Resynthesis and Peephole Transformations for the Optimization of Large-Scale Asynchronous Systems

Tiberiu Chelcea Steven M. Nowick*
Department of Computer Science
Columbia University
New York, NY, USA e-mail: {tibi,nowick}@cs.columbia.edu

ABSTRACT

Several approaches have been proposed for the syntax-directed compilation of asynchronous circuits from high-level specification languages, such as Balsa and Tangram. Both compilers have been successfully used in large real-world applications; however, in practice, these methods suffer from significant performance overheads due to their reliance on straightforward syntax-directed translation.

This paper introduces a powerful new set of transformations, and an extended channel-based language to support them, which can be used an optimizing back-end for Balsa. The transforms described in this paper fall into two categories: resynthesis and peephole. The proposed optimization techniques have been fully integrated into a comprehensive asynchronous CAD package, Balsa. Experimental results on several substantial design examples indicate significant performance improvements.

1. Introduction

Several approaches have been proposed for compilation of asynchronous circuits from high-level specification languages [1, 13, 4, 8]. Compilers such as Balsa [1] (from University of Manchester) or Tangram [13, 10] (from Philips Research Labs, and now used for commercial products) perform a syntax-directed compilation of high-level specifications into an intermediate representation using *handshake components*. These components are then synthesized into circuits using a template-based approach. An *occam*-based compiler [4] and an alternative translation approach from Caltech [8] have also been proposed.

Balsa and Tangram have been widely-used, but their syntax-directed translation methods introduce significant performance overheads [3, 15]. While these synthesis styles have the advantage of "transparency" (the designer is controlling the final results from the high-level program), they also have the disadvantage of avoiding the use of powerful back-end transformations, except for simple peephole optimizations. The Caltech approach uses only localized resynthesis techniques (such as handshake reshuffling and "guard symmetrization"), which are not systematically applied or captured at a higher language level. In contrast, this paper presents a much wider-ranging and more powerful set of transformations for the optimization of asynchronous systems.

This paper introduces three major contributions. The first contribution is a set of new transformations which can be used in a back-end optimizer for large asynchronous circuits. The transforms fall into two categories: peephole and resynthesis. A peephole optimization optimizes components in a sliding window; if the netlist of components in the window conform to a pattern, they are replaced in a template-based fashion by other existing components. In contrast, resynthesis optimizations attempt to replace components in a non-template-based fashion, by re-synthesizing them.

The second contribution of this paper is the extension to a component specification language, called CH. CH was introduced in [5] to model and manipulate only control components, and is extended here to model *datapath* components. CH is very important in the proposed approach: each transform is formalized as a simple language manipulation procedure in CH. The resulting specifications are thus independent of the synthesizable low-level specifications into which CH is translated. Burst-mode controller specifications are currently employed, but they are just one of several possible low-level specification styles. The final contribution of the paper is a new automated back-end for the Balsa synthesis system which incorporates the proposed transforms.

The proposed new transforms facilitate design-space exploration. By systematically applying them, the designer, starting with an unoptimized circuit, can obtain different improved implementations of the initial design. An additional advantage is that these transforms are formalized using an intermediate specification language, which allows the designer to observe tradeoffs between different related implementations at a higher level.

The integrated design flow was applied to three substantial asynchronous design examples. Pre-layout back-annotated Verilog simulations on technology-mapped implementations indicate up to 54% speed improvement over the unoptimized implementations. Improvements are significantly better than those obtained using previous optimizations proposed in [5].

The paper is organized as follows. Section 2 presents a brief overview of the proposed synthesis approach. Section 3 gives some asynchronous background, and Section 4 gives background information on the CH language. Section 5 introduces the new CH extensions, while Sections 6 and 7 present the proposed resynthesis and peephole optimizations, respectively. Finally, Section 8 presents experimental results, and Section 9 presents conclusions and directions for future work.

2. Overview of the Approach

Balsa [1, 2] is collection of programs that facilitate the description and synthesis of asynchronous systems. The original Balsa design flow takes a system description in the Balsa high-level description language and translates it into a netlist of handshake components. Each component is then mapped to actual circuits in a template-based fashion.

The improved back-end includes an *optimization step*, which consists of the proposed set of resynthesis and peephole transforms, as well as of the previous optimizations from [5]. The revised Balsa flow is shown in Fig. 1 (the shaded boxes indicate research contributions).

