Improved Logic Synthesis for Asynchronous Circuits

Karthi Srinivasan and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520

{karthi.srinivasan, rajit.manohar} at yale.edu

***Abstract*—Synthesis of asynchronous circuits from CHP programs has seen significant improvements in recent times. We present several improvements and extensions to the state-of-the-art synthesis technique in order to further improve the performance of generated circuits and also provide designers with more power and flexibility. The proposed modifications are benchmarked against the baseline with pre-layout SPICE simulations of generated netlists. Comparison in a 65nm node show average improvements of 9% in area, 16% in delay and 23% in energy consumption.**

***Index Terms*—Asynchronous circuits, Logic Synthesis, High-level Synthesis, Micropipelines, Bundled Data Pipelines**

I. Introduction

Logic synthesis is the process of converting a high-level behavioral description of a circuit into a gate-level implementation of the same. In the context of asynchronous circuits, there have been several proposed methods over the years. One of the first methods to synthesize asynchronous state-machines used Huffman flow-tables [1]. Huffman flow-tables can be compiled into circuits by extending Karnaugh-map style logic synthesis, and by introducing delay lines to form a feedback loop to hold state. More recent methods include burst-mode design [2], which is closely related to synchronous design by virtue of its use of a local timing signal that served the role of the clock. Both methods assume bounds on the delay of gates, and hence require timing assumptions to ensure correct operation.

At the other end of the spectrum are delay-insensitive (DI) circuits, which assume unbounded delays on all gates and wires. However, this assumption greatly restricts the class of implementable functionality [3]. A slight relaxation is the quasi-delay insensitive (QDI) model, where wire delays are assumed to be small relative to a sequence of gate delays from an adversarial firing chain of gates, but still allows gate delays to be unbounded. Synthesis approaches for QDI circuits include Martin's synthesis method [4] of handshaking expressions (HSE) to circuits, and Petrify [5], which translates signal transition graphs to circuits. However, these methods require an input description of the system that is at the level of individual signal transitions. This is quite different from the level of abstraction that users of synchronous circuits are familiar with (e.g. using a + operator to denote an entire adder circuit). This makes the design of large systems challenging.

Naturally, there has been significant work in the area of synthesis from high-level descriptions. The current state-of-the-art in this regard is Maelstrom [6], an open-source tool, which synthesizes QDI control circuits with a bundled datapath, from a CHP (Communicating Hardware Processes) description, the syntax of which is described in Appendix A. Despite the advances provided by Maelstrom, there is still room for improvement and optimizations of the techniques and circuits used to translate CHP to circuits.

Section II summarizes the principal techniques behind the working of Maelstrom. The enhancements we implement are: latch minimization (Section III), controller optimization (Section IV), special probe circuits (Section V), better multiple channel access handling (Section VI) asymmetric delay line topologies (Section VII), and an edge-triggered datapath option (Section VIII). Section IX presents quantitative evaluations of our improvements, and Section XI concludes the paper.

II. Background

The goal of sequential synthesis is to convert a CHP program into an asynchronous circuit that implements the same computation while preserving the synchronization semantics of the program. In Maelstrom, the fundamental construct for synthesis is an infinite loop of a sequence of actions: $*[S_1; S_2; ...; S_n]$. The semantics of this program is to execute each action in the specified sequence, then go back to the first and repeat forever. This is conceptually a ring of control elements with a single token flowing through it, with the location of the token encoding the action currently being executed. The structure of each of the control elements varies based on the type of the action - assignment, send or receive. Each of the elements connect to the elements representing the actions before and after it via a simple request-acknowledge interface. Each element waits for an activation signal from the element before it, performs its action, then activates the element following it. Performing the action can correspond to either evaluating an expression and transmitting it out over a channel (send), capturing a value sent over a channel into a data-storage element (receive), or evaluating an expression and placing it into a data-storage element (assignment). Finally, there is a special element that ties up the start and end of the ring - the initial token element that activates the first action and waits for the last to complete.

In order to support multiple parallel actions, there are special parallel fork and join elements that activate multiple el-

ements/sequences of elements, that represent the sub-programs running in parallel, at once and wait for completion of all of them. For the selection construct too, the implementation evaluates and stores the guards of all the branches in parallel and activates the chosen one. At the end of the selection, the control flow is merged back. Nested loops are handled via pre-processing to extract these into parallel loops, similar to extracting a piece of code out into a function call, with appropriate communication between the two parts of sending the arguments and receiving the return values.

Maelstrom supports several control circuit families, including the QDI C-elements, MOUSETRAP and GasP. The principal sources of improvements in Maelstrom over prior approaches are the efficient control elements, combined with the use of pulsed latches for the datapath in a static token form-style configuration. This datapath style reduces the complexity of the datapath interface circuits such as pulse generators and read ports. However, there are still several improvements that can be made in several places: the control circuits, the datapath implementation, delay line architecture etc. In the following sections, we detail several such improvements.

