Transformations for the Synthesis and Optimization of Asynchronous Distributed Control*

Michael Theobald Steven M. Nowick

Department of Computer Science
Columbia University
New York, NY 10027

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. [26] 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. Several commercial asynchronous chips have been produced in the last couple of years (e.g., microcontroller chips in Philips' commercial pagers and other designs [12]), and a number of companies are designing experimental asynchronous chips (e.g. [22] and [6]).

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 [10, 9, 25], 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 [22] 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 [4, 20]. 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 [20], developed by van Berkel et al., provides only one implementation per specification. There are no 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 design-space exploration [17, 4]. 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, 7, 15, 3, 5, 13, 16, 14, 21, 1]. Cortadella and Badia propose a synthesis style for the control unit where each datapath block is controlled by a dedicated sub-controller [7], 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 [13]. These approaches are 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) [18]. 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 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. For example, Kim et al.'s recent approach [13] is also based on CDFGs, however their method does not do design space exploration, and is limited to handling less concurrent specifications than our approach.

As a detailed case study, the transformations are applied to the well-known *differential equation solver* high-level synthesis benchmark [26, 18]. A highly-optimized asynchronous implementation by Yun et al. [26] 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.

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.

* This work was supported by NSF ITR Award No. NSF-CCR-0086036, NSF Award No. NSF-CCR-9988241, and a gift from Sun Microsystems.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

[1] These 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.

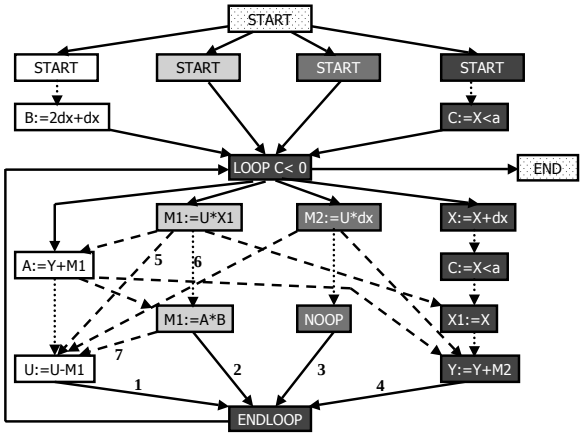

Figure 1: CDFG for DIFFEQ

2.1 CDFG Specification

The synthesis method receives as an input a scheduled and resource-bounded CDFG [18] 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, 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 *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 allocated 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 signal when the receiving CDFG node is allowed to execute.

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

Control flow (solid arcs): control arcs from and to START, END, IF, ENDIF, LOOP and ENDLOOP nodes.

Scheduling within a FU (dotted arcs): scheduling arcs to order the operations assigned to a functional unit

Data dependency (dashed arcs): *(i)* incoming arcs from operations that provide its operands R_2 and R_3, and *(ii)* outgoing arcs to operations that use the result R_1.

Register allocation (dashed arcs): *(i)* incoming arcs from all operations that use the old register value of R_1, to avoid early overwriting of R_1, and *(ii)* 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 $(LOOP, A := Y + M1)$ 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 $M1$.

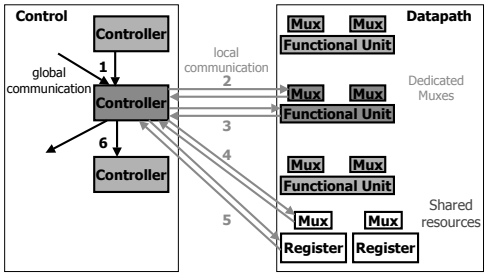

Figure 2: Target Architecture

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 [4], 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 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, a basic unoptimized 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 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

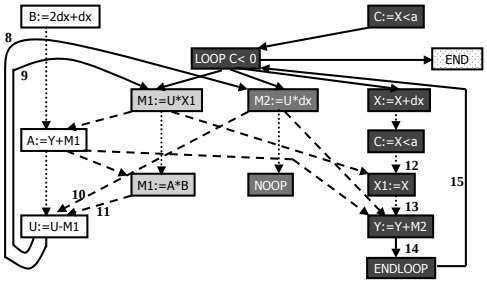

Figure 3: After GT1 and GT2. (Note: START and END nodes are omitted.)

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 functional unit controller is formally extracted from the CDFG (step 2), and can be synthesized using *extended burst-mode* finite state machines (AFSM) [19, 25, 11, 10, 23]. 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 and local transformations are introduced below (Sections 3 and 5). The global transformations (controller-controller) have two goals: reducing the number of communication channels between controllers, and improving performance (increasing concurrency, reducing critical path delays). These 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 11 for a fragment of 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). Each of the individual transforms has been automated. In the future, this approach will be extended by creating *scripts* that automatically apply (or derive) a *sequence* of transforms to find an optimal implementation. 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. (For more details see [24].)