The new back-end takes the list of handshake components, and performs the proposed peephole and resynthesis transforms on them, to obtain a list of optimized components. It then partitions the list

*This work was supported by NSF ITR Award No. NSF-CCR-0086036 and NSF Award No. CCR-99-88241, and by a grant from the New York State Microelectronics Design Center.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

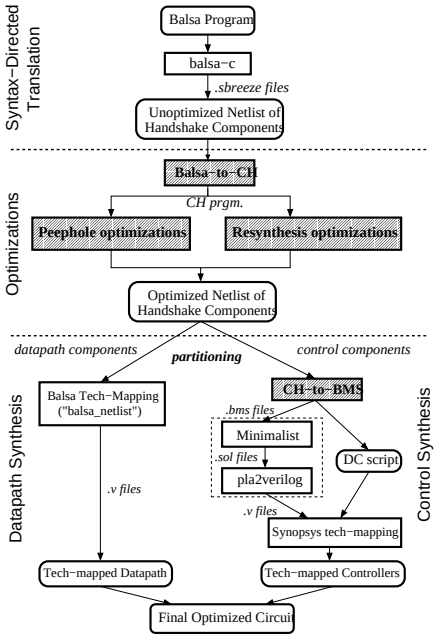


Figure 1: New Balsa System Design Flow

into datapath and control components. The datapath components are then synthesized using the existing Balsa system technology mapper (balsa-netlist). The control components are synthesized into hazard-free logic implementations using the Minimalist Burst-Mode CAD package [7], with speed-oriented scripts. The logic implementations are then technology mapped using a commercial tool (Synopsys Design Compiler). Hazards must not be introduced at this step. The proposed solution is to split the controllers into smaller modules (e.g. using Verilog, one module per level of a NAND-NAND implementation), technology map them separately, and then formally check the solutions for hazard freedom. In all cases, the technology-mapped circuits were correct (i.e. hazard-free). For more details on the technology-mapping step, see [5].

The current design flow incorporates some of the lower-level synthesis steps from the previously-proposed flow in [5]. The interactions between the optimizations and Minimalist, pla2verilog, Synopsys Design Compiler technology-mapping, and balsa-netlist are similar in both design flows. However, the major difference is in the optimization step, which is now enlarged and improved with the newly-proposed transforms.

2.1 Related Work

Several approaches for component modeling have been proposed ([2, 8, 13, 10]). However, some can only model a single behavior of a particular type (e.g. sequencing) rather than a desired set of closely-related variants (e.g. various "interleavings" of a basic sequencing protocol) [2, 10]. Others use low-level specifications where each signal transition is enumerated, which are cumbersome to manipulate [8, 13, 10].

Several peephole optimization and control resynthesis techniques for asynchronous systems have also been previously proposed. Previous peephole optimizations [13, 4] have typically been *behavior-preserving*, and most have dealt with simple components and improvements (such as redundancy removal). In contrast, unlike several of the previous techniques, all of the proposed transforms in this paper are *behavior-modifying*, and subsume many of the earlier ones.

Recent approaches to control resynthesis [11, 6] are mainly variants of component composition, using Petri-net or trace-theory formalisms, and are behaviorpreserving. In contrast, the proposed resynthesis techniques include much more powerful transformations for design-space exploration, including concurrency enhancement and protocol manipulation. In addition, the proposed transforms are integrated into a comprehensive CAD package for asynchronous synthesis. Finally, these previous resynthesis approaches did not report results on performance improvements, only on area.

3. Asynchronous Background

3.1 Asynchronous Communication Protocols

There are two key concepts in understanding the behavior of asynchronous components: their *communication protocol* and techniques to *interleave* two or more communication protocols.

Two asynchronous components communicating among themselves over one channel use a "communication protocol" (Fig. 2a). The

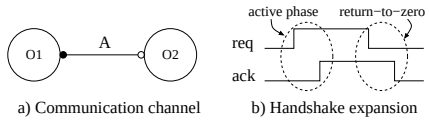


Figure 2: Communication Protocol

most common protocol is called "4-phase handshaking": component O1 initiates the communication, and component O2 completes the communication. The component's interface to a channel is called a port. Each port has a type: component O1 initiates the communication, and thus has an *active* port A (indicated by the black dot); component O2 completes the communication, and thus has a *passive* port A (indicated by the hollow dot). The communication channel is implemented by two wires: a request and an acknowledge. A communication (Fig.2b) has the following handshake expansion: A_r↑ A_a↑ A_r↓ A_a↓ (where A is the channel name, and r/a are the request/acknowledge wires), and consists of two phases: the *active phase* (rising transitions on the request and acknowledge wires, in this order), and a *return-to-zero phase* (falling transitions in the same order).

When a component is connected to two or more channels, an important issue is how the communication protocols on the channels are *interleaved*. For example, a component communicating on channels A and B with two other components might *enclose* the handshake on B within the handshake on A (A_r↑ B_r↑ B_a↑ B_r↓ B_a↓ A_a↑ A_r↓ A_a↓), or, it might *sequence* the communication on B after the communication on A (A_r↑ A_a↑ A_r↓ A_a↓ B_r↑ B_a↑ B_r↓ B_a↓). Different interleavings give different tradeoffs (such as speed, area, or power) in the component's implementation. Therefore, a component description language such as CH must define operators that model different interleavings.

3.2 Burst-Mode Specifications

Burst-mode specifications are the target of the synthesis path. They are a commonly-used Mealy-type specification for asynchronous controllers [9]. The Minimalist CAD package [7] is used in the design flow to synthesize the optimized Burst-mode controllers.

