Balsa-CUBE: an Optimizing Back-End for the Balsa Synthesis System [*]

Tiberiu Chelcea[1], Steven M. Nowick[1], Andrew Bardsley[2], Doug Edwards[2]
[1]Department of Computer Science
Columbia University
{tibi,nowick}@cs.columbia.edu
[2]Department of Computer Science
University of Manchester
{bardsley,doug}@cs.man.ac.uk

Abstract

Several approaches have been proposed for the syntax-directed compilation of asynchronous circuits from high-level specification languages, such as Balsa [1] and Tangram [13, 10]. 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 describes a new back-end optimizer (Balsa-CUBE, short for "Balsa-Columbia University Back-End") for an existing asynchronous CAD synthesis package, Balsa. The back-end incorporates several powerful transformations for the optimization of large-scale asynchronous systems. The transforms fall into two categories: peephole and resynthesis. Peephole optimizations replace existing configurations of components in a template-based fashion by other configurations of components. In contrast, resynthesis optimizations modify and resynthesize a collection of components in a non-template based fashion. To facilitate the application of these transforms, a new asynchronous component modeling language, called CH, is also introduced. All the proposed transforms are captured as simple language manipulation procedures in CH. The design flow includes an established tool for asynchronous controller synthesis, Minimalist [8], which synthesizes two-level logic implementations and includes a number of options for used design-space exploration, and an established synchronous technology-mapping tool, Synopsys Design Compiler [5]. Experimental results on several substantial design examples indicate system-level performance improvements of up to 54%.

1 Introduction

Several approaches have been proposed for compilation of asynchronous circuits from high-level specification languages. Hardware compilers such as Balsa [1] (from University of Manchester) and 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 directly mapped into VLSI circuits using a template-based approach. An *occam*-based compiler [2] and an alternative translation approach from Caltech [9] have also been proposed.

Balsa and Tangram have been widely-used, but their syntax-directed translation methods introduce significant performance overheads. 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 a few limited 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 presents a new automated, optimizing back-end ("Balsa-CUBE"), integrated in the Balsa synthesis system [1]. The back-end also includes the Minimalist CAD package [8] for burst-mode controller synthesis, and the Synopsys' Design Compiler for technology mapping. The back-end includes a complete hazard verifier, which verifies top-level controller specifications against the final technology-mapped implementation, based on 9-value hazard simulation [7]. The integrated design flow was applied to several substantial asynchronous design examples. Pre-layout back-annotated Verilog simulations on technology-mapped implementations indicate up to 54% speed improvement over the unoptimized implementations. The proposed back-end is the first integrated asynchronous design flow for large-scale systems which incorporates a significant number of powerful optimizing transforms.

The optimizing back-end incorporates two major contributions. First, a set of transformations is used to optimize large-scale asynchronous circuits. These transforms (introduced in [4, 3]) fall into two categories: *peephole* and *resynthesis*. A peephole optimization optimizes components within a sliding window; if the netlist of components in the window conforms to a pattern, they are replaced in a template-based fashion by other existing components. In contrast, resynthesis optimizations attempt to replace groups of existing components in a non-template-based fashion, by re-synthesizing them. Several of the proposed transforms are behavior-preserving [3], while some are behavior-modifying [4]. Some of the transforms are entirely new, including ones which perform radical changes on the interfaces of functional units (Protocol Reversal [4]). Other proposed optimizations are clustering transforms, somewhat analogous to the transforms presented in [11, 6]. However, these earlier transforms perform unlimited clustering and thus have limited practicality for optimizing large-scale asynchronous systems. In contrast, our clustering transforms identify several new and distinct subcategories for *limited* clustering (Activation Channel Removal, Call Folding and Call Distribution [3]), which are shown to

*This work was supported by NSF ITR Award No. NSF-CCR-0086036 and NSF Award No. CCR-99-88241, a grant from the New York State Microelectronics Design Center, and by ESPRC Grant No. GR/N19618.

be effective in practice for both performance improvements and synthesis run time. Finally, some of the transforms are analogous to “handshake reshuffling” [9]. However, in our new approach, unlike in [9], these transforms are captured in a higher channel-based component language, avoiding the need to explicitly specify every low-level signal transition.

The second contribution of this paper is a new asynchronous component specification language, called CH. The CH language is very important in the proposed approach: each transform is formalized as a simple and efficient language manipulation procedure in CH. Thus, CH provides a more natural and concise mechanism for exploring various design tradeoffs. The resulting specifications are thus independent of the synthesizable low-level specifications into which CH is translated. Burst-Mode controller specifications are currently employed [8], but they are just one of several possible low-level specification styles.