III. Latch Elimination

The original datapath synthesis strategy in Maelstrom was built upon the simple assumption that every assignment of or receive into a particular variable must result in the generation of a storage element - a pulsed latch or QDI register depending on the chosen circuit family. While evidently correct, this results in far more storage elements than the minimum required to correctly implement a given CHP program. Consider the simple CHP program:

$$*[L_1?x; L_2?y; y := y + x; R_1!y; L_3?z; R_2!(z + y)]$$

Here, there are four sets of latches generated as there are four assignments/receives to variables in the program.[1] However, notice that the latch for the statement $y := y + x$ produces only a transient value that is used to compute values later in the program. Hence, the latches for this assignment can be replaced by wires, thereby significantly reducing the circuit complexity. The next natural question that arises is whether this sort of determination can be performed statically for all CHP programs. It is easy to say for linear programs, but is the same also true in the presence of complex control flow?

The datapath synthesis style in Maelstrom uses static single-assignment form, where variables are renamed such that each one in the program is assigned to, at most once. Further, a (synthesizable) program always consists of a set of initial conditions and a single loop that wraps an acyclic CHP fragment, i.e. it is of the form $\{x_i := v_i\}; *[true \rightarrow P]$ where the control flow inside P is acyclic. Hence, for every use (read of the value held in it in an expression) of a variable in P, we can iteratively trace its origins backwards until we either arrive at a receive or at an initial condition. Intuitively, this means that every value that is computed in a CHP program of this form must have come from a receive or from the value of another variable from the end of the previous iteration of the loop.

This means that it is unnecessary to generate latches for every assignment of a variable in the CHP, and instead it is sufficient to generate latches only for all the receives and all the loop-carried variables. All other internal assignments of the form $x := e$ simply consists of the combinational logic for e, the output of which is exposed in the places where originally the output of the latch for x would be exposed. The original latching strategy resulted in a number of latches (N_L):

$$N_L = \sum BW(C?x) + \sum BW(x := e) + \sum BW(x_i := v_i)$$

i.e. the sum of (in order) bitwidths of receives, assignments and initial conditions, where the second term usually dominates for programs of non-trivial sizes. v_i are the initial values for the variables x_i. This penalized computing intermediate values, which are necessary for writing readable code. The new latching strategy eliminates the second term entirely, and applying this optimization results in an improvement of the generated circuits across all metrics as several points in the program that were originally data store operations that required waiting for the latch to complete writing are now essentially instantaneous. This change also has the additional benefit of simplifying the control element for an assignment greatly. The original controller consisted of a 2-input C-element and 2 inverters. This can now simply be replaced by wires as well, making the controller zero-cost. As an example, consider the two CHP programs:

$$*[A?x; B?y; (t := x; x := y; y := t); C!(x - y)]$$

$$*[A?x; B?y; C!(y - x)]$$

In the original datapath synthesis strategy, the circuit for the first program would contain 3 additional sets of latches relative to the second, but the new strategy would result in *exactly* the same circuit for both programs. The intermediate assignments in the first program would get synthesized purely into crossover of wires, without no additional gates being generated. Latch elimination always results in fewer gates and an improved delay and thus we always apply this optimization.

IV. Optimized Control Elements

The original control circuit synthesis consisted of 3 elements, one each for the three elementary actions in CHP — assignments, sends and receives. As seen in the previous section, the element for the assignment can effectively be deleted. The original implementations of the other two elements (send, receive) were chosen to be the cheapest possible in terms of transistor count. However, this has the drawback of not allowing the channels to reset completely in parallel. In the four-phase synthesis with the original circuits, multiple processes that are interconnected end up having their reset phase synchronized with each other as a consequence of the controller structure. This results in added latency that can be avoided by allowing the reset to happen in parallel. The

[1] Alternatively, a latch and a mux can be used, as discussed in [6].

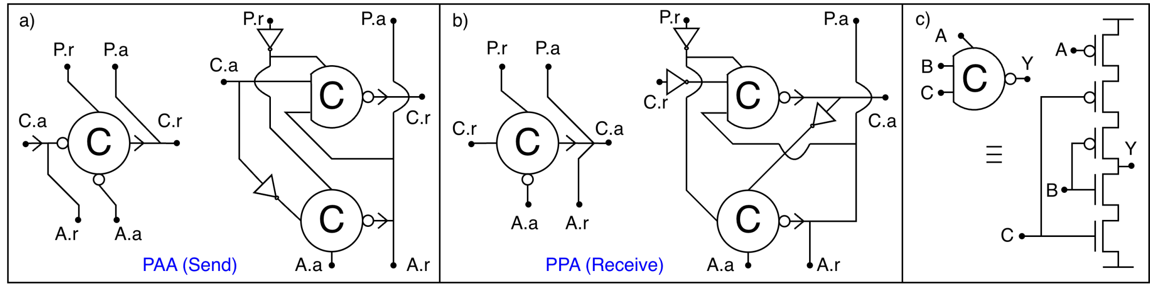

Fig. 1: Original (left) and improved (right) control elements for sends (a) and receives (b) for the four-phase C-element-based control circuit family. The new elements allow the reset half-phase of the channel handshake to happen in parallel, and consequently improve the cycle time of the circuit. c) Implementation of asymmetric C-element used in the new controllers.

new implementations of the send (PAA) and receive (PPA) elements are shown in Fig. 1. The ports labeled P and A connect to the previous and next elements in the ring, while the port labeled C is the channel port.

