero handshake protocol is used: $req+$, $ack+$, $req-$, $ack-$.

2.3 Synthesis and Optimization Approach

Asynchronous Distributed Control Synthesis

Given: Resource-bound and scheduled CDFG.
Result: Optimized set of interacting controllers.

1. Apply global transformations to optimize controller-controller communication.
2. Extract one AFSM for each functional unit.
3. Apply local transformations for each AFSM to optimize controller-datapath communication.

Before discussing the optimizing transforms, the basic (un-optimized) synthesis method is presented. In an initial CDFG (see Figure 1), all RTL statements bound to the same functional unit (i.e., shown in the same column) will be controlled by a single functional unit controller. A number of constraint arcs run between distinct columns (RTL statements executed by different functional units). Each such constraint arc will be translated into a *global communication channel* between the corresponding functional unit controllers, in the target architecture (see Figure 2). In particular, each communication channel is implemented by a single wire (“ready” signal). Note that while constraint arcs connect individual RTL statements in the CDFG, communication channels connect functional unit controllers (which may implement more than one RTL statement).

Each such functional unit controller is formally extracted from the CDFG (step 2), and can be synthesized using (*extended burst-mode*) finite state machines (AFSM) [29, 34, 16, 14, 15, 33, 28]. Burst-mode is a commonly-used approach to designing Mealy-like asynchronous controllers. This step will be explained in detail below (Section 4). Each constraint (arc) is translated into a single wire (channel). Each CDFG node (e.g., RTL statement) is translated into a series of micro-operations in the controller, where the controller interacts with and sequences the datapath: setting of input muxes, performing operations, writing results, etc.

Using the above approach, however, the resulting implementation may be quite poor. Therefore, this paper introduces optimizing transformations, both global (at the CDFG level, Step 1) and local (on the extracted AFSMs, Step 3), to further improve the design. Global (controller-controller) and local (controller-datapath) transformations are introduced below (Sections 3 and 5). The global transformations have two goals: reducing the number of communication channels between controllers, and improving performance (increasing concurrency, reducing critical path delays). The global transformations are defined in such a way that they preserve the precedence order of the original CDFG.

After the global transformations, one controller per functional unit can then be extracted, as described above (Step 2; see Figure 17 for the burst-mode controller for ALU1). Finally, local transformations improve the controller-datapath (Step 3) protocol for both speed and area: they remove or share wires, increase parallelism of operations, or reshuffle operations to initiate them earlier.

Note that the goal of this paper is to introduce the new set of transformations, which can be used to optimize a system (much like the transforms of SIS for multi-level logic synthesis). In the future, this approach will be extended to develop algorithms and rule-based approaches to better apply

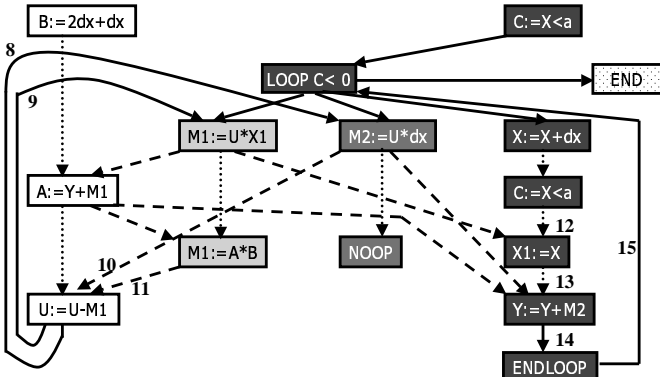


Figure 3. DIFFEQ CDFG after the application of (GT1) Loop Parallelism and (GT2) Dominated Constraints. Note that in order to clarify the presentation, the figure omits the START and END nodes.

these transforms. Also, note that the assumed architecture (e.g. one controller per functional unit) somewhat limits the design space; this restriction will be relaxed in the future (e.g. multiple controllers per functional unit, or one controller for several functional units).

3 Global Transformations: Controller-Controller

In this section, the set of global transformations to optimize controller-controller communication is described.

There are five global transformations: *loop parallelism* (GT1), *removal of dominated constraints* (GT2), *relative timing optimization* (GT3), *merging of CDFG nodes* (GT4), and *communication channel elimination* (GT5). All five transformations re-structure the CDFG.

Starting from the initial CDFG, the first four transformations manipulate constraint arcs in the CDFG and can be applied in any order. GT1 and GT4 modify constraint arcs in the CDFG to allow certain RTL statements to execute in parallel. The result is improved concurrency of the system. In contrast, GT2 and GT3 remove constraints in the CDFG that are not necessary for the correct operation of the distributed control. Since each constraint arc will become a channel, these two transforms therefore reduce the total number of communication channels. Reducing the number of channels may also simplify global system wiring and routing.

After the first four transformations have been applied, each remaining constraint arc is assigned to a distinct communication channel. Each communication channel connects the two functional unit controllers that correspond to the two CDFG nodes that the arc connects.

The final and powerful transformation then aims at eliminating as many communication channels between controllers as possible. The first and basic technique is *channel multiplexing* (GT5.1). This technique eliminates channels by sharing channels when there are multiple channels connecting the same set of units. The second and third techniques *concurrency reduction* (GT5.2) and *channel symmetrization* (GT5.3) both start from configurations where channel multiplexing is not directly applicable, and re-structure the communication channels so that channel multiplexing can be applied.

3.1 GT1: Loop Parallelism

The goal of the “loop parallelism” transform is to improve concurrency of the distributed control. The transform re-structures the CDFG to allow more parallelism between *successive* iterations of a loop. More parallelism is achieved by loosening the synchronization through the ENDLOOP node and replacing it by more localized synchronization constraints.

Compare Figures 1 and 3. In Figure 1, all four functional unit controllers are synchronized with an ENDLOOP node – by the arcs labelled 1 through 3. In contrast, in Figure 3, these arcs to the ENDLOOP-node are removed, and replaced by more localized synchronization constraints: the two *backward arcs* 8 and 9. As a consequence, greater loop-level parallelism is achieved. Note that backward arcs are special arcs in the sense that they are ignored during the first execution of a loop body. Effectively, a backward arc is a pre-enabled constraint for the first iteration of a loop.

The “loop parallelism” transform consists of four steps in sequence.

Parallelize_Loop(LOOP,ENDLOOP):

- **A. Remove synchronization at ENDLOOP.** The goal is to allow the overlap of successive loop body executions. The solution is to remove all arcs in the CDFG that are pointing to ENDLOOP; only the FU scheduling arc that connects ENDLOOP to its predecessor node in the FU scheduling – of the functional unit ENDLOOP is allocated to – remains.

Compare again Figures 1 and 3. In step A the three arcs labelled 1, 2, and 3 are removed. The FU scheduling arc 4 remains.

- **B. Add backward arcs: loop body variables.** In the unoptimized case, the loop body includes data and register dependency constraints to avoid early reads and writes of registers (cf. Section 2.1). The goal of this step is to add constraints to extend these constraints across the loop boundary. For each variable in the loop body, backward arcs from all last instances (one write or multiple parallel reads) of the variable to the first instances (one write or multiple parallel reads) are added.

In the example, step B adds the two *backward arcs* 8 and 9.

- **C. Add backward arcs: loop variable.** In the unoptimized case, the synchronization at ENDLOOP guarantees that the loop variable is updated before the LOOP-node examines it. In the optimized case, this requirement must be enforced explicitly. Thus, a backward arc from the last write of the loop variable in the loop body to the LOOP-node is added.

In the DIFFEQ example, step C does not add any constraint. The candidate arc from the node $C := X < a$ to the LOOP-node is implied by the path of constraint arcs 12, 13, 14, and 15. Thus, the candidate arc ($C := X < a, LOOP$) is a *dominated* constraint (cf. Section 3.2), and therefore not added.