A burst-mode specification (Fig. 8b) is a finite-state machine specification, and consists of a set of states and a set of arcs. An arc is labeled with an input burst (a set of input transitions, for example "B_a- C_a+"), followed by an output burst (a set of output transitions, for example "C_r-"), and connects two states. An input (output) burst describes a set of input (output) transitions: rising transitions are marked with a '+', and falling transitions are marked with a '-'. A BM machine waits for an input burst to arrive; transitions may come in any order and at any time. Once the complete input burst has arrived, the output burst is generated and the machine moves to the next specification state.

4. CH Language: Basics

The previously-proposed CH language [5] is an intermediate specification language. In the old version [5], CH could model only control handshake components; in this paper, CH is extended to model datapath components also. CH is important in the proposed approach: each optimization is formalized as a simple CH manipulation procedure on component specifications. This section introduces the original version of CH [5] through a series of examples, starting with channel modeling, and progressing through a series of component modeling examples. In the next section, the new extensions will be presented.

Channel Modeling. Channel communication protocols can be modeled easily and concisely using CH. Figure 3 shows two examples of components with only one port, each connected to a P-TO-P (point-to-point) channel. These channels are the most common ones, and they consist of one request and one acknowledge wire. Fig. 3a shows a point-to-point channel A connected to a *passive*

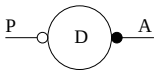
a) (p-to-p passive A) b) (p-to-p active A)

Figure 3: Channel Modeling in CH

port. The CH expression indicates the type of the channel (P-TO-P), the type of the port (active/passive), and the channel name, and it is a shorthand for the handshake expansion in Fig. 2b, where the handshake is initiated by an input transition on the request wire. In contrast, Fig. 3b shows a P-TO-P channel A connected to an *active* port. The new CH expression now indicates that the port has an active type, and has the same handshake expansion (Fig. 2b), but the handshake is initiated by an output transition on the request wire.

There are six more channel types in CH. Both MULT-REQ and MUX-REQ have multiple request wires, but during a MULT-REQ handshake there are transitions on *all* request wires, while during a MUX-REQ handshake there are transitions on *exactly one* request wire. In mirror, there are channels with multiple acknowledges (MULT-ACK, MUX-ACK), with a similar behavior. Finally, there are two special types of channels: VERB (which allows the designer to specify all transitions in a handshake) and VOID (used internally, during optimizations).

D-element. The first component example, a *D-element* (Fig. 4), was introduced by Martin [8]. The D-element has one passive port



(a) Block Diagram

```
(rep
    (enc-early   (p-to-p passive P)
                 (p-to-p active A)))
```

(b) CH specification

Figure 4: D-Element

P, and one active port A (Fig. 4a). When activated on P, the D-element performs a full handshake on the active port A, before completing the handshake on P. The CH program (Fig. 4b) concisely captures this behavior. The top-level operator REP indicates that the behavior is repeated forever. Then, handshakes on the passive and active ports are interleaved. The particular interleaving is an *early-enclosure* (ENC-EARLY), which encloses the handshake on A in the middle of the active phase of P. Unlike the CH expression, the Martin-style specification (*[[P_r↑]; A_r↑; [A_a↑]; A_r↓; [A_a↓]; P_a↑; [P_r↓]; P_a↓]) for the same element indicates every signal transition.

Sequencer Component. The second example is a useful control handshake component: a *sequencer* (Fig. 5) [2, 14, 10], used to activate two processes in turn. The component has one passive port

(a) Block Diagram

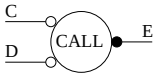
```
(rep (enc-early   (p-to-p passive A)
             (seq     (p-to-p active B)
                      (p-to-p active C))))
```

(b) CH program

Figure 5: Sequencer Component

(A) and two active ports (B and C); the active ports are connected to processes B and C on the respective channels. When activated on channel A, the component activates in turn each of the B and C channels (first B, then C) before completing the handshake on A. The CH expression (Fig. 5b) reads as follows. First, the top-level operator (rep) indicates that the behavior is repeated forever. Then, there is an early-enclosure (ENC-EARLY) interleaving between the handshake on A and the non-overlapped sequencing (SEQ) of handshakes on channels B and C. Unlike the CH expression, the Martin-style specification: *[[A_r↑]; B_r↑; [B_a↑]; B_r↓; [B_a↓]; C_r↑; [C_a↑]; C_r↓; [C_a↓]; A_a↑; [A_r↓]; A_a↓] is not hierarchical, and it is cumbersome for manipulation.