Consider the PAA element. Initially, all marked nodes are low. When $P.r$ is asserted, the top C-element fires, sending out $C.r$. Once $C.a$ arrives, the bottom C-element fires, asserting $A.r$ (and $P.a$). This transition also causes the top C-element to fire again, lowering $C.r$. In the reset phase, when $P.r$ is lowered, the transition is propagated down via the bottom C-element after appropriately waiting for the lowering of $C.a$, which will eventually happen as $C.r$ was lowered previously. Effectively, the activation of the downstream portion of the controller happens in parallel with the reset phase on the communication channel.

With these circuits, the channel can complete the reset phase (i.e. resetting the request and acknowledge wires to zero) in parallel with the execution of the remaining elements in the ring. Note that these elements are used only for the 4-phase datapath where the parallel reset completion is desirable. In the 2-phase datapath, there is no explicit reset phase, and as such, the original elements are efficient. Although the sizes of the controllers are now increased, this implementation actually exhibits lower latency due to distribution of load across the two gates. Further, the overall controller overhead for the entire program is now greatly reduced. Originally, the number of controller gates was proportional to the number of actions in the program, i.e. the sum of the number of receives, sends and assignments. With the improved method, it is now proportional to the number of channel accesses, which is typically a much smaller number. Section IX contains a detailed quantitative comparison of the improved circuits with the original.

V. Specialized Probe Circuits

Maelstrom2 now supports general synthesis of probes and negated probes. The circuit makes use of a killable arbiter [7] in order to safely implement a non-deterministic selection (NDS) that is compatible with the rest of Maelstrom2 's circuits. Fig. 2(a) shows the two-way non-deterministic selection circuit for a 4-phase datapath. The $ctrl$ port connects the circuit to the preceding part of the ring. The B_1 and B_2 ports connect to the two branches of the selection. The execution of the circuit is as follows: the asymmetric C-elements C_1 evaluate the guards when triggered by the $ctrl$ port. As both guards can simultaneously be true, we use a killable arbiter to pick one of the branches. The $ctrl$ port also activates the arbiter by raising the enable signal. The chosen branch then executes. Upon reset, the $ctrl$ port disables the arbiter which causes both its outputs to go low and trigger reset in the branches. These transitions are acknowledged via the OR-gate that generates the active-low kill-acknowledge (k_a) signal. The C_1 elements then set their output low. These two transitions are also acknowledged by the C_2 element which then propagates the lowering of $ctrl.a$ backwards. The necessity for complexity in this circuit arises from the fact that the guards may or may not become false once a branch is activated as a branch may contain a communication action on a channel that is probed, and the environment may change the state of the probe at any time.

For the 2-phase datapath case (circuit shown in Fig. 2(b)), there is more synchronization necessary as we do not have the luxury of a full, dedicated reset phase. The operation is as follows: Initially all gate outputs are low. The assertion of $ctrl.r$ enables the arbiter, which makes a decision. Upon the assertion of k_a, the latch blocking $ctrl.r$ opens and the latches blocking $B_i.r$ close. The delay line ensures that these latches close before the rising edge can reach the PL blocks (these are pulsed latches, which comprise of a latch and a pulse generator which creates a pulse on both the rising and falling edges of the input). The pulsing of the PL blocks latches the arbiter's decision and then disables the arbiter, via the XOR-gate feeding back to the en signal. Once the arbiter has been disabled, the latches open, activating the appropriate branch. On the next data-phase, which happens when $ctrl.r$ de-asserts, the reader may confirm that the same sequence of events repeats.

The general N-way NDS is built upon these two-way selection circuits. The circuit comprises of $N-1$ two-way NDS circuits with the second branch of one feeding into the control of the next one, effectively starting the evaluation of the next guard if the currently evaluated one was false. This effectively implements a sequential evaluation of the guards.