- **D. Limit parallelism.** In the unoptimized scheme, global controller-controller communication is implemented by transition signaling without explicit acknowledgments (cf. Section 2.2). Effectively, there is always a chain of other events that provides an acknowledgment. After removing the arcs pointing to ENDLOOP in step A, this requirement no longer holds for arcs from the LOOP-node

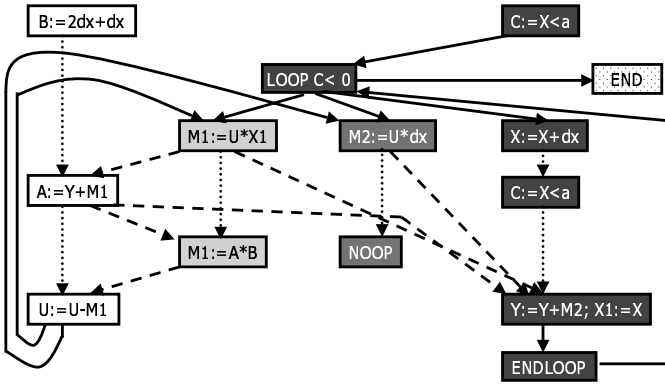


Figure 4. CDFG after GT3 and GT4

to the first node of a functional unit in the loop. The requirement is reinstated by adding arcs from the first node of each functional unit in the loop to the ENDLOOP-node. In effect, this limits parallelism to two consecutive iterations of a loop: the added arcs express that the next loop iteration can only be started after all functional units have completed the first operation in the current loop.

In the example, step D does, like step C, not add any constraints. The first CDFG nodes of each functional unit – ALU1: $A := Y + M1$, MUL1: $M1 := U * X1$, MUL2: $M2 := U * dx$, ALU2: $X := X + dx$ – is already connected to ENDLOOP through a path of constraints.

In some cases, there is an issue whether it is *safe* to apply the loop transform. This issue concerns the final exiting from the loop. After applying the loop transform, the LOOP-node examines the loop variable *while* other functional units are still executing statements of the previous iteration. Hence, if the loop condition is false, the LOOP-node “exits” the loop, by sending “ready” signals to other controllers, possibly *before* the last iteration of the loop is finished. In this scenario, the transform is safe as long as a system timing constraint is satisfied: all units complete their operation before needed. If this constraint is not satisfied, then special constructs must be added to enforce that the loop is only exited until after the last iteration is completed. However, this topic is beyond the scope of the paper.

3.2 GT2: Removal of Dominated Constraints

The goal of the transformation is to remove constraints that are *implied* by other constraints. A constraint arc from node a to node b is implied if there is a path of other constraints starting at node a and ending at node b . More formally, the constraint is removed if it contained in the transitive closure of all other constraints.

Consider constraint arc 5 in Figure 1. This constraint is implied by the path consisting of the two constraints 6 and 7. Thus arc 5 can be removed, and therefore simplifying the global inter-controller communication – because arcs become channels (cf. Section 2.3), and so no channel is needed.

3.3 GT3: Relative-Timing Optimization

In asynchronous design, “relative timing” refers to the exploitation of knowledge about the relative occurrence of events in order to simplify design. Relative timing assumptions have been effective [32, 11]. However, these approaches were limited to single controllers.

GT3 extends the use of relative-timing information to optimize interacting controllers. In the unoptimized case, a controller must wait for a set of “ready” (“request”) signals from other controllers before it starts executing the current RTL statement. However, in some cases, the same controller always signals last. In such cases, the “ready” signals from “faster” controllers can be removed, and the controller only waits for the “slowest” one.

Both GT2 and GT3 remove constraint arcs in the CDFG that are not necessary for the correct operation of the system. However, while GT2 removes arcs based on the actual ordering of events enforced by the CDFG itself, GT3 removes arcs based on analyzing user-supplied timing information.

Consider Figure 3. There are two constraint arcs from other controllers to $U := U - M1$, labelled 10 and 11. The former gets enabled after one computation – $M2 := U * dx$ – while the latter gets enabled after three computations – $M1 := U * X1$, $A := Y + M1$, $M1 := A * B$. Thus, the latter constraint arc (11) is “slower” under most assumptions. Hence, the former arc (10) is deleted in Figure 4. A detailed timing analysis must be performed to determine where this transformation can be applied: it must be verified that the removed constraint arc is under no execution path the last one to happen.

3.4 GT4: Merging of Assignment Nodes

Merging assignment nodes is aimed at improving the speed of a functional unit controller. In the unoptimized scheme, each CDFG node is assigned to a functional unit. However, assignment nodes, i.e. $R_i := R_j$, simply examine and write registers. Thus, assignment nodes do not use the functional unit. An assignment node can therefore be executed in parallel with the preceding or succeeding RTL operation assigned to the same functional unit.

Compare Figures 3 and 4. In Figure 3, the two nodes $Y := Y + M2$ and $X1 := X$ are both assigned to the ALU2 functional unit. Since the node $X1 := X$ does not use the ALU2 functional unit, the assignment can be executed *in parallel* with executing the RTL-node $Y := Y + M2$. Thus, the two nodes are merged into one node $Y := Y + M2; X1 := X$ in Figure 4.

Before merging an assignment node with an adjacent node, a check is required to ensure no deadlock. In particular, there must not be a path of constraints between the two nodes to be merged – except for the trivial FU scheduling arc that connects both nodes. When merging assignment nodes, the set of incoming and outgoing arcs of the resulting node is the union of the sets of incoming and outgoing arcs of the two original nodes to be merged. If such a path existed, one of the outgoing arcs of the new node would be a predecessor to one of the incoming arcs, and thus a prerequisite for the node to “fire” could never be asserted, resulting in deadlock. However, in practice, it is rare that an assignment node cannot be merged with at least one of its adjacent nodes.

3.5 GT5: Communication Channel Elimination

After the first four transformations GT1 through GT4 have been applied, each remaining constraint arc is assigned to a distinct communication channel. Each communication channel connects the two functional controllers that correspond to the two CDFG nodes that the arc connects (see Figure 5).

The goal of “communication channel elimination” is to delete as many communication channels between controllers as possible. The first and basic technique is (GT5.1) *channel multiplexing*. This transform eliminates channels by shar-

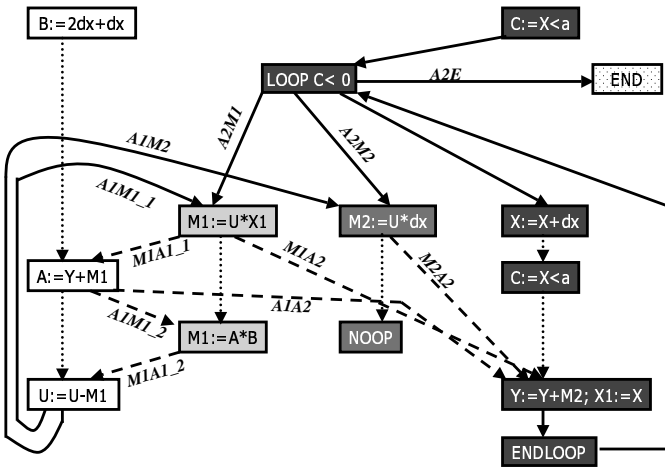


Figure 5. Communication channels after GT1 - GT4.

ing channels when there are multiple channels connecting the same set of units. After multiplexing has been applied, the two different events on the two channels become different phases on the shared channel. The second and third techniques (GT5.2) *concurrency reduction* and (GT5.3) *channel symmetrization* both start from configurations where channel multiplexing is not directly applicable, and re-structure the communication channels so that multiplexing can be applied, by adding and replacing arcs.

