Parallel Tools for Asynchronous VLSI Systems

Yi-Shan Lu*, Samira Ataei†, Jiayuan He*, Wenmian Hua†, Sepideh Maleki*, Yihang Yang†,
Martin Burtscher‡, Keshav Pingali*, and Rajit Manohar†

*University of Texas at Austin
{yishanlu,hejy,smaleki,pingali}@cs.utexas.edu

‡Texas State University
burtscher@txstate.edu

†Yale University
{samira.ataei,wenmian.hua,yihang.yang,rajit.manohar}@yale.edu

***Abstract*—We propose to develop a collection of electronic design automation tools for asynchronous circuits. To reduce design turn-around time, we will implement parallel versions of the key algorithms using the Galois system. These tools will be open-sourced when mature.**

I. Introduction

Scalable computer systems are designed as a collection of modular components that communicate through well-defined interfaces. The interfaces must be robust to delays and uncertainty in the physical implementation of communication. This view applies to computer systems at many levels of abstraction. The Internet is a collection of communicating computers with message-passing through the Internet protocol. A modern datacenter is a collection of servers that communicate via message-passing over commodity network hardware. Even large software systems consist of a collection of modules that use well-defined application programming interfaces (APIs) to communicate. Almost all computer systems disciplines have made the wise choice to partition their problem into components that communicate via protocols that *are independent of their physical realization—such as timing, energy, or size.*

However, in current chip designs, this modular approach is abandoned in favor of global synchrony. A global synchronization signal (the "clock") dictates the time budget for every step of the computation—regardless of *what* is being computed.

Although this clocked design paradigm dominates the design of computers today, engineers are struggling to preserve the fiction of simultaneity required by the clock, even within an individual chip. This struggle is an inevitable result of advancing technology. As transistors get smaller and faster, the delay of communication over wires dominates the cost of local computation with transistors. Such progress renders the clocked paradigm a poorer and poorer abstraction for chip design. Modern application-specific integrated chips (ASICs) are designed as a collection of small clocked "islands" that communicate via interfaces that break the clocking abstraction.

We propose to develop a collection of electronic design automation (EDA) tools that isolate the designer from the details of the physical implementation technology, especially when it comes to delays and timing uncertainty.[1] The approach is based on an *asynchronous*, *modular* and *hierarchical* design methodology for complex chips, and it permits component re-use from one technology to another with little or no modification. While individual (small) modules of the chip could be clocked, the overall system uses an asynchronous integration approach to achieve modular composition.

To reduce the turn-around time for designs, we will implement parallel versions of the key algorithms in this tool chain, using the Galois system described in Section III. The Galois system supports parallelization of irregular algorithms such as those in which the key data structures are graphs and hyper-graphs. Since circuits can be viewed as hyper-graphs, the Galois system is a good platform for this parallelization effort.

II. EDA for Asynchronous VLSI Systems

The phrase "asynchronous digital circuits" refers to a large family of circuits that do not use a global clock signal to sequence steps of the computation. There are many asynchronous logic families, ranging from circuits that are highly robust to timing uncertainty to those that rely on strict timing constraints for correct operation. While the range of possible circuit families is large, they all share some common features.

First, the set of *gates*, or building-blocks, are more general than standard combinational logic and flip-flops. One of the most commonly used circuit elements is the C-element, which is a state-holding gate whose output changes only when the two inputs are equal. Second, the performance of these circuits is governed by the delays of *cycles* of gates. A special case where this is clear is one where the circuit contains an odd number of inverters to create an oscillator (the clock), and the oscillator is used to control flip-flops. This is a traditional synchronous circuit, and the performance is governed by the delay of the cycle of inverters. In general, the cycles are more complicated, and so the performance analysis problem is very different from that of the clocked domain. Third, correct

[1]It is not possible to entirely decouple the logical correctness of a design from timing to create completely delay-insensitive circuits [12, 15]. However, it is possible to make a very mild and local timing assumption that is easy to satisfy in practice [11].

operation of an asynchronous circuit requires relative path delay timing constraints on some sets of paths.