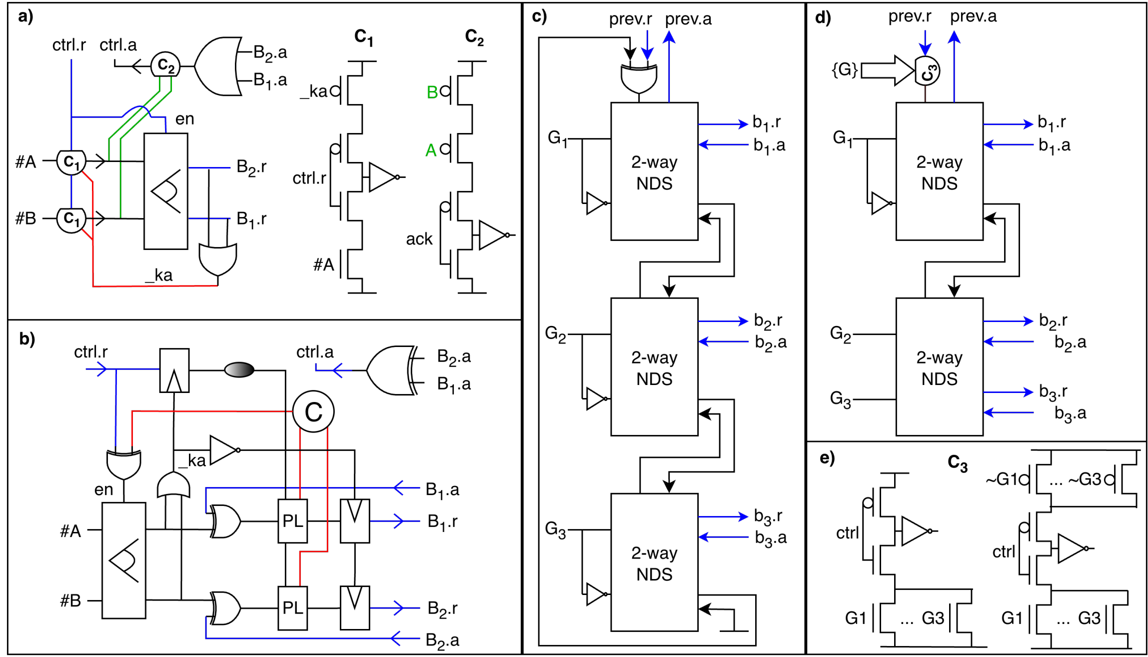

Fig. 2: Circuits for implementing general non-deterministic selections. (a) The killable arbiter-based implementation of a 2-way non deterministic selection for the 4-phase datapath. (b) The same as (a) for the 2-phase datapath. (c) Implementation of a general N-way non-deterministic selection where probes and negated probes may be present. (d) Implementation of an N-way non-deterministic selection when probes occur, but only in the positive sense. (e) Implementation of the $C3$ gate, for the 4-phase (left) and 2-phase (right) datapaths.

Note that there is an issue when dealing with negated probes in selections: it is possible that none of the guards are true when the circuit begins its evaluation. Further, it possible that a guard that was evaluated and deemed to be false can become true after the evaluation proceeded past it. This is only an issue in sequential evaluation of guards. The problem does not exist if concurrent evaluation of guards is performed, but the circuit complexity of this is much higher than the sequential case. Fig. 2(c) shows the general N-way NDS. The circuit implements a round-robin evaluation of the guards and picks the first true guard. If all the guards are false, it resets and re-evaluates them all in order again.

For the case of stable guards, i.e. there are no negated probes in any of the guard expressions, the above circuit can still be used but it is wasteful in terms of energy if all guards are false for a long time, due to multiple re-evaluations. In order to improve on this, we exploit the fact that with stable guards, once a guard is true, it cannot become false until the circuit performs an action that is observable to the environment. Hence, we can delay the evaluation until we know that at least one guard is true. The circuit in Fig. 2(d) contains a pre-checking circuit element ($C3$) that does precisely this, and fires when the control and at least one of the guards is true. This method also has the added benefit of reducing the number of arbiters required by one. Fig. 2(e) shows the exact implementation of $C3$ for the four-phase and two-phase datapaths.

Finally, consider the case of the single-branch selection: $[G \to S]$. This is an unnecessary construct if the guard G only consists of local variables, as we assume that there are no shared variables across processes. A process that contains this would either deadlock or always find G to be true on evaluation. Only the latter case is relevant and here the snippet can be replaced simply with S. However, if G contains probes, then it becomes a useful wait condition and we can implement this with no arbiters at all. Following the same notation as in Fig. 2(c,d), the production rule set (which can be trivially made CMOS-implementable) is simply:

$$prev.r \wedge G \;\to\; b1.r\!\uparrow \qquad\qquad b1.a \;\to\; prev.a\!\uparrow$$

$$\neg prev.r \;\to\; b1.r\!\downarrow \qquad\qquad \neg b1.a \;\to\; prev.a\!\downarrow$$

This construct, combined with the latch elimination allows us to write essentially 'latch-less' programs that are purely control circuits in CHP. For example, consider a buffer that tightly synchronizes the actions on the two channels: $*[L?x \bullet R!x]$.[2] In order to obtain a circuit that achieves this behavior via synthesis, one can write: $*[[\overline{L} \to x := L]; R!x; L?]$, or better yet: $*[[\overline{L} \to R!L]; L?]$ (which makes use of channel expressions). These two programs are semantically equivalent to the first one as in all three cases, the environment must

[2] In Martin's CHP, the bullet operator is the tightest form of synchronization, ensuring that the number of completed actions on the two channels are always the same.

perform $L! \parallel R?$ in order for the system to not deadlock.