Figure 6 gives a summary of the significant impact of the three GT5 transforms on simplifying communication. On the left side the communication channels before the application of GT5 transforms is shown, and on the right side the communication channels after several GT5 transforms – multiplexing, concurrency reduction, symmetrization – have been applied. In the example, GT5 transforms reduce the number of channels from ten to five, which includes two multi-way channels. The result is much simpler inter-controller communication. The CDFG corresponding to the left side of the figure is shown in Figure 5, and the CDFG corresponding to the right side is shown in Figure 7.

Now, each of the three transforms GT5.1 through GT5.3 is explained.

• **GT 5.1: Channel Multiplexing**

The idea of “channel multiplexing” is to share communication channels to reduce the number of channels. Multiplexing can be applied to channels that connect the same functional units and that are never concurrently active.

Consider Figure 8, where a CDFG fragment is shown on the left side and the corresponding controller structure is shown on the upper right side. The CDFG fragment contains two nodes bound to ALU1 and two nodes bound to MUL1. There are four arcs between the two functional units, and initially each one is implemented by a separate communication channel. The result is two channels running from ALU1 to MUL1 and two channels running from MUL1 to ALU1.

“Multiplexing” the two channels running from ALU1 to MUL1 leads to sharing one communication channel

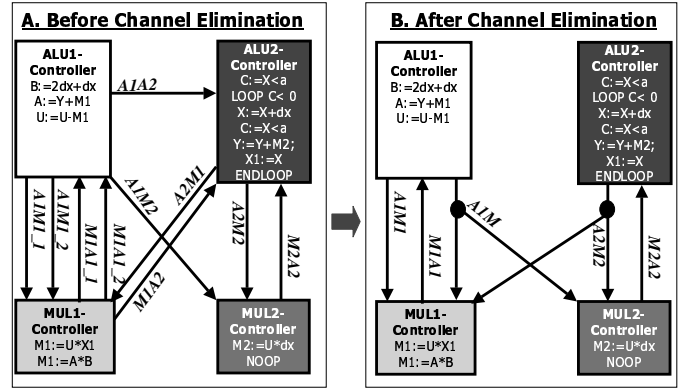


Figure 6. GT5: Channel Elimination for DIFFEQ Example.

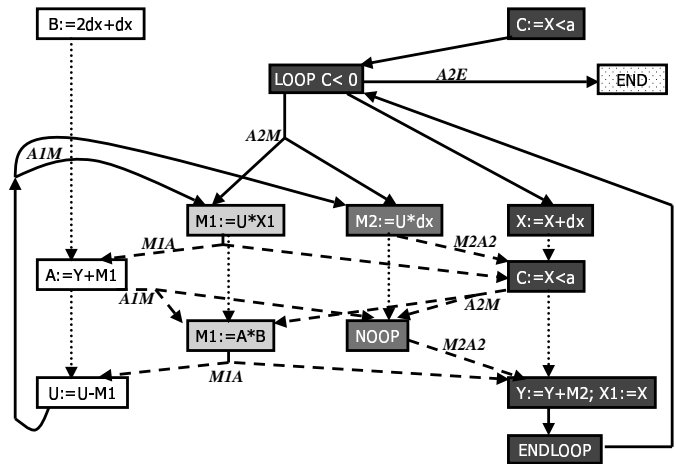


Figure 7. After Channel Elimination.

(and thus a wire, since each channel becomes a wire, cf. Section 2.3) from ALU1 to MUL1 (bottom of right side). Similarly, the two channels running from MUL1 to ALU1 are multiplexed. As a consequence, the number of channels is reduced from four to two.

• **GT 5.2: Concurrency Reduction**

“Concurrency reduction” starts from a configuration where channel multiplexing is not directly applicable, and re-structures constraints so that multiplexing can be applied. The transform replaces a *simple* constraint from a node *a* to a node *c* by a *chain of two other constraints*: a constraint from *a* to *b*, and a constraint from *b* to *c*. Thus, the additional hub may reduce the concurrency of the system (it possibly delays the start of executing node *c*), but it eliminates a channel by re-using an existing channel.

The goal of the transform is to apply it to non-critical constraints, and to replace a constraint in such a way with a chain that the resulting two constraints can be multiplexed with existing constraints. (If any of the two constraints already exists in the CDFG, then it is not added.)

Consider the CDFG in Figure 9. The constraint arc 4_{old}

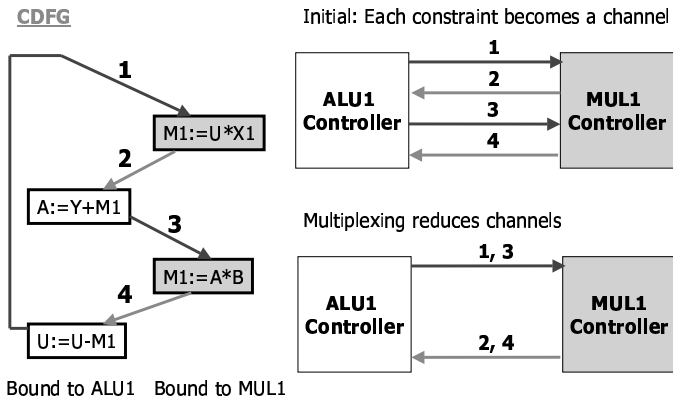


Figure 8. GT5.1: Multiplexing Constraints on Communication Channels

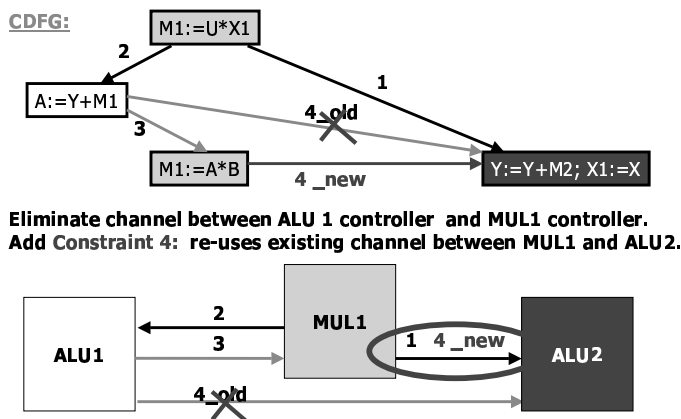


Figure 9. GT5.2: Concurrency Reduction

is replaced by the existing arc 3 and a new arc 4_{new} . The new arc can be multiplexed with the arc 1 since both arcs connect the same functional units. Hence, the number of communication channels is reduced, and the overall communication structure is simplified: there no longer is a direct communication channel between the leftmost (ALU1) and rightmost (ALU2) controllers, see Figure 9 (bottom).

• **GT 5.3: Channel Symmetrization**

Like GT5.2, “symmetrization” starts from a configuration where “channel multiplexing” is not directly applicable. Constraints are added to the CDFG so that multiplexing becomes possible.

Unlike other transforms, the goal of symmetrization is to create *multi-way* channels. A multi-way channel connects a single CDFG source node to multiple CDFG destination nodes. Each node must correspond to a distinct functional unit. Event sent by the “sender” are seen by all receiving functional units. Given two sets of channels that have the same sending functional unit, but have overlapping but *not identical sets* of receiving functional units, the idea of the transform is to first make the receiving sets symmetric, by “safe addition” of arcs in

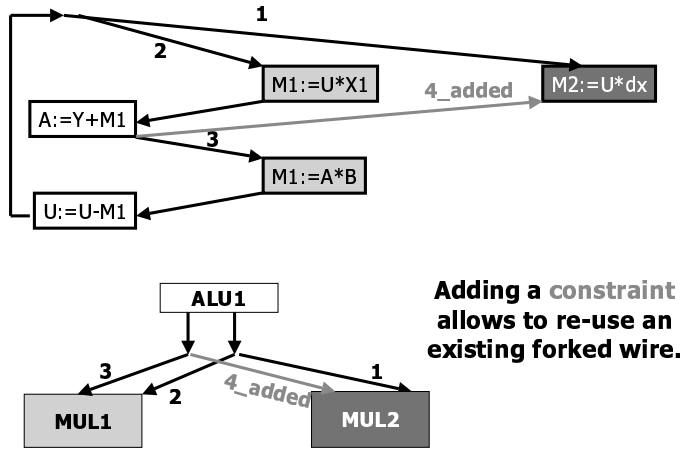


Figure 10. GT5.3: Channel Symmetrization

the CDFG. Next, each set is transformed to a multi-way channel. Finally, the pair of channels is multiplexed.