Call Component. The final example is a *call* component (Fig. 6) [2, 14, 10]. A CALL provides access to a resource whenever requested by one of several components, called callers, where the requests must be mutually-exclusive. The particular interleaving between a caller request and the resource access is a *middle-enclosure* (ENC-MIDDLE). The enc-middle operator takes two arguments (e.g. channels C and E) and returns an interleaving in which the active/return-to-zero phases of the E handshake are enclosed within the active/return-to-zero phases, respectively, of C (C_r↑ E_r↑ E_a↑ C_a↑ C_r↓ E_r↓ E_a↓ C_a↓). In addition, CH also defines a *late-enclosure* operator (enc-late), with the following handshake expansion: C_r↑ C_a↑



(a) Block Diagram

```
(rep (mutex
        (enc-middle  (p-to-p passive C)
                     (p-to-p active E))
        (enc-middle  (p-to-p passive D)
                     (p-to-p active E))))
```

(b) CH program

Figure 6: Call Component

C_r↓ E_r↑ E_a↑ E_r↓ E_a↓ C_a↓.

Burst-Mode Aware Restrictions. Once a component is modeled in CH, it must be translated into a synthesizable Burst-Mode [7, 9] specification (the chosen low-level component specification language). Therefore, several restrictions (called "burst-mode aware") must be imposed on CH: only certain combinations of operators and argument types are allowed. For "burst-mode aware" interleavings, both the interleaving operator and the passivity/activity of relevant ports must be considered; all legal combinations are given in [5]. These restrictions are enforced in two steps. First, each initial handshake component is modeled only with "burst-mode aware" CH expressions. Then, each proposed transform is also restricted: starting with BM-aware CH specifications, their result (a CH expression) is checked to be BM-aware. If it is, the transform has succeeded, otherwise it has failed. For the peephole transforms, this check is performed only once: when the transform is first formalized. In contrast, for the resynthesis transforms, this check is performed each time the transform is applied, since the transforms do not assume fixed configurations of components.

5. New CH Extensions

CH, as defined in [5], can model a variety of control handshake components. In this section, CH is extended to handle datapath component modeling. In particular, CH is extended with channel types and datapath operators. There is one final useful extension to CH, a concurrent sequencer operator; however, this operator is used to model control (not datapath) components, and therefore is more suitable for presentation in the next section.

Data Channel Modeling: Formalism. A four-phase bundled-data channel consists of a pair of control wires (a request and an acknowledge), and a bundle of data wires on which a data item is passed [10]: the request is delayed until after the data item is valid. The communication protocol on a data channel is uniquely defined by the following parameters, which are also discussed in [10]:

- **Data Flow Direction**: as indicated, a data channel consists of two control wires and data wires. There are two options as to the direction in which data flows: in the same direction as the request ("push"), or in the same direction as the acknowledge ("pull").
- **Data Validity**: in asynchronous communication, the sender and the receiver must know when the data item being passed is valid. There are three common data-validity schemes: *broad*, *early*, and *late* (see [10] for formal definitions of these data-validity types).
- **Data Item Type**: each data item has an associated data type (such as byte, word, bool). To be compatible for communication, the sender's and the receiver's ports must both be able to exchange the same data type.
- **Data Width**: a data type is encoded on a number of data wires. To be compatible for communication, the sender's and receiver's ports must both have the same number of data wires encoding a data item.

The formal syntax for a data channel in CH is: ([push|pull] [b|e|l] name [data type] [data width]). The first parameter indicates the data-flow direction. Data validity is selected by the second parameter: broad, early, or late. The name of the channel is given by name, and the last two parameters describe the data item type and the number of data wires used to encode it. When a data channel is connected to a particular port, an additional port attribute (passive/active) is also used: ([passive|active] [?|!] [b|e|l] name [data type] [data width]). In this case, the data-flow attribute (push/pull) can be transformed into the simpler notation ?|!, which indicates whether data is input (?) or output (!).

Modeling of Datapath Operators: Formalism. A handshake datapath component consists of two parts: its interface communication protocol and its functional operation. Therefore, at the CH

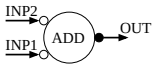
level, the component is modeled by two expressions: one that models the control, and which contains only interleaving operators and the newly-introduced data channels; and one CH expression for the datapath computation, which contains only *datapath operators*.

The two CH expressions for modeling a datapath component are used as follows. The CH control part, which models the behavior on the interfaces, is translated into a low-level specification (Burst-mode, in this approach) and then directly synthesized. In contrast, the CH expression for the datapath computation is used both to select the appropriate datapath circuit from a library of components, and to model a structural view of the connections to this datapath circuit.

Three new *datapath* operators are introduced to model the datapath computation:

- **func**: this operator models datapath circuits which compute a function (e.g. adder). The syntax is: (func circuit_type out1 ... outn inp1 ... inpn) where circuit_type indicates the type of computation performed (for example, addition, subtraction), inp1 ... inpn are the interfaces on which the inputs are received, and out1 ... outn are the interfaces on which the results are sent.
- **transf**: this operator describes datapath circuits that simply transfer their inputs to their outputs (wire connections). The syntax is: (transf out inp) where inp is the port on which data is received, and out is the port on which data is transferred.
- **var**: this operator describes a register datapath circuit (used to store data items). The syntax is: (var inp out1 ... outn) where inp is the channel name whose data wires are connected to the write port of the register, and out1 ... outn are the channel names whose data wires are connected to the read ports of the register.