The circuit implementation for these programs results in zero latches/storage elements and consists purely of control C-elements. This structure is even more effective when creating latch-less (and slack-less) copies, splits etc., especially when the bitwidth of the data variable grows and one wants to avoid the overhead of latching it (or avoid introducing slack). For example, a latch-less 3-way copy process can be written as: $*[[\overline{L} \rightarrow R_0!L, R_1!L, R_2!L]; L?]$. In the next section, we see how this same technique can be applied to implementing multiple channel accesses in a correct and more efficient way.

VI. Slack-less Multiple Channel Access

Maelstrom uses a state-tracking mechanism to rewrite CHP that has syntactically multiple accesses of the same channel. However, this has the side-effect of introducing one unit of slack on the channel due to the handler process emplaced in between the original process and the environment. As a result, the rewrite requires the system to be slack elastic. In order to see the problem, consider the following CHP with a multiple send on the channel X:

$$*[X!0;\, Y!1;\, X!2] \parallel ([|\overline{X} \longrightarrow good \,[]\, \overline{Y} \longrightarrow bad\,|]...)$$

Here, the bad program can never get executed as the non-deterministic selection will always pick the only branch whose guard is true, i.e. the first one. Under the rewrite rule used in Maelstrom, this would get rewritten into:

$$\begin{aligned} &*[X_0!0;\, Y!1;\, X_1!2] \parallel ([|\overline{X} \longrightarrow good \,[]\, \overline{Y} \longrightarrow bad\,|]...) \\ &\parallel \quad s := 0;\;\; *[[s = 0 \longrightarrow X_0?z \,[]\, s = 1 \longrightarrow X_1?z];\\ &\qquad\qquad X!z,\;\; s := 1 - s] \end{aligned}$$

In this program, bad has a possibility of getting executed, which is semantically different from the originally specified program; this problem arises from the unintended addition of slack. We amend this by changing the structure of the handler process, in order to forward the value from the channel first to environment and delaying the completion of the handshake. This tightly couples the channel actions on the original (X) and alias (X_0, X_1) channels, resulting in zero slack addition. For the example above, the correct handler process would be:

$$\begin{aligned} &*[X_0!0;\, Y!1;\, X_1!2] \parallel ([|\overline{X} \longrightarrow good \,[]\, \overline{Y} \longrightarrow bad\,|]...) \\ &\parallel \quad s := 0;\;\; *[[s = 0 \longrightarrow [\overline{X}_0 \longrightarrow z := X_0] \\ &\qquad []\, s = 1 \longrightarrow [\overline{X}_1 \longrightarrow z := X_1]]; \\ &\qquad X!z;\, [s = 0 \longrightarrow X_0? \,[]\, s = 1 \longrightarrow X_1?];\, s := 1 - s] \end{aligned}$$

The assignment of a variable to a channel $(z := X_1)$ stores the value pending on the channel (and we know there is one pending for sure since the probe must have evaluated to true) into the variable, without performing a handshake on it. Later, after forwarding the value on X, we complete the handshake on the appropriate alias channel. The reader can verify that with this handler process, bad can never execute, thereby preserving the semantics of the original program. This has the added benefit, due to latch elimination, of the handler process never latching the data on the channel and purely forwarding it to the original process.

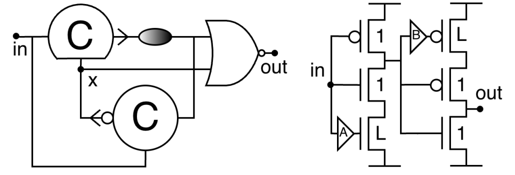

Fig. 3: Asymmetric delay lines that propagate a rising edge slower than a falling edge. Left: Double reuse of a symmetric delay to emulate an asymmetric delay. Right: Recursively constructed delay line by using asymmetrically sized inverters.

The same problem arises in the case of a multiple receive as well, and the solution is essentially the same. As a simple example, the correct rewrite for $*[X?x;\, X?y]$ is:

$$\begin{aligned} &*[X_0?x;\, X_1?y] \quad \parallel \quad s := 0;\;\; *[[\overline{X} \longrightarrow z := X]; \\ &\quad [s = 0 \longrightarrow X_0!z \,[]\, s = 1 \longrightarrow X_1!z];\, X?,\, s := 1 - s] \end{aligned}$$

Although prior work [8] has tackled the multiple channel access problem in slack-elastic systems, this method is applicable in the general case where elasticity cannot be assumed.