As indicated above, the proposed transforms fall into two categories: behavior-preserving and behavior-non-preserving. Nearly half of the transforms are behavior-preserving (four out of nine) and have been formally verified: Activation Channel Removal, Call Folding, Call Distribution, and Protocol Reversal. These include all of the results reported in the Results section for “clustering transforms”. The remaining transforms are behavior-non-preserving: Enc Replacement, Seq Replacement, Data Sampling, Passive Output Port, and Loop Enabling. Since these modify system-level concurrency, care must be taken to insure no data or structural hazards, or deadlock result from their application. As work in progress, we are currently formalizing a set of safe constraints which ensure the safe application of such transforms. In addition, the existing Balsa simulator is currently being extended to handle CH expressions and to verify, on-the-fly, the safety of the new transforms whenever they are applied.

2 Overview of the Approach

This section presents a brief overview of the proposed approach, including the optimization strategies and the new integrated tool flow, as well as a comparison to related work.

2.1 Proposed Transformations

A graphical depiction of the proposed optimization approach is shown in Fig. 1 for two abstract asynchronous systems. For each system, the “Before” configuration shows the system before applying any transformations, while the “After” configuration shows the same system after applying transformations.

The system in Fig. 1a is partitioned into datapath and control, and consists of a collection of asynchronous components, called *handshake components* [13, 10]. Each component is a primitive concurrent asynchronous process, typically implemented by a few gates. The components are connected by arcs representing *communication channels*. These channels are implemented by a pair of request and acknowledge wires, and possibly datapath wires, where communication is implemented by a *four-phase handshaking protocol*.

There are several types of proposed transforms in the new back-end. Some transforms cluster together only control handshake components into larger controllers (1), while others (2) cluster together small configurations of mixed control and datapath components (Fig. 1a). Some

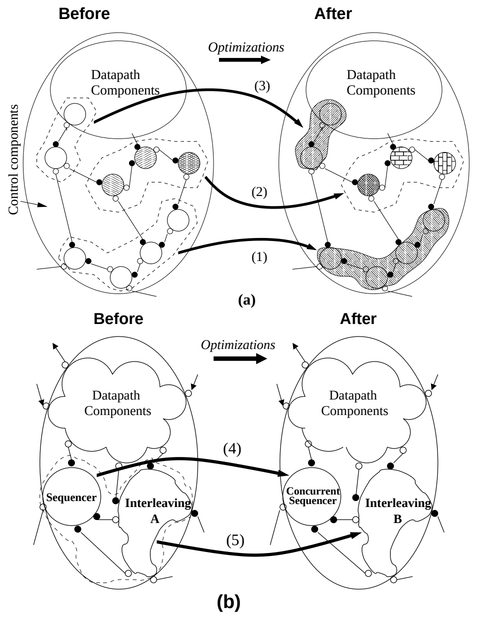

Figure 1: Overview of the Transformations

transforms (3) simply take a fixed configuration of handshake components in a window and replace it in a template-based fashion by a new configuration of existing components. Finally, other transforms manipulate the behavior of individual components (Fig. 1b); these can either be single primitive components or clustered components which result after applying earlier optimizations. These other transforms either (4) introduce more concurrency in the sequential execution of systems components (as shown, the “Sequencer” component becomes “Concurrent Sequencer”), or (5) modify the interleaving of handshakes on the component’s interfaces (as shown, *interleaving A* becomes *interleaving B* for the clustered component).

2.2 Integrated Tool Flow

Balsa [1] is collection of programs developed at University of Manchester, that facilitate the description and synthesis of large-scale 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 typically a simple primitive concurrent process, often containing only a few gates. The handshake components are then each individually mapped to actual VLSI circuits in a template-based fashion.

The tool flow for the new back-end is shown in Fig. 2; it now includes an *optimization step*, which consists of the proposed set of resynthesis and peephole transforms. The shaded boxes indicate new research contributions.

The new back-end takes the unoptimized 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 into datapath and control components. The datapath components are then synthesized using the existing Balsa system technology mapper (balsa-netlist [1]). The control components are first translated into Burst-Mode Specifications [8] and then synthesized. A CH expression for a control component is first translated into an intermediate form, which contains signal transitions and labels for states, goto statements and keywords indicating external input choices.