Figure 10 visualizes the symmetrization transform. Consider the three constraints 1, 2, and 3 in the CDFG, where 1 and 2 form a set of channels, and 3 is a singleton set. The first set connects ALU1 to MUL1 and MUL2, while the second set connects ALU1 to MUL1. The transform makes the two sets symmetric by adding constraint 4_{added} to the CDFG, and also to the singleton set. The two sets become two multi-way channels connecting ALU1 to MUL1/MUL2 (see bottom of the figure). These two channels are then multiplexed.

When symmetrizing channels, arcs must be added in a “safe” way to ensure no deadlock. Deadlock can arise when an added constraint forms a circular dependency which prevent units from executing. To detect deadlock, path analysis is performed on the CDFG for illegal cycles. Consider the addition of an arc from node a to b . This arc must lie within a single block due to the block-structuring requirement (Section 2.1). Within this block, a and b are partially ordered (in the initial CDFG): either a precedes b , a follows b , or a and b are unordered. If a follows b , no regular arc can be added, since a deadlock would result.² If deadlock occurs, symmetrization cannot be applied.

There is one small technical issue that must be considered: only arcs of the same “type” (regular vs. backward) can be combined into a multi-way channel. Recall that backward arcs can be thought of as “pre-enabled” (to set up the first iteration), while regular arcs are not. If the two types are combined in one multi-way channel, the backward arcs cannot be uniquely pre-enabled.

4 Individual Controller Extraction

After the global transformations have been applied, an asynchronous finite state machine (AFSM) is extracted for each functional unit controller. The extraction algorithm is a direct deterministic translation from the CDFG (see Figure 7) into

²Backward arcs, which enforce data dependencies *between* loop iterations, will not cause deadlock.

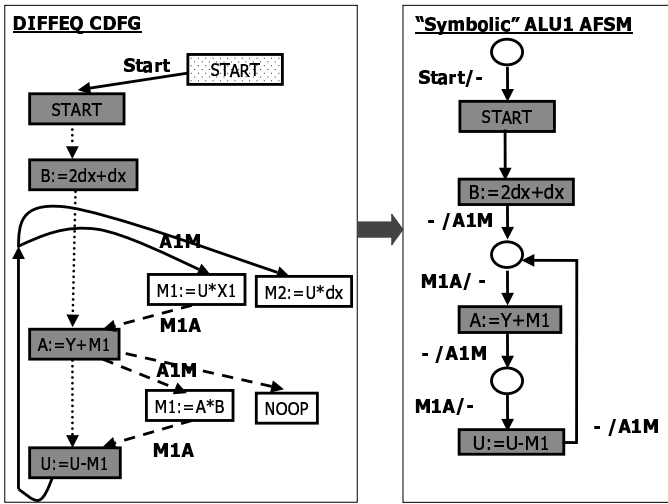


Figure 11. Burst-Mode Extraction

asynchronous Burst-Mode Controllers. Before explaining the four-step algorithm, the targeted extended Burst-Mode AFMSMs are briefly reviewed. For details refer to [29, 34, 16, 14, 15, 33, 28].

4.1 Background on Burst-Mode FSMs

In a Burst-Mode AFMSM, as shown in Figure 17, state transitions occur when a specified *input burst* (set of variables before the “/”) has been received. During the transition to the next state, the corresponding *output burst* (set of variables after the “/”) is generated. BM specifications must obey two properties. First, the so-called *maximal set property* stipulates that no arc leaving a state may possess an input burst that is a subset of any other arc leaving that state. This property guarantees that for each state, the machine can unambiguously decide whether to follow a transition or to wait for additional inputs. Second, in a given state, *only a specified input burst may occur*, i.e a burst that corresponds to an outgoing arcs; other input combinations are forbidden.

Extended Burst-Mode AFMSMs (XBM) allow two important extensions. First, inputs may arrive early, and thus more concurrency is allowed: While in a Burst-Mode AFMSM, only input signals corresponding to the current input burst may arrive, in an extended Burst-Mode AFMSM, input signals that may arrive early are attached to previous bursts and marked with an asterisks (*directed don't care*). However, each input burst needs to contain at least one signal that cannot arrive early (*compulsory signal*). Second, extended burst-mode AFMSMs allow sampling of level-sensitive variables in input bursts (called *conditionals*), indicated by brackets, e.g. $[C+]$, $[C-]$. The value of a conditional in an input burst is inspected only after all transition signals of the burst have occurred. Thus, this is a set-up condition: conditionals must arrive and be a stable before any compulsory event arrives.

4.2 Burst-Mode Extraction: Overview

The extraction method is based on a direct translation scheme for each CDFG node. Consider Figure 11, which illustrates the extraction of the ALU1 controller. On the left side of the figure a partial CDFG that includes all nodes bound to ALU1 is shown, and on the right side is the corresponding “symbolic” AFMSM. The “symbolic” AFMSM includes one symbolic

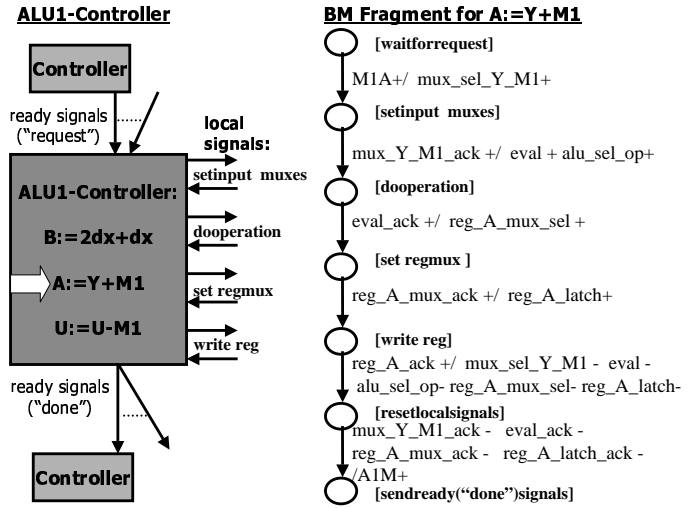


Figure 12. BM Expansion of RTL-node $A:=Y+M1$

node for each CDFG node. Each symbolic node is then expanded into a Burst-Mode fragment, implementing the operation.

The basic idea of translating a CDFG node – *expanding the corresponding symbolic node* – into a Burst-Mode fragment can be illustrated by using an RTL node as an example. In Figure 12, the left side shows the ALU1-controller, and its communication with other controllers and its datapath. The controller controls three RTL statements ($B := 2dx + dx$, $A := Y + M1$, and $U := U - M1$), and it is waiting to execute the RTL node, $A := Y + M1$. The right side of the figure shows the Burst-Mode fragment corresponding to the RTL node $A := Y + M1$. This fragment will be explained in detail below.

A BM fragment for an RTL node implements the basic protocol: (a) wait for a set of ready signals (“requests”) from other controllers, (b) perform the datapath operation, and finally (c) send “ready” signals (“dones”) to other controllers to indicate that it has finished executing the RTL statement. “Ready” signals are single transitions on wires. In contrast, local communication with the datapath is based on a 4-phase protocol: $req+$, $ack+$, $req-$, $ack-$.