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.

Compare Figures 1 and 3. In Figure 1, all four functional unit controllers are synchronized with an ENDLOOP node — by the arcs labeled 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.

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 schedule remains. Compare again Figures 1 and 3. In step A the three arcs labeled 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 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, an arc from the last write of the loop variable in the loop body to the ENDLOOP-node is added. In the DIFFEQ example, step C does not need to add any constraint. The candidate arc from the node $C := X < a$ to the ENDLOOP-node is implied by the path of constraint arcs 12, 13, and 14. Thus, the candidate arc $(C := X < a, ENDLOOP)$ 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 may no longer hold for arcs from the LOOP-node to the first use of a functional unit, so multiple requests may be queued on the same wire, in the loop. The requirement is reinstated by adding arcs from the first use of each functional unit in the loop to the ENDLOOP-node. In effect, these arcs restrict parallelism to two consecutive iterations of a loop: thus, 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.

There is one timing assumption that must hold if this transform is to be *safely* applied. This case concerns the final exiting from the loop. After applying the loop transform, the final execution of a LOOP-node examines the loop variable *while* other functional units may still be executing statements of the previous iteration. Hence, the LOOP-node "exits" 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 loop components complete their operation before needed.

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 is 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.

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 [22, 8]. However, these approaches were limited to single controllers. GT3 extends the use of relative-timing information to optimize interacting controllers, and uses it to remove unnecessary arcs.

Consider Figure 3. There are two constraint arcs from other controllers to $U := U - M1$, labeled 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 to occur.

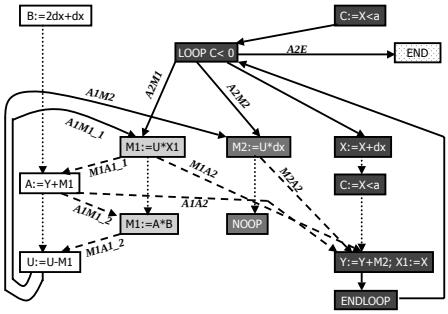

Figure 4: Channels after GT1 - GT4

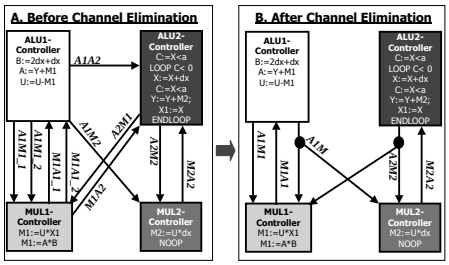

Figure 5: GT5: Channel Elimination for DIFFEQ

3.4 GT4: Merging of Assignment Nodes

Merging of 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, and thus do not use the functional unit. Such nodes 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.

3.5 GT5: Communication Channel Elimination

After the first four transformations GT1 through GT4 have been applied to optimize at the graph level (i.e. CDFG), 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 4). The goal of "communication channel elimination" is to delete as many communication channels between controllers as possible.

Figure 5 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, including 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 4, and the CDFG corresponding to the right side is in Figure 6.

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 two channels that connect the same functional units and that are never concurrently active. After multiplexing, the two different events on the two channels typically become different phases on the shared channel.

Consider Figure 7, 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. Thus, there are two channels from ALU1 to MUL1, and two from MUL1 to ALU1. "Multiplexing" the

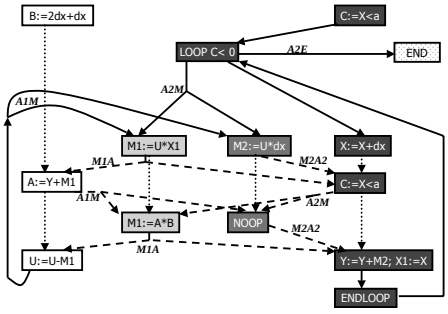

Figure 6: After Channel Elimination

two channels from ALU1 to MUL1 leads to sharing one communication channel (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 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.

Consider the CDFG in Figure 8. The constraint arc 4_{old} 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: the direct communication channel between the leftmost (ALU1) and rightmost (ALU2) controllers has been eliminated, as shown in Figure 8 (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. Events 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 the CDFG. Next, each set is transformed to a multi-way channel. Finally, the pair of multi-way channels is multiplexed.

Figure 9 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), which are then multiplexed.

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 6) into asynchronous Burst-Mode Controllers [19, 25, 11, 10, 23].