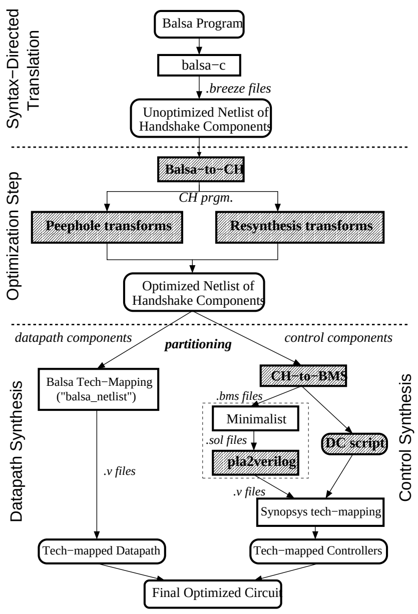

Figure 2: New Balsa System Design Flow

The intermediate form is then linearly traversed to extract the Burst-Mode specification. These specifications are then synthesized into hazard-free logic implementations using the Minimalist Burst-Mode CAD package [8], with speed-oriented scripts. Finally, the logic implementations are then technology-mapped using a commercial tool (Synopsys' Design Compiler [5]). The optimization flow also includes a hazard verifier, which compares the burst-mode controller specification with the technology-mapped netlist, using a 9-valued simulation based on Kung's approach [7].

2.3 Related Work

Peephole optimization and control resynthesis techniques for asynchronous systems have been previously proposed. Previous peephole optimizations [13, 2] have typically been *behavior-preserving*, and most have dealt with simple components and improvements (such as redundancy removal). In contrast, many of the newly 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 behavior preserving. In contrast, the proposed resynthesis techniques include more powerful transformations for design-space exploration, including concurrency enhancement (Enc-Replacement, Seq-Replacement [4]) and protocol manipulation (Protocol Reversal [4]). Furthermore, extensions to a component modeling language, CH, are proposed which formalize and facilitate these manipulations.

A final difference is that, unlike the previous approaches, the new transforms are being integrated into a comprehensive CAD package for asynchronous synthesis, Balsa, which provides an entire design flow from a high-level language down to layout. Furthermore, these previous resynthesis approaches did not report results on performance improvements, only on area.

3 Case Study: Binary Counter

This section discusses in more detail the optimization of an asynchronous system, a binary counter. The counter receives a positive integer value from the environment; it then counts down from the received value down to zero, writing each intermediate value back to the environment.

Figure (3 *before*) shows the unoptimized implementation of the binary counter. The control components (i.e. the components that control the flow of the program) are represented by diamond shapes; the datapath components (i.e. the components that implement the data computation) are represented by oval shapes. The connections between components are either channels (represented by thick lines) or wires (represented by thin lines). It is interesting to notice that some Balsa components (such as While, in this example), are internally represented in Balsa-CUBE by more controllers (in this case two: `count_bin_m3` and `count_bin_m4`). This happens because burst-mode specifications cannot directly handle bundled data; thus, for the While component, a slave controller `count_bin_m4` simply transforms bundled data into dual-rail data, which can then be used directly by the master controller `count_bin_m3`.

Figure (3 *after*) shows the optimized implementation of the same system. As before, the control and datapath components are identified by diamond and oval shapes, respectively. In addition, the components optimized through peephole transforms are shown in octogonal shapes.

The system was optimized by applying two transforms: Activation Channel Removal [3] (three times) and Protocol Reversal [4] (two times). For each transform, the initial set of components on which each transform is applied is identified in the *before* configuration; the thick labeled arrows lead to the resulting components. Activation Channel Removal clusters together control components; for this system, four initial control components are clustered into a single final control component. Protocol Reversal switches the port type for unary functional units; it thus moves the computation performed by such units from the critical path into background. For the binary counter, Protocol Reversal has been applied for two separate configurations.

4 Results

The entire synthesis flow has been tested for a number of examples. Each example was described using the Balsa language, and synthesized with `balsa-c`, to obtain the initial netlist of handshake components. These components were then read into the optimization program proposed in the paper to obtain the optimized netlist of components. Each component was then synthesized and technology mapped into the AMS 0.35μm library. The final implementations were back-annotated using `pearl`, and simulated with Cadence Verilog-XL.

This section discusses the results of applying the new back-end to six designs: a four-handshake systolic counter, a programmable eight-bit binary counter, an eight-place wagging register, an eight-bit word, three-place low-latency FIFO, an eight-bit, three-place stack, and a 32-bit microprocessor core.

Table 1 shows, for each example, three sets of results. The "Balsa" column shows the area and speed (throughput and/or latency) of the basic Balsa designs without any optimizations. The "Clustering Transforms" and "All Transforms" columns show the results after applying only clustering transforms, and after applying all transforms, respectively, to each design. These two last columns indicate the total area and area overheads, and speed (latency or throughput, as indicated) and the improvements in speed