The given BM fragment in Figure 12 consists of a series of six state transitions to implement the microoperations: (i) *wait for request and set input muxes*, (ii) *do operation*, (iii) *set register mux*, (iv) *write register*, (v) *reset local signals*, and (vi) *send done signals*. Each of the microoperations (i) through (iv) is done by a $req+$ and $ack+$ pair, the first half of a 4-phase handshake. In the figure the microoperation labels are placed between the corresponding $req+$ and $ack+$ pair. In (v) all req/ack -pairs are then re-set to 0 in parallel ($req-$, $ack-$). Microoperation (ii) “do operation” includes two tasks: (a) selecting the operation to be performed by the FU – FUs such as ALUs can execute different operations, and (b) starting the evaluation of the functional unit.

After describing the main ideas of BM extraction, the actual algorithm is summarized as follows. The BM extraction method is implemented by an algorithm containing four steps. First, each CDFG node is translated into a BM frag-

ment³. Second, BM fragments are stitched together to obtain a near-complete specification for the controller. Third, signal phases to global communication signals are assigned. There is one more final step to ensure that the BM specification may accept early requests. Each basic stitched template assumes all “ready” (“request”) signals arrive just when needed. However, in reality, the system may be more concurrent and thus the fourth step modifies the BM specification to “back-annotate” the early arrival of requests. The generated Burst-Mode controller for ALU1 is shown in Figure 17.

Now each of the four steps of the algorithm is explained in turn.

4.3 Step 1: Translation of CDFG Nodes into BM Fragments

In Step 1, each CDFG node is translated into a BM fragment by using translation templates. There is one translation template for each of the different CDFG nodes: regular RTL-node, IF-node, ENDIF-node, LOOP-node, ENDLOOP-node, START-node. Before explaining the different templates, the common issue of generating global ready signals is presented.

Global “Ready” Signals

All BM fragments include global ready signals to implement the basic protocol (cf. Section 2) for a CDFG node. A *simplified view* is as follows: (i) wait for “request” ready signals, (ii) interact with local datapath (if necessary), and (iii) send “done” ready signals to waiting controllers. In addition, the BM fragments must include a state transition [after (iii)] to the BM fragment corresponding to the CDFG node that is to be executed next by the functional unit. In fact, the next CDFG node may not be unique: the next CDFG node is determined by *which* “request”-signals arrive from other controllers.

The solution is to break with the simplified view and to generate BM fragments in a reversed way: (a) interact with datapath (corresponds to ii), (b) send “done” ready signals (iii), and (c) wait for ready signals to *decide* which is the next CDFG node to be executed. In (c) state transitions to all possible next CDFG nodes, i.e. their corresponding BM fragments, are generated (see Figure 13).

Templates

There is one translation template for each of the different CDFG nodes.

The template for RTL-nodes is shown in Figure 13. The local communication for an RTL node was already explained in 4.2. The global communication in the template is reversed: the global “request” ready signals are at the bottom of the template, as explained in the previous section. As a further optimization, when BM fragments are generated, only mux select signals that are necessary are included in the fragment. For example, if a register is only written by one functional unit, no mux is necessary, and thus no mux signal is generated.

Control-flow CDFG nodes (IF, ENDIF, LOOP, ENDLOOP, START) are translated by modified templates. There are two major differences. First, no local signals need to be generated. Second, in the case of LOOP and IF-nodes transitions for the two possible values of the level-sensitive decision variable need to be generated. Refer to Figure 14 which shows the Burst-Mode template for the LOOP node: Once the ready signals have arrived, the loop variable is inspected. If the value is 1, the “true” ready-signals are sent, and otherwise the “false” ready-signals.

³Steps 1 and 4 may introduce features only available in *extended* Burst-Mode

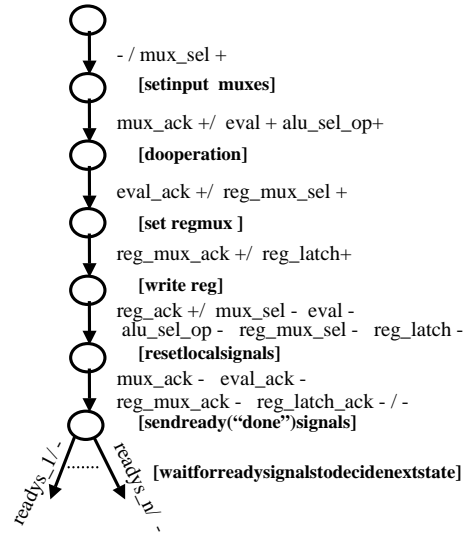


Figure 13. BM template for RTL node

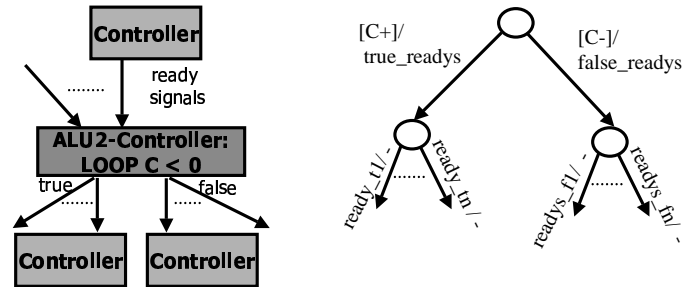


Figure 14. BM template for LOOP node

4.4 Step 2: Stitching of BM Fragments

After Step 1 is done, the BM fragments are stitched together. If BM fragments were simply “appended”, this would violate the class of BM/XBM specifications, because the first transitions of the templates (see Figures 13 and 14) have input bursts, which are empty or do not include a compulsory signal (cf. Section 4.1). The solution is to merge the first transition of a fragment with the last transition of all previous fragments. There are two similar subcases, one for RTL BM fragments, and one for IF/LOOP BM fragments.

Stitching a BM fragment for a regular RTL node merges the empty-input-burst of its first transition with all immediately preceding transitions. This process is visualized in Figure 15. In part A (bottom) the first transition of an RTL node fragment (labelled $-/o4$) is shown. Suppose there are three preceding fragments, corresponding to CDFG nodes that immediately precede the RTL node in some execution path. A valid specification is obtained by merging $-/o4$ with each of the transitions $i1/o1$, $i2/o2$, $i3/o3$. The result is shown in Figure 15B.

Stitching LOOP and IF-node BM fragments with preceding BM fragments merges the two transitions with conditionals (see Figure 14) with all preceding transitions. The LOOP and IF-node BM fragments generate input bursts that only include conditionals, i.e. these bursts do not include a compulsory signal. These input bursts would represent an XBM

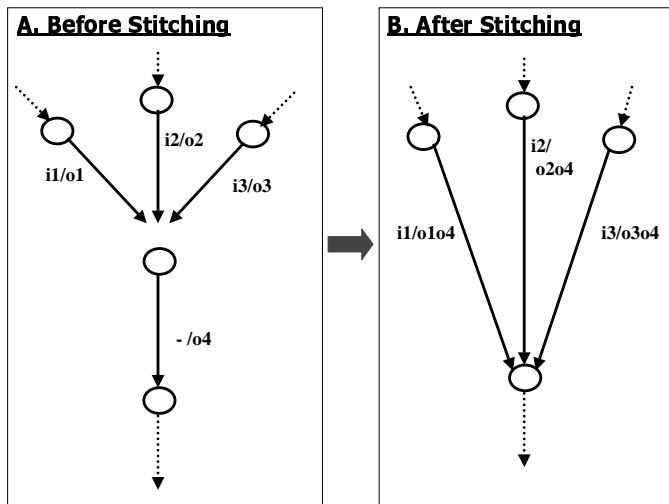


Figure 15. Stitching an RTL-node BM Fragment with Preceding Fragments.