The open-source design flow we plan to create starts with the circuit specified using the Communicating Hardware Processes (CHP) programming language [14], which is based on Hoare's CSP [4]. The language is a simple sequential programming language augmented with communication channels and send, receive, and probe operations. The next step in the design flow is to decompose the CHP into a massively parallel collection of processes that cooperate to perform the computation specified by the original program. This (large) step is where all the "micro-architectural" decisions are made. This transformation is performed in a hierarchical manner as a sequence of small changes, where each change can be easily verified. A number of well-established techniques can be used for this purpose, including techniques such as process decomposition and projection [13, 16]. After this step, data encoding and protocol choices are made so that every statement in each CHP process can be represented as a collection of operations on Boolean-valued variables. After this step, the program is in handshaking expansion (or HSE) form. Finally, each individual component is converted into a circuit based on the logic family being used, and the concurrent composition of all the circuits implement the original specification. The core idea behind this synthesis procedure was developed in the 1980s [14].

The flow we plan to develop includes a design language called ACT (for asynchronous circuit toolkit). This is a hierarchical design language that includes communication channels as first-class objects. The language supports representing circuits at multiple levels of abstraction, including CHP, HSE, gate-level, and transistor-level descriptions. By using an integrated language, we preserve the relationships between different levels of abstraction in the design throughout the design flow. Design tools can be viewed as transformations in the ACT framework. For example, logic synthesis elaborates a CHP-level description of a module into a gate-level description of the *same module* without changing its interface. A summary of the design flow and linkages to standard, commercially used file formats is shown in Figure 1.

This language is the result of an evolution over almost three decades of research in asynchronous design grounded in the implementation of over a dozen asynchronous VLSI chips ranging in complexity from 0.5M transistors to 5.4B transistors, and in technologies ranging from 0.6μm CMOS to 28nm CMOS. We also plan to develop linkages to and from the ACT language to standard commercially-used languages and formats such as Verilog and SPICE. This will allow interoperability between our flow and commercial flows whenever possible.

We plan to develop key components of an open-source EDA flow for asynchronous circuits that implements several of the unique aspects of the design flow. Some of the major components include:

- A cell generator leveraging our previous work in supporting non-standard logic gates [6];

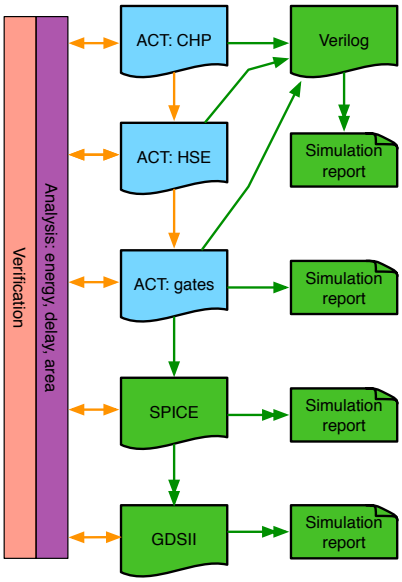

Fig. 1: Overview of the design flow for asynchronous circuits that is under development.

- A tool that can import synchronous logic into an asynchronous framework to support synchronous module integration;
- Timing analysis for asynchronous circuits based on recently developed analyses [1, 5];
- An asynchronous memory compiler based on the OpenRAM framework [2];
- A timing-aware placement and routing flow that respects the timing constraints required for the correct operation of asynchronous logic and extends previous work that only supported a small class of timing requirements [7].

In support of the flow, we also plan to develop formal equivalence checking tools across different levels of abstraction (e.g., CHP versus HSE and HSE versus PRS) using a combination of traditional model checking and *inverse synthesis*, which attempts to "undo" steps in the design flow [9, 10].

The modular nature of the asynchronous design flow makes it inherently suitable for parallelization. Module-level parallelism is easy to achieve, and the high complexity of some of the tasks (e.g., precise logic synthesis requires a form of state-space exploration) requires parallelism to speed up the design process. We plan to use the Galois framework (described next) as a way to simplify the development of highly parallel EDA tools.