Modeling of Datapath Components: Adder Example. To better understand the modeling of data channels and datapath components, an eight-bit adder handshake component (Fig. 7) is now presented. Notice that two CH expressions, E1 and E2, are needed to model this component. The adder's handshaking behavior is described by expression E1. The adder's interfaces are described by CH data channel declarations: out is a passive output port (connected to a pull channel), and inp1, inp2 are active input ports (also connected to pull channels). The arrow indicates a data channel, and its direction indicates the direction of data flow (in on the two inputs, out on the output). The data validity on each channel is broad (b): during a communication, the data item passed on the channel is valid from before *req*↑ until after *ack*↓. The last two parameters in the channel declaration indicate the data item type (byte) and the number of wires which encode it (8). The component is activated on the out port, and then concurrently fetches the operands on inp1 and inp2; when the addition is complete, the result is output on out, and the interfaces are returned-to-zero in the same order.



(a) Block Diagram

E1: (rep (enc-middle (passive ! b out byte 8)
(enc-middle (active ? b inp1 byte 8)
(active ? b inp2 byte 8))))

E2: (func add out inp1 inp2)

(b) CH program

Figure 7: Eight-Bit Adder Component

Expression E2 describes the datapath portion of the asynchronous adder: the func CH datapath operator indicates that the inp1, inp2 data wires connect to the the inputs of the addition circuit, and that the out data wires are connected to its outputs. This expression can then be used to select an eight-bit adder from a library of datapath circuits.

6. Resynthesis Transforms

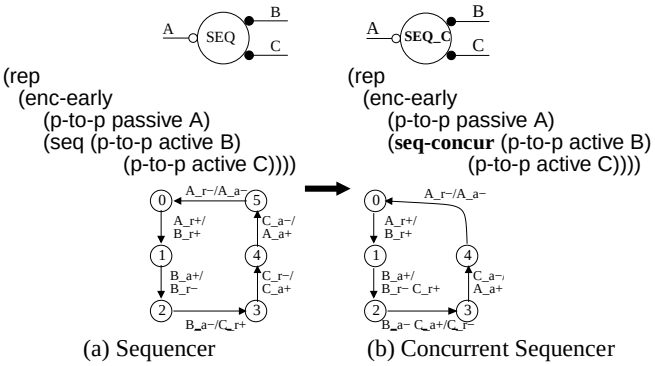
This section introduces two related transforms: Enc Replacement and Seq Replacement, which operate on individual components, and replace some of their interleaving operators by others. These transforms are simple, yet powerful: each one is formalized as a CH language manipulation procedure, and they allow for higher-level exploration of tradeoffs in asynchronous synthesis.

6.1 Seq Replacement

The Seq Replacement transform replaces a non-concurrent sequencing operator (seq) by a more concurrent sequencing operator (seq-concur). The potential benefit of the transform is increased throughput for the entire subsystem controlled by this operator. This subsection first introduces and formalizes the new seq-concur operator. It then illustrates its use in the new transform where sequencer components are replaced by concurrent sequencer components.

The new seq-concur operator defines a concurrent interleaving of handshakes on two channels. The operator takes two arguments (channels) and interleaves them such that the first channel completes its active phase; next, the return-to-zero phase of the first channel is concurrent with the active phase of the second channel; and, finally, the return-to-zero phase of the second channel is executed. More formally, if the seq-concur operator is applied to channels B and C, it returns the following handshake expansion: B_r↑; [B_a↑]; B_r↓, C_r↑; [B_a↓ ∧ C_a↑]; C_r↓; [C_a↓].

A sequencer component can be easily transformed into a concurrent sequencer using the Seq Replacement, as illustrated in Fig. 8[1]. Each component has one passive port A and two active ports (B and C), whereas each of the active ports is connected to other components on the respective channels. When activated on channel A, the non-concurrent sequencer performs a non-overlapped activation of channels B and C in turn. In contrast, with the concurrent sequencer, the return-to-zero phase on channel B is overlapped with the active phase on channel C. At the CH level, a sequencer component is easily changed to the a concurrent sequencer by changing the seq operator into the newly-introduced seq-concur operator. The Burst-mode specifications for the two components are shown in Fig. 8.



(a) Sequencer (b) Concurrent Sequencer

Figure 8: The Seq Replacement Transform

There is an important correctness issue in the use of concurrent sequencers. The newly-introduced concurrency between processes may potentially result in data [12] and structural hazards. For example, data hazards may occur when two concurrent processes share a variable. Suppose a 4-way sequencer (activating four processes P1 ... P4) is to be made more concurrent, but there is a data hazard between processes P2 and P3. For example, P2 writes a variable, and P3 reads that variable; if a concurrent sequencer is used, so that P2 and P3 are concurrent, the variable might be read by P3 even before P2 finishes latching a new value.