VII. Asymmetric delay lines

Maelstrom presented circuit implementations of CHP that support 2-phase as well as 4-phase operation. However, in order to allow the 4-phase circuits to achieve their maximum possible performance, asymmetric delay lines i.e. ones which propagate a rising edge (1) and falling edge (0) at different rates, are required. These allow the first half-phase of the execution to proceed at the speed required by the datapath and the reset half-phase to proceed as fast as possible. Hence, we are interested in delay lines that propagate a rising edge much slower than a falling edge. There have been several proposed solutions for this [9], [10]. Fig. 3 shows two topologies that allow a large rising edge delay and the shortest possible falling edge delay, that of 1 NMOS and 1 PMOS switching. In the topology on the left, production rule for the asymmetric C-element is: $in \wedge x \rightarrow y\!\uparrow, \neg in \rightarrow y\!\downarrow$, which can trivially be made CMOS-implementable. Initially, x is high, out is low, and the rising edge on in caues the gate to fire. After going through the delay line once, x is pulled low by the bottom C-element firing. After another trip through the delay line, the output of the NOR-gate goes high, effectively using the delay line twice to generate a large delay. When in goes low, out goes low after merely two gate firings, resulting in an extremely short delay. This strategy is quite area-efficient for large delays as the internal delay line only needs to be half as long as if a simple delay line was used.

The topology on the right is a recursively constructed asymmetric delay line, which internally uses the same topology as a sub-circuit (the elements depicted as buffers, A and B, are themselves the same delay line). In order to understand how this works, consider the base case, where the buffers are just wires, and the delay line is asymmetric due to the sizing, and propagates a rising edge slower than a falling edge. There also exists the dual, which propagates a falling edge

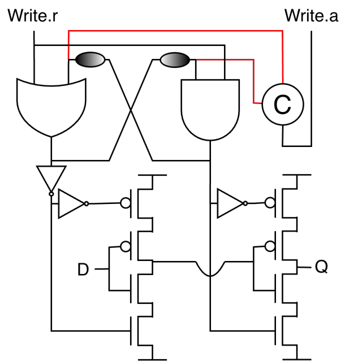

Fig. 4: Edge-triggered data capture element which consists of two latches connected in series, with a controller to open and close them in the correct sequence.

slower than a rising edge. Now, to construct the next recursive one, we simply use these asymmetric delay lines as A and B respectively. The process can then be performed repeatedly to get delay lines of increasing one-sided delay while maintaining the delay of the opposing edge to just 2 transitions. The intermediate node would also require a staticizer as there is a period of time where it is undriven.

We incorporate these topologies in the circuit library alongside existing ones and pick between them automatically during circuit construction, as the range of attainable delays is different for each topology.

VIII. Edge-Triggered Datapath

Maelstrom's implementation of the datapath relied on pulsed latches. While efficient in terms of area, pulse generators are sensitive to loading effects and it is helpful to have edge-triggered elements as an option, as they void the need for a pulse generator. Fig. 4 shows an edge-triggered storage element that effectively looks like a D flip-flop, with the added benefit of eliminating the hold-time constraint. In a conventional D flip-flop, the hold-time constraints arises from the fact that there is a short window of time where both latches are transparent. This creates a transient combinational loop, which enforces the requirement that the minimum delay around the loop must be larger than the time window for which these latches are open. The controller eliminates this by changing the sequence of transitions such that the two latches are never open simultaneously. The hold time requirement is replaced by the internal delay that ensures that the two latches are both closed for a small window of time. On assertion of write request, the first latch closes, followed by the second opening, which captures the data on the input. This is identical to an ordinary D flip-flop. However, on de-assertion of the write request, the sequence is flipped, with the second latch closing before the first opens, as can be seen by the sequence of transitions on the OR- and AND-gates. Hence, there is no instant of time with both latches open. Note that the transitions in the controller are strictly sequenced and this scheme is is actually insensitive to the delay of the OR- and AND-gates.

IX. Quantitative Results

The improvements to the circuits and synthesis method in general were incorporated into the open-source Maelstrom synthesis engine, implemented in the ACT framework [11]. The tool was then used to synthesize complex CHP programs into asynchronous circuits in a 65nm technology node. For combinational logic that implements datapath arithmetic, we use the Yosys [12] synthesis suite, which internally uses the ABC logic synthesis system [13]. Since we use a single-rail bundled datapath and match the delay in the control path using delay lines, the output circuits from ABC are allowed to glitch, as long as they settle within the time constraint imposed by the delay line.

Table I contains the key metrics for the synthesized circuits. We report layout area from a placed power-routed and global-routed design of 100 instances of each circuit. The reported area number is a per-instance average area, across the 100 instances. We also report throughput and energy-per-token metrics from pre-layout SPICE simulations. On average, the circuits produced by Maelstrom2 have an improvement of 9% in area, 16% in token time and 23% in energy consumption, when compared to Maelstrom.

The first number in each cell corresponds to the 4-phase circuits, where the datapath is only activated on one phase of the handshake, with the other used exclusively for reset. These circuits can exploit the new asymmetric delay lines. The numbers in parentheses are for the 2-phase circuits, where the datapath is activated on both phases of the handshakes that propagate through the ring, instead of just on the positive half-phase. These circuits must use symmetric delay lines as they need to match the datapath delay on both phases. All ratios are relative to the Maelstrom2 2-phase value for a given test case, which is taken to be 1 and thus omitted in the column.