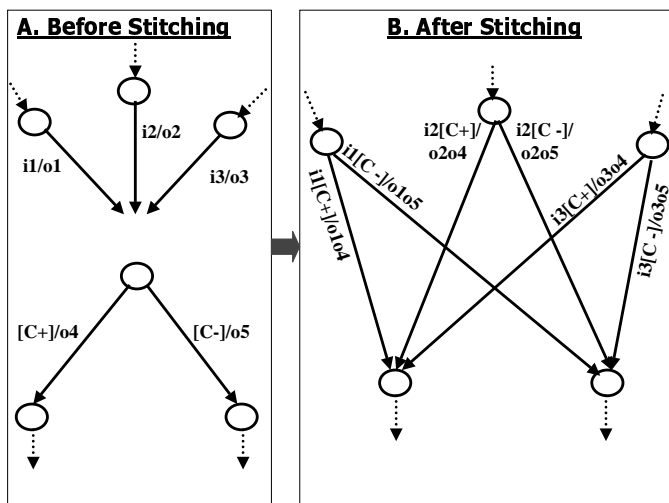


Figure 16. Stitching an IF/LOOP BM Fragments with Preceding Fragments

violation (cf. Section 4.1). The solution is to merge each of the two transitions with conditionals with all preceding transitions. This process is visualized in Figure 16.

4.5 Step 3: Assignment of Signal-Phases to Global Ready Signals

Steps 1 and 2 assign signal phases to local signals but leave global communication signals – “ready”-signals – untouched. Consider Figures 13 and 14 which include “symbolic” ready-signals without signal phases. The goal of Step 3 is to assign signal phases to the “ready”-signals. That is, each symbolic “ready”-signal *ready* will become *ready+* or *ready-*. Assigning signal phases to “ready-signals” is a straightforward encoding process and may involve unrolling the AFSM, i.e. replicating parts of the AFSM. Consider Figure 17, which shows the final Burst-Mode controller after phase assignment.

There are two *M1A* transitions: State 10 waits for an *M1A+* transition whereas state 18 waits for an *M1A-* transition. In this example, signal phase assignment did not involve any replication.

4.6 Step 4: Back-Annotation To Allow Early Requests

In some cases, the AFSMs generated by Steps 1 to 3 are oversimplified: they ignore some of the possible concurrency of the system. In particular, each basic stitched template assumes all global “request” ready-signals arrive *just when needed* (in the first input burst). In reality, the system may be more concurrent: some requests may arrive during earlier operations. The goal of Step 4 is to modify the burst-mode controller specifications to “back-annotate” the early arrival of requests.

A Burst-Mode AFSM is back-annotated by augmenting transitions with directed don’t care signals. Directed don’t care signals are a feature of extended Burst-Mode specifications (cf. Section 4.1). The following approach is applied to make the BM robust to early arriving global ready (“request”) signals: For each transition in the BM specification, ready signals that may arrive early, i.e. during the transition but which are not part of the input burst of the transition, are added as directed don’t cares to the input burst of the transition.

A reachability analysis is used to determine which ready signals may arrive early. This step is performed only once for the entire system. First, only the global ready-signals between BM controllers are considered. At any time, the values of these signals indicate the global state of the system. Second, a state space exploration of the system is performed to determine which global states are reachable. Finally, the resulting reachability information is then used to backannotate each transition in each BM specification with directed don’t cares (to indicate potential early arrival of ready signals).⁴

5 Local Transformations: Controller-Datapath

The outcome of “individual controller extraction” (Section 4) is a BM specification for each functional unit controller. In particular, the global interaction between controllers (“ready signals”) is now fixed.

Local transformations can be applied to each of the individual controllers. The transformations aim at optimizing the handshake protocol between a functional unit controller and both its dedicated and shared datapath. In the unoptimized approach, communication between the controller and its datapath uses a series of 4-phase standard handshakes with req/ack wires: *req+*, *ack+*, *req-*, *ack-*. The goal is to reduce both the critical path delay and area of each controller.

There are five local transformations. *Move-up* (LT1) issues output signals earlier to shorten the critical path when safe. *Move-down* (LT2) delays generation of output signals to provide opportunities for applying further local transforms. *Mux-Preselection* (LT3) removes the selection of source input muxes or register muxes from the critical path. *Remove Acknowledgments* (LT4) removes the ack wire that runs from a datapath unit to its functional unit controller when safe. *Signal sharing* (LT5) merges two distinct local wires into one forked wire.

⁴Details on this step are omitted due to the length of the paper - the current implementation does not perform this step.

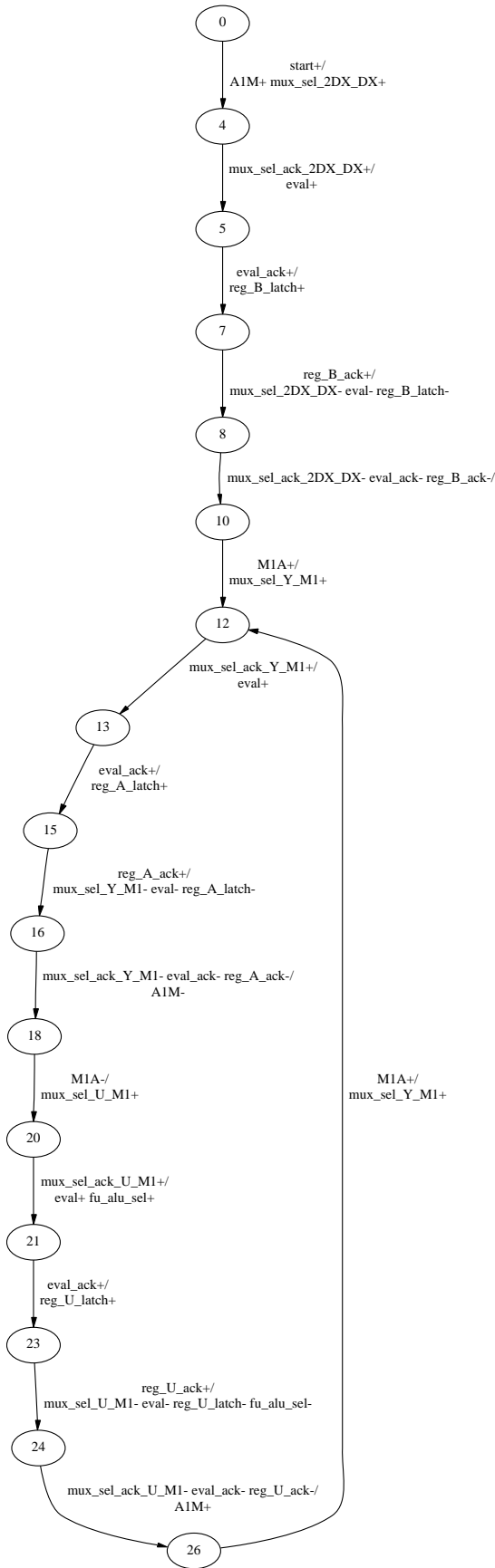


Figure 17. ALU1 controller with datapath signals.

LT1 and LT4 improve the protocol based on *local timing-based optimization*. Local timing-based optimizations are similar to the global timing-based optimizations. In fact, in the local case, more drastic optimizations due to the locality of the wiring may be possible. In contrast, LT2 and LT3 are not based on timing considerations. They explore alternatives to the strict ordering of events in the protocol between the functional unit and its datapath, introduced by the template-based translation (cf. Section 4).

5.1 LT1: Move-Up

The transformation “move-up” safely moves an output signal of the Burst-Mode controller to an earlier burst. The output can be either a local signal (triggering a microoperation) or a global ready (“done”) signal. The primary aim of the transform is to reduce the critical path delay to start the execution of an operation. A secondary aim of the transform is to provide further opportunities for applying other transforms. When “move-up” is applied to a global “done” signal, it effectively shortens the execution time of the current RTL operation.