III. Parallel EDA flow

Since circuits can be viewed abstractly as graphs and hypergraphs, a system for supporting the design and implementation of a parallel EDA tool-chain must have the following characteristics.

- It must support clean abstractions for reasoning about and expressing the available parallelism in graph (and hypergraph) algorithms.

- It must hide parallelization details such as synchronization from EDA algorithm designers.
- It must be scalable; as long as the algorithm has sufficient parallelism, performance should improve if more cores are used.

A. Operator formulation of algorithms

A clean abstraction for expressing parallelism in graph algorithms is the *operator formulation*, a *data-centric* abstraction in which algorithms are described as a composition of a *local view* and a *global view* of the computation.

The local view is described by an *operator*, which is a graph update rule applied to an *active node* in the graph (some algorithms have active edges). Each operator application, called an *activity* or *action*, reads and writes a small region of the graph around the active node, called the *neighborhood* of that activity. An active node becomes inactive once the activity is completed. *Morph* operators can modify the graph structure of the neighborhood by adding and removing nodes and edges. AIG rewriting [21] deploys morph operators. *Label computation* operators, in contrast, only update labels on nodes and edges without changing the graph structure. FPGA routing [17], formulated as an SSSP problem within a routing resource graph, uses label computation operators.

The global view of a graph algorithm is captured by the location of active nodes and the order in which activities must appear to be performed. Topology-driven algorithms make a number of sweeps over the graph until some convergence criterion is met, e.g., the Bellman-Ford SSSP algorithm. Data-driven algorithms begin with an initial set of active nodes, and other nodes may become active on the fly when activities are executed. They terminate when there are no more active nodes. Dijkstra's SSSP algorithm is a data-driven algorithm. The second dimension of the global view of algorithms is *ordering* [3]. Activities in *unordered* algorithms such as SSSP can be performed in any order without violating program semantics, although some orders may be more efficient than others.

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. The resulting parallelism is called *amorphous* data-parallelism. It is a generalization of the standard notion of data-parallelism [20].

B. Galois system

The Galois system implements this data-centric programming model (see details in [18]). Application programmers write programs in sequential C++, using certain programming patterns to highlight opportunities for exploiting amorphous data-parallelism. The Galois system provides a library of concurrent data structures, such as parallel graph and work-list implementations, and a runtime system. The data structures and runtime system ensure that each activity appears to execute atomically. In this way, the Galois system encapsulates parallelization details and realizes performance scalability at the same time.

The Galois system has been used to implement parallel programs for many problem domains including finite-element simulations, n-body methods, graph analytics, intrusion detection in networks [8], FPGA routing [17], and AIG rewriting [21].

C. Galois for open-source EDA flow

1) Framework for EDA tool builders: We propose a parallel programming framework customized for EDA algorithms that is based on the Galois system. The parallelization can be among modules in a hierarchical design or within a module for a specific design stage such as placement. In this framework, we will provide the following building blocks for EDA tool builders to quickly prototype EDA algorithms without worrying about details of parallelization:

- A set of parallelization-aware data structures, e.g., gate-level net-lists, chip layouts, cell libraries.
- A set of operators common to EDA algorithms, e.g., the operator for SSSP in routing algorithms or the delay computation and time propagation in timing analysis.
- A set of schedulers frequently used in EDA algorithms, e.g., topological-order execution in timing analysis.

The above programming constructs can be parameterized or instantiated to fit specific needs. For example, router designers need to carry out only the following steps: (1) instantiate a placed net-list and read in technology information from, for example, a cell library; and (2) specify, according to the quality metrics being considered, the cost function for the SSSP operator and the schedule/priority function for applying the operators. Then they can evaluate the resulting router and improve the quality of the results iteratively.

By providing building blocks at the proper abstraction level, our proposed approach will support parallelization seamlessly for EDA algorithms. Deterministic execution can be provided when required [19]. The framework will also provide means for tool designers to specify their own operators and schedules when such needs arise.