The circuits for extremely simple CHP programs, such as those that correspond to dataflow elements are essentially unchanged, as these were already on par with hand-optimized circuits. For larger programs, the effect of the improved control elements and reduction of latches is apparent as the synthesized circuit is smaller, faster and uses less energy per computation.

A. *Comparing with Synchronous State Machines*

In order to compare our synthesis with synchronous synthesis, we write Verilog programs that are behaviorally equivalent to the CHP programs, with channels replaced with standard ready-valid interfaces implement flow control. In order to synthesize the Verilog, we once again use Yosys to maintain fairness in terms of combinational logic optimization. The synthesized circuits are timed using Cadence Tempus to determine the critical path and thus the maximum clock frequency for the circuit. We use this clock frequency in SPICE simulations to extract energy and throughput metrics.

We see that on most programs, the result from Maelstrom2 are comparable with equivalent synchronous state machines. The key to improving the performance of the asynchronous circuits is latch elimination, which obviates several unnecessary

delays and latchings. Further, for some programs, where there are fast and slow paths, it is possible for the asynchronous circuit to have a smaller average cycle time than the critical period. As an example, consider the skipping multiplier (which can skip the multiplication if either of the inputs is zero). This corresponds to the following CHP:

$$*[I_1?x, I_2?y; [x = 0 \vee y = 0 \longrightarrow z := 0$$
$$[] else \longrightarrow z := x*y]; \;\; O!z]$$

If the input pairs contain a high density of zeroes, then the average cycle time of the synthesized circuit for this CHP can be smaller than the delay of the multiplier. However, the equivalent Verilog state machine cannot dynamically adjust its speed based on the data and is thus constrained by the worst-case (critical) delay, which is the multiplier delay. For cases such as this, the asynchronous circuit outperforms the synchronous one. The same effect is also observed in the L2-Norm program, which is just a sequence of square-and-accumulate operations. Here too, the asynchronous circuit outperforms in the presence of sparsity.

The final missing pieces that are required for the asynchronous circuits to close the remaining gap in performance are physical optimizations, such as gate-sizing and buffer insertion. In order to substantiate this claim, we perform gate-sizing manually for one of the test programs (cross-correlation) and report metrics for the sized circuit as well. As seen in Table I, this results in the cycle time of the asynchronous circuit being virtually the same as the synchronous one, while still expending lesser energy per token. The gate-sizing and buffer insertion steps can be automated at the netlist level and we plan to add this to synthesis flow in the near future.

Finally, notice that the difference in token time between the synchronous and asynchronous circuits for the Fibonacci and GCD test cases is significantly larger than the rest. This is due to the fact that these are iterative (i.e. multi-cycle) tests. Hence, even though the asynchronous circuit is only slower than the synchronous one by 200-400ps (without gate sizing), this lag adds up over the multiple cycles that it takes for the circuit to produce a single output data token. Thus, the apparent lag gets multiplied by the number of CHP loop iterations taken to compute each token. Hence, in this case, the benefit of gate-sizing would also be proportional to the number of cycles, as this would improve the circuit cycle time by a constant factor, not the time between output tokens.

X. Discussion

The improved logic synthesis technique proposed in this work supports 2-phase as well as 4-phase style circuits. However, it is not the case that one uniformly dominates the other in terms of performance. This is dependent on the exact nature of the CHP program that is being synthesized. The 4-phase circuit has more complex handshake (controller) elements that allow for parallel reset, while the 2-phase can accommodate a more lightweight controller as there is no explicit reset phase at all. On the other hand, the circuits that implement selections in 2-phase are more complex than in the 4-phase case as they need to, in some sense, 'remember' each branch that was taken and route the control pulse propagating through the selection on a particular iteration. The 4-phase circuit has the luxury of an entire dedicated phase just for reset and thus requires a simpler circuit for implementing selections. In order to see this more clearly, consider this simple CHP for a merge:

$$*[C?c; [c = 0 \longrightarrow X?x [] c = 1 \longrightarrow Y?x]; Z!x]$$

Suppose that the sequence of values on the channel C is $0, 1, 0$. Now, the 4-phase circuit simply activates the $c = 0$ branch, resets, activates the $c = 1$ branch, resets, activates the $c = 0$ branch, and resets. Hence, we know that the positive-, or data-half-phase is always a rising edge that propagates through the circuit. In the 2-phase circuit for the same program, the circuit must send an active-high control wave through the $c = 0$ branch, do the same to the $c = 1$ branch without affecting the $c = 1$ branch, then send an active-low control wave through the $c = 0$ branch without affecting the $c = 1$ branch. At the end, the circuit is in a different state than when we started, with the control elements in the $c = 1$ branch still holding a high. This orchestration is implemented with XOR-gates and one-bit latches and adds overhead.