As an example, consider the transform applied to the final *A1M* ready (“done”) signal in the ALU1 controller, as shown in Figure 17. The signal *A1M* + indicates the completion of the current RTL operation, and is issued after latching *B* has been acknowledged (*reg_U_ack*), and even re-set (see transition from state 24 to 26). Figure 18 shows the modified controller after the move-up transform. Latching the result (*reg_U_latch*) and sending a “done” signal to other controllers (*A1M* +) are now performed in parallel (see transition from state 21 to 24).

Move-up modifies the control specification as follows. The transform is given an output signal and the “destination burst” – given by an input signal. This input will serve as the new “trigger event” for the output signal. In the actual algorithm, the output signal is moved step-wise, i.e. one transition up at a time until the specified input signal is found. If, during this process, a state with more than one in- or outgoing transition is encountered, the signal is replicated and moved along each of these transitions.⁵

The transform can only be safely applied if certain timing requirements are satisfied. In particular, it must be analyzed and verified that the moved signal does not enable an operation (either local or global) before it is safe: an early-enabled global RTL operation must have correctly updated operands in shared registers, and early-enabled local microoperations must be safe under relative-timing assumptions.

5.2 LT2: Move-Down

The “move-down” transformation moves output signals that are not on the critical path to a later burst. The motivation is that moving signals to later bursts provide opportunities for the application of the “signal sharing” transform (LT5). “Move-down” is typically applied to the re-set phases of local signals (*req*–, or *ack*–). The transformation is implemented analogously to “Move-up”; instead of moving a signal to an earlier burst, the signal is moved to a later burst.

5.3 LT3: Mux-Preselection

Selection of muxes is often on the critical path for a system. Traditionally, muxes are selected on demand, that is, at the time they are needed. The idea of “Mux-Preselection” is to break with that concept and pre-select muxes early.

⁵Due to space limitation, details on this step are skipped.

For a functional unit executing the current RTL operation, it is typically deterministic which RTL operation is next, so its controller can start *pre-selecting* the muxes for the next operation at the end of the *current* RTL operation’s execution.

Consider Figure 17, where mux selection $mux_sel_Y_M1+$ is performed after $M1A+$ is received, see transition from state 10 to 12. In contrast in Figure 18, the selection is made at the end of the previous RTL statement (see transition from state 5 to 8), i.e. before $M1A+$ is received, thus reducing the latency for the RTL execution.

“Mux-preselection” is an important special case of “move-up” (LT1), but highlights a new aspect. The main idea of “Move-up” was to move signals to earlier bursts within the state sequence corresponding to the datapath operations of the *current* RTL operation. “Mux-preselection” goes one step further. It “moves-up” the mux select signal, effectively merging it with the datapath operations of the *previous* RTL operation.

5.4 LT4: Remove Acknowledgments

The local transform LT4 removes acknowledgment wires that are not essential for the correct behavior of the controller. In the unoptimized approach, communication between the controller and its datapath uses a 4-phase standard handshake protocol: $req+$, $ack+$, $req-$, $ack-$. The transform replaces the req/ack wire pair by just a req -wire whenever possible.

User-supplied timing information is used to verify that the controller operates correctly once the acknowledgment wire has been deleted. In the simple case, the transformation leaves events in order, but simply deletes acknowledgments. It must be verified that removing the acknowledgment does not change the arrival order of events at any datapath or control unit. This scenario can be applied after “Mux-preselection” (LT3), to delete unnecessary acknowledgments after pre-selecting and resetting the mux.

In a more aggressive scenario (see below), the transform may result in sequenced operations being merged into one step. In this case, more careful timing analysis is required to ensure that the microoperations can complete correctly.

The LT4 transform modifies the BM control specification by simply removing the the ack signal from each input bursts in which it appears. If this operation leaves an input burst empty, which is not allowed in BM specifications, a post-processing step is applied which merges such a transition with its preceding transition(s). This step is identical to the one explained in Section 4.4.

Compare Figures 17 and Figure 18. In the latter, acknowledgments from registers and muxes have been removed.

5.5 LT5: Signal Sharing

Finally, “Signal sharing” aims at reducing the number of output signals of a controller. Reducing output signals is achieved by merging distinct control wires into a single forked wire. The forked wire then activates several datapath operations concurrently.

Signal sharing can be applied to two wires that carry the same signal value at all times. That is, two wires can be merged into a single forked wire if their corresponding signals appear in precisely the same set of output bursts in a BM specification (i.e. across all RTL statements executed by the controller).

Consider Figure 18. In this example, the two signals reg_U_latch and $mux_sel_Y_M1$ are always asserted and de-asserted in the same transitions. Therefore, the two local

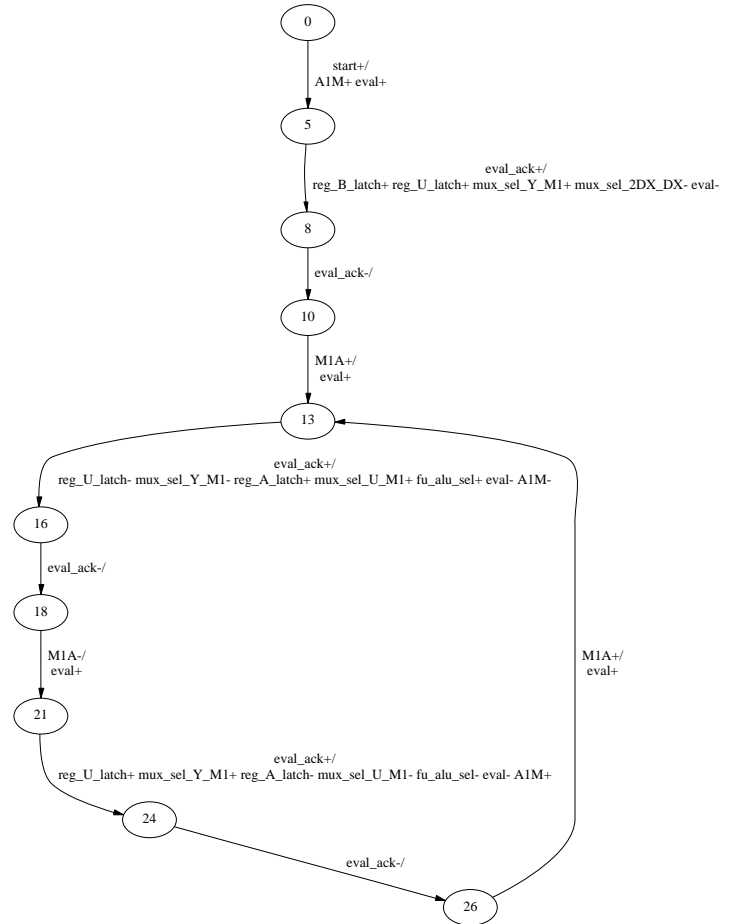


Figure 18. ALU1 controller after LT1 through LT4.

communication wires for latching and selecting can be collapsed into one, and a single forked wire used to control the two components: register U , and the input muxes to the functional unit that computes $Y + M1$. The result is indicated in Figure 19, where $reg_USSmux_sel_Y_M1$ now corresponds to a single forked wire.

6 Experimental Results

A prototype version of the presented method has been implemented in C++. As a case study, the method is applied to the well-known *differential equation solver* synthesis benchmark. A highly-optimized implementation was manually designed by Yun et al. [35]. Their circuits used many aggressive optimizations which have been inaccessible to existing asynchronous CAD tools.

Our automated tool has been applied to this example in three experiments, as shown in Figure 20. The *unoptimized* controllers were generated directly from the original CDFG with no global or local transformations applied; the *optimized-GT* controllers only after the application of global transformations, and the *optimized-GT-and-LT* controllers after application of both sets of transformations. The final specification of our ALU1 controller for the *optimized-GT-and-LT* case is shown in Figure 19.