4.1 Background on Burst-Mode FSMs

In a Burst-Mode AFSM, state transitions occur when a specified *input burst* (set of variables before the "/") has been received. A fragment of a BM AFSM is shown in Figure 11. During the transition to the next state, the corresponding *output burst* (set of variables after the "/") is generated. Extended Burst-Mode AFSMs (XBM) allow

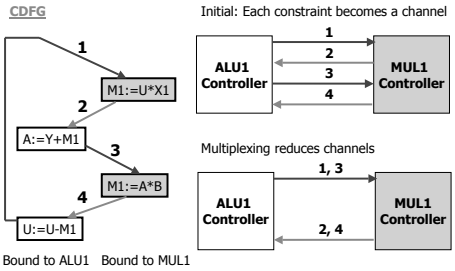

Figure 7: GT5.1: Multiplexing Constraints

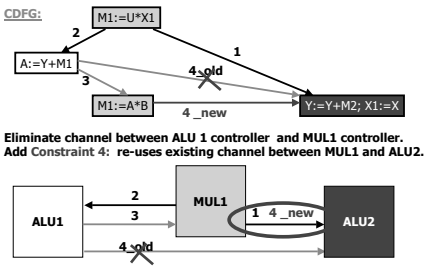

Figure 8: GT5.2: Concurrency Reduction

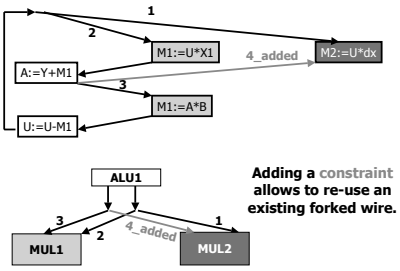

Figure 9: GT5.3: Symmetrization

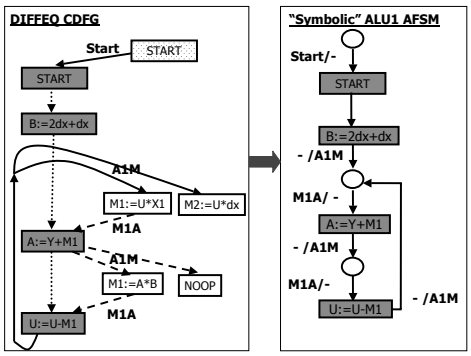

Figure 10: Burst-Mode Extraction

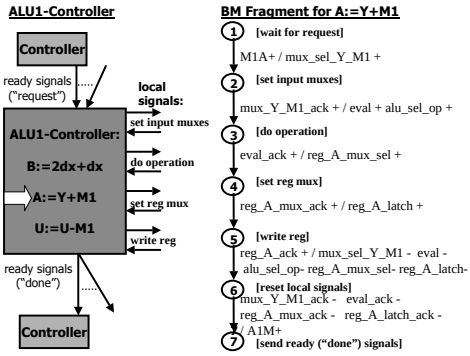

Figure 11: BM Expansion of RTL-Node A:=Y+M1

two important extensions. Selected inputs may arrive on earlier arcs (*directed don't cares*), and level inputs can be sampled (*conditionals*) (see [19, 25] for details).

4.2 Burst-Mode Extraction: An Overview

The proposed extraction method is based on a direct translation scheme for each CDFG node. Consider Figure 10, 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" AFSM. This AFSM includes one symbolic node for each CDFG node.

Each node is then expanded into a Burst-Mode fragment which implements the operation. In Figure 11, the left side shows the ALU1 controller, and its communication with other controllers and its datapath. The controller executes three RTL statements ($B := 2dx + dx$, $A := Y + M1$, and $U := U - M1$), and it is now 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.

The given BM fragment in Figure 11 consists of a series of six state transitions to implement the micro-operations: (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 micro-operations (i) through (iv) is done by a $req+$ and $ack+$ pair, the first half of a 4-phase handshake. In the figure the micro-operation labels are placed between the corresponding $req+$ and $ack+$ pair. In (v) all req/ack-pairs are then re-set to 0 in parallel ($req-$, $ack-$). Micro-operation (ii) includes two tasks: (a) selecting the operation to be performed — FUs such as ALUs can execute multiple operations, and (b) initiating the execution in the functional unit.

The actual algorithm for BM extraction is summarized as follows. First, each CDFG node is directly translated into a corresponding BM fragment. Second, BM fragments are stitched together to obtain a near-complete specification for the controller. Third, signal phases to global communication signals are assigned. 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. (For more details see [24].) Figure 11 shows the translation of a single CDFG node (for $A := Y + M1$) into a BM fragment with global signal phases assigned (e.g. *M1A+*, *A1M+*).

5. Local Transformations: Controller-Datapath

The outcome of 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 now be applied to each of the individual controllers. (For more details see [24].)