The solution is to define *hybrid* sequencers, which result from the selective application of Seq Replacement. The CH expression for the hybrid sequencer to be used in the above example is:

```
(rep (enc-middle (p-to-p passive a)
        (seq-concur  (p-to-p active P1)
                     (seq    (p-to-p active P2)
                             (seq-concur (p-to-p active P3)
                                         (p-to-p active P4))))))
```

Effectively, a concurrent sequencer operator is used whenever there are no data hazards, and a non-concurrent sequencer operator whenever there are hazards (between P2 and P3). Now, using this hy-

[1] It should be noted, however, that the optimization is not limited to sequencers: any component that has a sequencing operator (for example, WHILE [1]) can be optimized with this transform.

brid sequencer, P2 finishes writing the variable before P3 is started; therefore, P3 correctly reads the new variable value.

6.2 Enc Replacement

The Enc-Replacement transform replaces one "enclosure" interleaving operator in a CH expression by another one (for example, replaces enc-early with enc-middle). Thus, the transform modifies the interleaving of communications on two or more channels connected to a given component's ports. This transform is now illustrated on two examples.

(L/A;R) element. The (L/A;R) element was introduced in [8] to implement the sequential execution of asynchronous processes. In this example, it is shown how this sequencing component is transformed into a Tangram *parallel component* [10].

The implementation of the (L/A;R) element corresponds to the following CH expression:

```
(rep (enc-middle (enc-late     (p-to-p passive L)
                               (p-to-p active R))
                 (p-to-p active A)))
```

Like a sequencer component, the (L/A;R) element is activated on a passive port, and then performs sequential handshakes on the active ports A and R, before completing the handshake on L. The Martin-style specfication for this element is: *[[L_r↑]; A_r↑; [A_a↑]; L_a↑; [L_r↓]; A_r↓; [A_a↓]; R_r↑; [R_a↑]; R_r↓; [R_a↓]; L_a↓].

Enc Replacement can be applied to the enc-late operator, which is simply changed to the middle-enclosure operator:

```
(rep (enc-middle (enc-middle  (p-to-p passive L)
                              (p-to-p active R))
                 (p-to-p active A)))
```

Now, the behavior of the (L/A;R) component is radically changed: the active/return-to-zero phases of the A and R handshakes are *concurrently* executed in the middle of the active/return-to-zero phases, respectively, of the L handshake (*[[L_r↑]; A_r↑, R_r↑; [A_a↑ ∧R_a↑]; L_a↑; [L_r↓]; A_r↓, R_r↓; [A_a↓∧R_a↓]; L_a↓]). In effect, the (L/A;R) component is now a parallel component: the subsystems attached to each of the A and R channels are executed concurrently.

New Adder Component. The second example involves the adder datapath components. In Section 4 an adder component was modeled. Replacing the top-level middle-enclosure operator (see Fig. 7b) by an early-enclosure operator changes drastically the behavior of the adder component. The new CH expression is:

```
E1: (rep      (enc-early
                  (passive ! b out byte 8)
                  (enc-middle      (active ? b inp1 byte 8)
                                   (active ? b inp2 byte 8))))
E2: (func add out inp1 inp2)
```

The new adder now performs all of its computation (requesting the operands, producing the result, and releasing the operands) in the middle of the active phase of the channel out handshake. The return-to-zero phase is extremely fast. In contrast, the previous adder releases its operands late, at the end of the return-to-zero phase on channel out. In effect, the new adder facilitates *resource sharing*: since the operands are now released early, they can be immediately used by other processes.

7. Peephole Transforms

This section introduces two classes of peephole transforms. The first class contains a single powerful transform (Protocol Reversal on Functional Units), which is applied to functional units, and replaces one port type by another. The second class is a collection of various template-based clustering transforms. Each transform is *behavior non-preserving*: the resulting component is equivalent up to a different ordering of the signal transitions of the initial components.

7.1 Protocol Reversal on Functional Units

This peephole transform changes the interface types (active/passive) of functional units and also of the components connected to these units. The goal of the transform is to decrease the latency of a subsystem. Very much like Enc/Seq Replacement, this transform manipulates CH constructs (in this case, the interface type parameter in channel declarations).

Before illustrating the transform in detail, it is worth giving some motivation for its application. In Balsa, functional units typically have a passive output port and several active input ports. A functional unit is activated on the output port when a result is needed; the unit then actively requests the operands on the input channels, receives the data, and performs the operation. The idea of the transform is to modify the unit's interfaces so that the *data operands* themselves activate the functional unit, thus computing a result even before requested on the output port. As a result, the latency to compute the result may be significantly decreased.

As an example, consider the initial peephole window shown in Fig. 9a. It consists of a unary functional unit, a variable, and a transferer [2, 14]. In general, the functional unit may have more than one input port. The variable component is used to store data

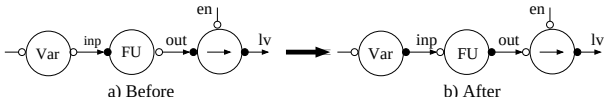


Figure 9: Protocol Reversal on Functional Units

items; it has a passive port (on which a data item is written), and an active port (on which a data item is read). First, the variable (VAR) latches a data input. Later on, the control enables the transferer (on channel en), which, in turn activates the functional unit (FU). The unit obtains the data input from the variable, computes the result, which is then transfered on lv to its destination. The whole cycle repeats with the variable latching the next data item.