	#comm. channels	ALU1		ALU2		MUL1		MUL2	
		#states	#trans	#states	#trans	#states	#trans	#states	#trans
unoptimized	17	26	29	45	52	21	24	12	14
optimized-GT	5	16	18	26	32	12	14	8	10
optimized-GT-and-LT	5	7	9	11	13	6	6	4	5
YUN (manual)	5	7	9	14	16	4	4	3	3

Figure 20. State Machine Comparison

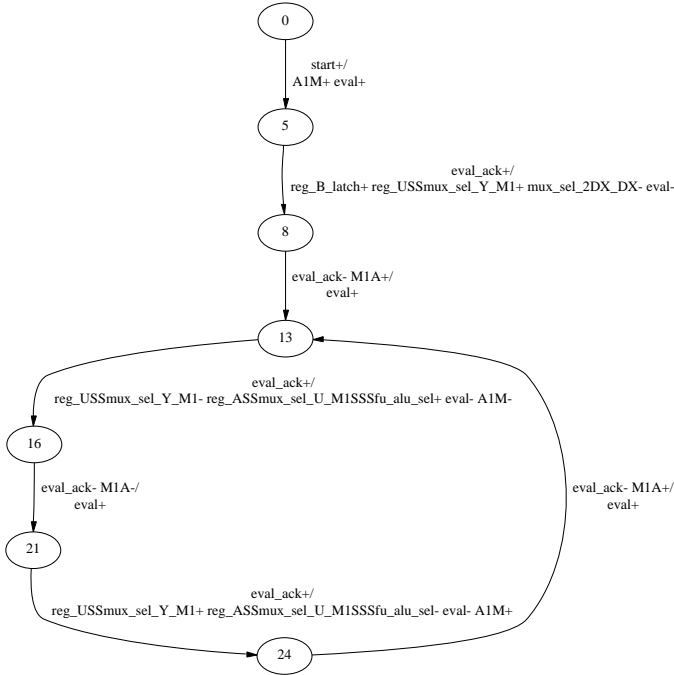


Figure 19. ALU1 controller after LT5. Note that all figures do not include the re-set transitions, in this case from state 24 to 0 and from state 13 to 0. Including re-sets, there are 7 states and 9 transitions in total.

	Yun (manual)		our method	
	#prod	#lits	#prod	#lits
ALU1	18	110	14	83
ALU2	46	141	40	113
MUL1	19	41	11	30
MUL2	10	15	8	18
total	93	307	73	247

Figure 21. Gate-Level Comparison

Column 1 of Figure 20 compares the number of communication channels. For the DIFFEQ example, the number of channels was reduced from 17 (*unoptimized*) to 5 (*optimized-GT*) showing the impact of the global transformations.

Columns 2 through 9 focus on the state machines of the four functional controller: ALU1, ALU2, MUL2, and MUL2. For each of the four controllers, the impact of the transformations is immense. For example, for ALU2, the number

of states and transitions are reduced from 45 to 11 and 52 to 13, respectively. In comparing the final *optimized-GT-and-LT* controller specifications to Yun’s, on average the specifications are comparable in terms of number of states and transitions. In particular, while our controllers are slightly worse for the small designs (MUL1 and MUL2), they are at least as good as Yun’s for the larger ones (ALU1 and ALU2). In ALU2, our controller is actually smaller than Yun’s (11 vs. 14 states and 13 vs. 16 transitions), which is striking, considering the amount of work that was put into designing Yun’s controllers.

Figure 21 compares the gate level-implementations of our best experiment (*optimized-GT-and-LT*) with Yun’s gate-level implementations. All functions are implemented by 2-level logic. For each of the methods, the number of products, literals, and state bits are listed. For ALU1 Minimalist [14] was used, but for the other controllers we had to resort to 3D [34] to synthesize the burst-mode specification, since only ALU1 is a “pure” burst-mode specification; the other ones are so-called extended burst-mode, and can currently not be handled by Minimalist. Unfortunately, 3D does single-output logic minimization only, and thus does not share products among functions as Minimalist does.

Figure 21 clearly shows that our approach of applying systematic transforms leads to very efficient implementations: the total number of literals is reduced by almost 30%, when compared to Yun’s controllers.

7 Conclusions and Future Work

A key bottleneck in the synthesis of large-scale systems has been the lack of high-quality CAD tools which allow design-space exploration. This paper is the first to introduce and automate a wide-ranging and powerful set of optimizing transformations, which allow systematic design space exploration for the synthesis of asynchronous distributed control.

The transforms implement techniques such as the exploitation of global relative timing assumptions and loop parallelism, channel multiplexing and symmetrization, and the pre-selection of muxes. They include aggressive timing- and area-oriented optimizations, several of which have not been previously supported by existing asynchronous CAD tools. We have shown that this set of transformations is powerful enough to derive controllers that are similar to or even better – up to 30% less area – than controllers that have undergone a labor-intensive manual design to make them highly-optimized.

Algorithmic heuristics based on the set of transformations presented in the paper are forthcoming. We also plan to broaden the targeted architecture to allow multiple controllers per functional unit, as well as one controller for several functional units.

References

- [1] B. M. Bachman, H. Zheng, and C. J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. International Conf. Computer Design (ICCD)*, 1999.
- [2] R. M. Badia and J. Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, Feb. 1993.
- [3] P. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *ICCAD-1992*.
- [4] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
- [5] K. v. Berkel and M. Rem. VLSI programming of asynchronous circuits for low power. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.
- [6] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.
- [7] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.
- [8] E. Brunvand, H. Jacobson, and G. Gopalakrishnan. High-level asynchronous system design using ack. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.
- [9] B. Coates, J. Ebergen, J. Lexau, S. Fairbanks, I. Jones, A. Ridgway, D. Harris, and I. Sutherland. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 161–172, Apr. 1999.
- [10] J. Cortadella and R. M. Badia. An asynchronous architecture model for behavioral synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 307–311. IEEE Computer Society Press, 1992.
- [11] J. Cortadella, M. Kishinevsky, S. Burns, and K. Stevens. Synthesis of asynchronous control circuits with automatically generated relative timing assumption. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1999.
- [12] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E80-D(3):315–325, Mar. 1997.
- [13] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 38-Sup23, pages 231–286. Marcel Dekker, Inc., 1998.
- [14] R. Fuhrer, S. Nowick, M. Theobald, N. Jha, and L. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. In *International Workshop on Logic Synthesis*, 1998.
- [15] R. Fuhrer, S. Nowick, M. Theobald, N. Jha, and L. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. Technical Report CUCS-020-99, Columbia University, 1999. Download site is <http://www.cs.columbia.edu/~nowick>.
- [16] R. M. Fuhrer. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. PhD thesis, Columbia University, 1999.
- [17] H. v. Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.
- [18] E. Kim, J.-G. Lee, and D.-I. Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.
- [19] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar. 1996.
- [20] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *DAC-1994*.
- [21] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.
- [22] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *DAC-91*.
- [23] M. Lighthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.
- [24] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.
- [25] G. D. Micheli. *Synthesis And Optimization Of Digital Circuits*. McGraw-Hill, 1994.
- [26] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *ICCD-1992*.
- [27] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.
- [28] S. Nowick and B. Coates. UCLOCK: automated design of high-performance unlocked state machines. In *IEEE International Conference on Computer Design*, pages 434–441, October 1994.
- [29] S. M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.
- [30] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.
- [31] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Y. n, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: an asynchronous instruction-length decoder. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70. IEEE Computer Society Press, April 1999.
- [32] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, Apr. 1999.
- [33] M. Theobald and S. M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design*, Nov. 1998.
- [34] K. Yun and D. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1992.
- [35] K. Y. Yun, A. E. Dooply, J. Arceo, P. A. Beerel, and V. Vakiltojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997.