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 micro-operation) 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 transition from state 6 to state 7 in the BM fragment in Figure 11. Under most local timing requirements, it is safe to "move-up" the global "done" signal $A1M+$ to the state transition from state 4 to 5, i.e. latching the result (*reg_U_latch*) and sending a global "done" signal to other controllers ($A1M+$) are now performed in parallel.

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 provides opportunities for the application of the "signal sharing" transform (LT5). "Move-down" is typically applied to the reset phases of local signals ($req-$, or $ack-$).

5.3 LT3: Mux-Preselection

Mux selection 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. 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.[2]

[2] "Mux-preselection" can be viewed as an important special case of "move-up".

	#comm. channels	ALU1 #states	ALU1 #trans	ALU2 #states	ALU2 #trans	MUL1 #states	MUL1 #trans	MUL2 #states	MUL2 #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 12: State Machine Comparison

	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	244

Figure 13: Gate-Level Comparison

5.4 LT4: Remove Acknowledgments

The transform LT4 removes local 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. As an example, the signals *reg_A_ack* and *reg_A_mux_ack* might be removed from the BM fragment in Figure 11.

5.5 LT5: Signal Sharing

Finally, "signal sharing" aims at reducing the number of outputs of a controller. Eliminating outputs is achieved by merging distinct control wires into a single forked wire. The forked wire then activates several datapath operations concurrently. This transformation can be applied to two wires that carry the same signal value at all times, i.e., if their corresponding signals appear in precisely the same set of output bursts in a BM specification across all RTL statements executed by the controller.

6. Experimental Results

A prototype version of the presented method has been implemented. 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. [26]. 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 12. 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.

Column 1 of Figure 12 compares the number of communication channels. For the DIFFEQ example, the number of channels was reduced from 17 (*unoptimized*) to 5 (*optimized-GT*), thus showing the impact of the global transformations. Columns 2 through 9 focus on the state machines of the four functional 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.

Figure 13 compares the gate-level implementations of our best experiment (*optimized-GT-and-LT*) with Yun. All functions are implemented by two-level logic. For each of the methods, the number of products and literals are listed. For ALU1 Minimalist [10] was used, but for the other controllers we used 3D [25] to synthesize the burst-mode specification, since only ALU1 is a "pure" burst-mode specification; the other ones are extended burst-mode specifications, and currently cannot be handled by Minimalist. Unfortunately, 3D does single-output logic minimization only, and thus does not share products among functions as Minimalist does. Figure 13 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. 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 and scripts 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.

8. REFERENCES

[1] B. M. Bachman, H. Zheng, and C. J. Myers. Architectural synthesis of timed asynchronous systems. In *ICCD*, 1999.

[2] R. M. Badia and J. Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *EDAC*, 1993.

[3] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral verilog HDL. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 2000.

[4] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[5] E. Brunvand, H. Jacobson, and G. Gopalakrishnan. High-level asynchronous system design using ack. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 2000.

[6] B. Coates, J. Ebergen, J. Lexau, S. Fairbanks, I. Jones, A. Ridgway, D. Harris, and I. Sutherland. A counterflow pipeline experiment. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1999.

[7] J. Cortadella and R. M. Badia. An asynchronous architecture model for behavioral synthesis. In *EDAC*, 1992.

[8] J. Cortadella, M. Kishinevsky, S. Burns, K. Stevens. Synthesis of asynchronous control circuits with automatically generated relative timing assumption. In *ICCAD*, 1999.

[9] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. on Fundamentals of Electronics Communications and Comp. Sci.*, Mar. 1997.

[10] 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/async.

[11] R. M. Fuhrer. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. PhD thesis, Columbia University, 1999.

[12] H. v. Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1998.

[13] E. Kim, J.-G. Lee, and D.-I. Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 2000.

[14] T. Kolks, S. Vercauteren, B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1996.

[15] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *DAC*, 1996.

[16] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 2000.

[17] 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, Addison-Wesley, 1990.

[18] G. D. Micheli. *Synthesis And Optimization Of Digital Circuits*. McGraw-Hill, 1994.

[19] S. M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Ph.D. Thesis. CSL-TR-95-686, Computer Systems Laboratory, Stanford University, 1993.

[20] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[21] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1996.

[22] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: an asynchronous instruction-length decoder. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1999.

[23] M. Theobald and S. M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Trans. on Computer-Aided Design*, Nov. 1998.

[24] M. Theobald and S. M. Nowick. Transformations for the Synthesis and Optimization of Asynchronous Distributed Control. Technical Report, Columbia University, 2001 (to appear). Download site is http://www.cs.columbia.edu/async.

[25] K. Yun and D. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *ICCAD*, 1992.

[26] K. Y. Yun, A. E. Dooply, J. Arceo, P. A. Beerel, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Intl. Symp. Adv. Res. in Asynchronous Circ. and Sys.*, 1997.