In the optimized configuration (Fig. 9b), the port types in each component are now changed. This change corresponds to modifying the initial CH expression for the functional unit:

(rep (enc-middle (passive ! b out T1 L1) (active ? a inp T2 L2))) ⇒
(rep (enc-middle (**passive** ? a inp T2 L2) (**active** ! b out T1 L1)))

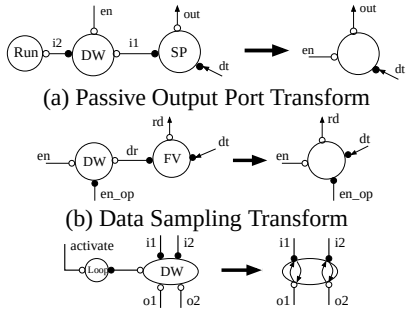
The activity types on out and in are switched from passive/active to active/passive. Correspondingly, the inp port in the variable is now active, and the out port in the transferer is now passive.

The operation of the new configuration is now different. The variable, once written, immediately starts the computation in the functional unit. The result is sent to the new transferer. At some time later, the control will finally enable the transferer, which *immediately passes* the result on channel lv, thus decreasing the latency of the whole operation.

Unfortunately, it is not easy to generalize this peephole optimization for n-input functional units. For unary units, the pattern of new data items is fixed: the variable is written each cycle. However, for n-ary units, the pattern of new data items cannot be pre-determined. For example, within one cycle, a single variable can be written, while the others not; or, all but one variable can be written, while the last variable not. We are currently looking for ways to generalize this approach, perhaps using more information than just the connections of functional units.

7.2 Clustering Transforms

The final class of peephole optimizations uses template-based matching to identify and optimize targeted clusters of components. The three new transforms are graphically summarized in Fig. 10 by showing the initial and final configurations for each one. It is expected that a number of additional configurations can be added to this list in the future.



(a) Passive Output Port Transform

(b) Data Sampling Transform

(c) Loop Enabling Transform

Figure 10: Clustering Peephole Transforms

Passive Output Port Transform. This transform optimizes a Balsa subsytem which outputs a data item to the environment, whenever

the environment requests data. The transform takes three components, and clusters them into one.

The initial Balsa implementation (Fig. 10a) consists of three components: a SEQUENCE PULL [2], a DECISION WAIT [2], and a RUN [2, 13] component. The SEQUENCE PULL, when requested for data on channel OUT, first synchronizes with the DECISION WAIT on channel i1 before requesting a data item on channel dt; when the data item is received, it is passed on out. In turn, DECISION WAIT first synchronizes with the Balsa process on channel en, and then synchronizes with the RUN component on channel i2. The RUN component always completes any handshake started on its passive port. Thus, after it is requested for data, the subsystem performs four sequential operations: it synchronizes with the Balsa process, synchronizes with the RUN component, receives a data item on dt, and outputs it on out.

The result of this transform is a single clustered component, which has the same interfaces minus the i1 and i2 channels. The output cycle is now shortened: the component first concurrently synchronizes the data item request with the Balsa process, then receives a data item on channel dt, which is then output on channel out. The new component has a very simple implementation, just one C-element and wires between data inputs and data outputs.

Data Sampling Transform. This transform optimizes a Balsa subsystem which reads a data item from the environment. The transform takes two components, and clusters them into one.

The initial Balsa implementation (Fig. 10b) consists of only a FALSE VARIABLE component [2] and a DECISION WAIT component. An n-way FALSE VARIABLE component has one passive input port (dt – used to receive a data item), an active control port (dr – used to indicate data received), and n passive output ports used to sample the data item (Fig. 10b shows a one-way FALSE VARIABLE). The subsystem performs three sequential operations: it receives a data item, synchronizes with the Balsa process on channel en, and, finally, enables other processes (on channel en_op) to read the data item on channel rd.

The result of this transform is a single clustered component, which has the same interfaces, minus the eliminated dr channel. The data reading cycle is now shortened: the component first concurrently synchronizes the receiving of a data item with the Balsa process (on dt and en), and then enables other processes to read the data item. The implementation of the new component is very simple, just a C-element and wire connections on datapath signals.

Loop Enabling Transform. This transform operates on handshake components that synchronize two independent Balsa subsystems. The transform takes two components, and replaces them by just pairs of wires.

The transform takes as an input a LOOP component [2, 14] connected to both a special activate channel (automatically synthesized, used to start the whole system) and to a DECISION WAIT component (Fig. 10). The loop component, once started on the activate channel, enables forever the DECISION WAIT component to pass handshakes from a passive port to a corresponding active port.

The resulting component consists of just pairs of wires. The intuition behind the transform is simple: since the loop component *unconditionally* enables the DECISION WAIT component, the enabling logic in the DECISION WAIT is redundant and can be eliminated; in addition, the LOOP component is also eliminated.