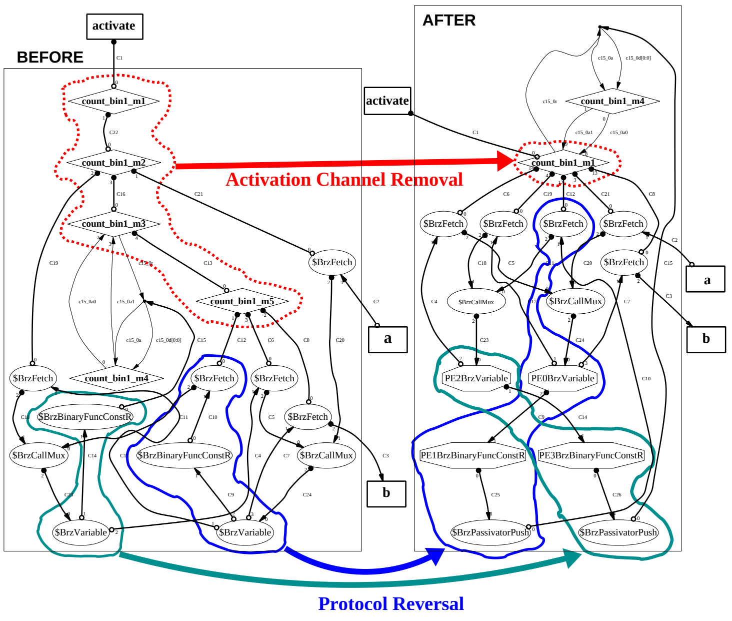

Figure 3: Binary Counter: Initial and Optimized Implementations

Examples	Balsa		Clustering Transforms					All Transforms			
	(No Optimizations)										
	Area	Speed	Synthesis	Area		Speed		Area		Speed	
	(μm^2)	(ns)	Run Time	(μm^2)	Overhead	(ns)	Improve	(μm^2)	Overhead	(ns)	Improve
Systolic Counter	27.84	24.81	0.8 s	35.46	27.37%	16.06	35.26%	24.91	-10.52%	11.28	54.50%
Binary Counter	113.04	236.30	1.9 s	121.58	7.55%	217.33	8.03%	118.34	4.69%	126.96	46.26%
Wagging Register	228.93	49.82	2.7 s	283.71	23.93%	42.43	14.83%	298.15	30.24%	34.40	30.95%
FIFO **latency**		17.33				15.19	12.36%			10.32	40.45%
FIFO **put cycle time**	205.08	8.41	17.2 s	334.72	63.21%	8.06	4.16%	312.32	52.29%	6.22	26.04%
FIFO **get cycle time**		11.78				9.91	15.87%			8.28	29.71%
Stack	99.07	121.58	2.9 s	119.52	20.64%	107.70	11.40%	111.84	12.88%	62.94	48.23%
Microprocessor Core	453.76	66.48	3h 17m 0s	563.47	24.18%	60.65	8.76%	589.18	29.18%	58.14	12.55%

Table 1: Experimental Results - Applying Groups of Transforms

for each design; in addition, the "Clustering Transforms" column also shows the synthesis run time for Minimalist.

For each example, the peephole optimizations are applied first, and then the resynthesis optimizations.[1] The following transforms were successfully applied: *Systolic Counter* – Activation Channel Removal, Call Distribution, Loop Enabling Transform; *Binary Counter* – Activation Channel Removal, Protocol Reversal on Functional Units, Seq Replacement; *Wagging Register* – Activation Channel Removal, Seq Replacement; *Low-Latency FIFO* – Activation Channel Removal, Seq Replacement, Loop Enabling, Data Sampling, and Passive Output Port; *Stack* – Activation Channel Removal, Loop Enabling, Passive Output Port; *Microprocessor Core* – Activation Channel Removal, Call Distribution, Call Folding, Passive Output Port.

References

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

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

[3] T. Chelcea, A. Bardsley, D. Edwards, and S.M. Nowick, "A Burst-Mode Oriented Back-End for the Balsa Synthesis System", Proceedings of *Design Automation and Test in Europe*, pp. 330-337, March 2002.

[4] T. Chelcea, S.M. Nowick, "Resynthesis and Peephole Transformations for the Optimization of Large-Scale Asynchronous Systems", Proceedings of the *Design Automation Conference*, pp. 405-410, June 2002.

[5] "Design Compiler Family Datasheet", http://www.synopsys.com/products/logic/design_comp_ds.html.

[6] T. Kolks, S. Vercauteren, and B. Lin, "Control Resynthesis for Control-Dominated Asynchronous Design", Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 233-243, 1996.

[7] D.S. Kung, "Hazard-non-increasing gate-level optimization algorithms", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 631-634, 1992.

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

[9] 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.

[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.

[1] Currently, this application order of the transforms is the default; however, Balsa-CUBE allows the users to select which transforms are applied, and the order in which they are applied. In the future, it will be beneficial to conduct a wide-ranging set of detailed experiments to evaluate the best order of applying transforms, as well as to evaluate area overheads.