2) Tools for chip designers: For chip designers, we plan to prototype a parallelized flow for timing-driven physical design, the most time-consuming step in chip design processes. There are a set of well-established core algorithms for the major components in standard EDA tools. State-of-the-art EDA tools combine these algorithms with sophisticated heuristics that are guided by benchmarks to achieve excellent quality of results within a reasonable runtime budget. Our goal in developing parallel tools is to demonstrate the effectiveness of the Galois framework in speeding up the core algorithms used in different parts of a timing-driven physical design flow.

Given a gate-level net-list, chip area and process technology information, for instance, liberty or LEF/DEF files, our proposed flow will generate a mostly design-rule-clean layout with the following components:

- A static timing analysis engine that provides timing information at various design stages.

- A placer that assigns locations for gates to meet constraints in timing, routability, density/utilization, etc. and optimizes for timing.
- A router that connects pins of nets subject to design constraints and optimizes for timing.
- A gate sizer that adjusts driving strengths of gates to meet timing constraints in the physical design flow.

The above components will support both synchronous designs with one clock domain and asynchronous designs.

The components will have built-in parallelization, so chip designers can enjoy the speedup from parallelization by just providing the number of threads they want to use. In addition, users can use scripts to create any iterative flow they wish or start from any point in the physical design such as from routing given a placed net-list.

IV. Conclusion

We presented a plan for implementing a parallelized physical design flow for asynchronous circuits. The flow supports modular design and parallelization naturally. We believe that this tool chain will promote chip design with clean abstractions and fast turn-around times.

References

[1] S M Burns and A J Martin. Performance analysis and optimization of asynchronous circuits. 1990.

[2] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. Openram: An open-source memory compiler. In *Proc. 35th International Conference on Computer-Aided Design*, pages 93:1–93:6, 2016.

[3] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

[4] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[5] Wenmian Hua and Rajit Manohar. Exact timing analysis for asynchronous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):203–216, 2018.

[6] Robert Karmazin, Carlos Tadeo Ortega Otero, and Rajit Manohar. celltk: Automated layout for asynchronous circuits with nonstandard cells. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 58–66. IEEE, 2013.

[7] Robert Karmazin, Stephen Longfield, Carlos Tadeo Ortega Otero, and Rajit Manohar. Timing driven placement for quasi delay-insensitive circuits. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 45–52. IEEE, 2015.

[8] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Commun. ACM*, 59(5):78–87, April 2016.

[9] Stephen James Longfield and Rajit Manohar. Inverting martin synthesis for verification. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 150–157. IEEE, 2013.

[10] Stephen Longfield Jr and Rajit Manohar. Removing concurrency for rapid functional verification. In *Proc. 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 332–339. IEEE Press, 2014.

[11] Rajit Manohar and Yoram Moses. Analyzing isochronic forks with potential causality. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 69–76. IEEE, 2015.

[12] Rajit Manohar and Yoram Moses. The eventual c-element theorem for delay-insensitive asynchronous circuits. In *Asynchronous Circuits and Systems (ASYNC), 2017 23rd IEEE International Symposium on*, pages 102–109. IEEE, 2017.

[13] Rajit Manohar, Tak-Kwan Lee, and Alain J Martin. Projection: A synthesis technique for concurrent systems. In *IEEE International Symposium on Asynchronous Circuits and Systems*, page 125. IEEE, 1999.

[14] Alain J Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed computing*, 1 (4):226–234, 1986.

[15] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278, 1990.

[16] Alain J Martin. Synthesis of asynchronous vlsi circuits. Technical Report CS-TR-93-28, California Institute of Technology, 1993.

[17] Yehdhih Ould Mohammed Moctar and Phillip Brisk. Parallel fpga routing based on the operator formulation. In *DAC '14: Design Automation Conference*, 2014.

[18] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[19] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[20] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI 2011*, pages 12–25, 2011. doi: 10.1145/1993498.1993501.

[21] Vinicius Possani, Yi-Shan Lu, Alan Mishchenko, Keshav Pingali, Renato Ribas, and Andre Reis. Unlocking fine-grain parallelism for aig rewriting. In *ICCAD '18: International Conference on Computer Aided Design*, 2018.