8. Results

The entire synthesis flow has been evaluated on a number of substantial examples. Each example is described using the Balsa language, and synthesized with `balsa-c`, to obtain the initial unoptimized netlist of handshake components. The list of components is then optimized: the peephole optimizations are applied first, and then the resynthesis optimizations. Each optimized component is then synthesized and technology-mapped into the AMS 0.35μm library using Synopsys Design Compiler. The final implementations were back-annotated using `pearl`, and simulated with Cadence Verilog-XL. All steps, except the implementation of the enc/seq transforms, have been automated.

Three complete systems have been optimized and simulated: a four-handshake systolic counter [14], an eight-bit word three-place low-latency FIFO, and a programmable eight-bit counter.

Examples		Balsa	Previous Transforms [5]		All Transforms	
		Speed (ns)	Speed (ns)	Improvement	Speed (ns)	Improvement
Systolic Counter		24.81	16.06	35.26%	11.28	54.50%
FIFO	latency	17.33	15.19	12.36%	10.32	40.45%
	put cycle time	8.41	8.06	4.16%	6.22	26.04%
	get cycle time	11.78	9.91	15.87%	8.28	29.71%
Binary Counter		236.30	217.33	8.03%	126.96	46.26%

Table 1: Experimental Results

Table 1 shows, for each example, the throughput and latency improvements over the unoptimized Balsa circuits. The cycle time is reported for the systolic counter and programmable counter, while for the FIFO, the put and get cycle times, as well as the latency of a data item from input to output through an empty FIFO are reported. The "Previous Transforms" column indicates the results when only the previously-introduced transforms (Activation Channel Removal and Call Distribution [5]) are applied. The "All Transforms" column indicates the results when applying both the previous transforms and the newly-introduced transforms. For each example, the following transforms were successfully applied: *Systolic Counter* – Loop Enabling Transform; *Low-Latency FIFO* – Seq Replacement, Loop Enabling, Data Sampling, and Passive Output Port; and *Binary Counter* – Protocol Reversal on Functional Units, Seq Replacement. The performance improvements for on the simulated examples range up to 54%. Furthermore, the improvements using the new transforms are significantly better than those using only previous techniques [5].

9. Conclusions and Future Work

This paper introduces a powerful new set of transformations, and an extended channel-based language to support them, which can be used in an optimizing back-end for Balsa. The transforms described in this paper fall into two categories: resynthesis and peephole. Results on several substantial examples indicate significant performance improvements over unoptimized Balsa implementations, as well as over the implementations using only previous control-oriented transforms.

The current approach is still limited to supplying primitive transforms (somewhat analogous to the operations in `SIS` for logic optimization), and does not yet present systematic scripts or algorithms for their optimal application. This is a topic for future work.

10. REFERENCES

[1] A. Bardsley and D. Edwards, "Compiling the Language Balsa to Delay-Insensitive Hardware", *Hardware Description Languages and their Applications (CHDL)*, April 1997, pp. 89-91.

[2] A. Bardsley, "Implementing Balsa Handshake Circuits", PhD. Thesis, Department of Computer Science, University of Manchester, 2000.

[3] A. Bardsley, Personal Communication, 2000.

[4] E. Brunvand, "Translating Concurrent Communicating Programs into Asynchronous Circuits", Technical Report CMU-CS-91-198, Carnegie Mellon University, 1991.

[5] T. Chelcea, A. Bardsley, D. Edwars, and S.M. Nowick, "A Burst-Mode Oriented Back-End for the Balsa Synthesis System", Proceedings of *Design Automation and Test in Europe*, March 2002.

[6] T. Kolks, S. Vercauteren, and B. Lin, "Control Resynthesis for Control-Dominated Asynchronous Design", *IEEE ASYNC'96 Symposium*, pp. 233-243.

[7] R.M. Fuhrer and S.M. Nowick, "Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools", Kluwer Academic Press, 2001.

[8] 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 Institute on Concurrent Programming, Addison-Wesley, 1990, pp. 1-64.

[9] S.M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers", Technical Report CSL-TR-95-686, Stanford University, March 1993.

[10] A.M.G. Peeters, "Single–Rail Handshake Circuits", PhD. Thesis, Department of Computer Science, Technical University of Eindhoven, 1996.

[11] M.A. Pena and J. Cortadella, "Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits", *IEEE ASYNC'96 Symposium*, pp. 222-232.

[12] L.A. Plana and S.M. Nowick, "Architectural Optimization for Low-Power Nonpipelined Asynchronous Systems", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 6, No. 1, March 1998.

[13] K. van Berkel, "Handshake Circuits: an Intermediary between Communicating Processes and VLSI", PhD. Thesis, Department of Computer Science, Technical University of Eindhoven, 1992.

[14] K. van Berkel, "Handshake Circuits: an Asynchronous Architecture for VLSI Programming", *International Series on Parallel Computation*, Vol. 5, Cambridge University Press, 1993.

[15] K. van Berkel, Personal Communication, 1999.