In general, the cost of the 4-phase vs. 2-phase circuit depends on the balance between the number of handshake elements (which is proportional to the number of channel actions) and the number of selections, as one is cheaper in one family and vice versa.

XI. Conclusion

In this work, we presented several improvements and novel additions to the state-of-the-art synthesis system for asynchronous circuits from high-level CHP descriptions. The changes result in higher quality of synthesized circuits across all relevant metrics - area, delay and energy, as verified by SPICE simulations. We also compare with synchronous state machines and show that the performance of the asynchronous circuits matches that of the synthesized synchronous logic.

Appendix A

Communicating Hardware Processes (CHP) is a hardware description language used to describe clockless circuits that is derived from Hoare's Communicating Sequential Processes (CSP) [14]. A full description of CHP and its semantics can be found in [4]. Below is an informal description of a subset of that notation that we use, listed in descending precedence, replicated from [15]. For a complete discussion of the interaction between the handshake expansions of channel actions and the composition operators, see [16].

A **Channel** X consists of a **request** X.r and either an **acknowledge** X.a or **enable** X.e. The acknowledge and enable serve the same purpose, but have inverted sense. With these signals, a channel implements a network protocol to transmit data from one process to another.

- **Skip**: *skip* does nothing; continues to the next command.
- **Assignment**: $x := e$ sets the variable x to the value e, where e is an expression.

TABLE I: Results of synthesis using Maelstrom2 and Maelstrom for CHP, and Yosys for Verilog. Area is from a placed, power- and global-routed design in a commercial 65nm technology. The token time and energy numbers are extracted from SPICE simulations. All ratios are relative to the Maelstrom2 2-phase value for a given test case. MaelstromL refers to Maelstrom with only the latch optimization added.

Program	Method	Datapath Registers	Area		Token Time		Energy per Token	
			μm^2	Ratio	ns	Ratio	pJ	Ratio
Buffer	Maelstrom (2-phase)	16 (16)	685 (648)	1.05 (1.00)	0.581 (0.149)	3.89 (1.00)	0.15 (0.07)	2.14 (1.00)
	MaelstromL (2-phase)	16 (16)	685 (648)	1.05 (1.00)	0.581 (0.149)	3.89 (1.00)	0.15 (0.07)	2.14 (1.00)
	Maelstrom2 (2-phase)	16 (16)	685 (648)	1.05 (-)	0.581 (0.149)	3.89 (-)	0.15 (0.07)	2.14 (-)
	Synchronous	16	716	1.10	0.540	3.62	0.10	1.42
GCD	Maelstrom (2-phase)	168 (168)	10130 (13686)	0.84 (1.14)	43.97 (35.95)	1.46 (1.19)	21.00 (26.40)	1.23 (1.55)
	MaelstromL (2-phase)	102 (102)	8912 (13154)	0.744 (1.10)	43.06 (31.54)	1.43 (1.05)	19.57 (21.43)	1.15 (1.26)
	Maelstrom2 (2-phase)	102 (102)	8598 (11970)	0.718 (-)	42.34 (30.02)	1.41 (-)	18.95 (16.95)	1.11 (-)
	Synchronous	53	10721	0.89	25.10	0.83	18.11	1.06
Fibonacci	Maelstrom (2-phase)	162 (162)	5011 (6225)	0.98 (1.23)	54.74 (42.05)	1.98 (1.52)	31.28 (22.61)	2.78 (2.01)
	MaelstromL (2-phase)	90 (90)	4217 (5384)	0.83 (1.06)	52.76 (29.12)	1.91 (1.05)	16.13 (13.84)	1.43 (1.23)
	Maelstrom2 (2-phase)	90 (90)	4004 (5068)	0.79 (-)	51.86 (27.51)	1.88 (-)	12.53 (11.25)	1.11 (-)
	Synchronous	70	4555	0.89	21.23	0.77	12.45	1.10
Matrix Multiply	Maelstrom (2-phase)	600 (600)	59731 (65126)	1.01 (1.10)	3.18 (1.97)	1.65 (1.03)	6.80 (4.92)	1.42 (1.03)
	MaelstromL (2-phase)	144 (144)	54981 (61893)	0.92 (1.04)	3.07 (1.93)	1.59 (1.01)	6.13 (4.86)	1.28 (1.02)
	Maelstrom2 (2-phase)	144 (144)	53038 (59143)	0.89 (-)	3.04 (1.92)	1.58 (-)	5.92 (4.76)	1.24 (-)
	Synchronous	69	52531	0.88	1.90	0.99	5.43	1.14
Cross-Correlation	Maelstrom (2-phase)	248 (248)	19571 (19487)	1.12 (1.12)	4.62 (2.42)	2.13 (1.12)	3.78 (1.71)	2.86 (1.29)
	MaelstromL (2-phase)	160 (160)	17923 (18076)	1.03 (1.03)	4.23 (2.25)	1.95 (1.04)	2.97 (1.41)	2.25 (1.07)
	Maelstrom2 (2-phase)	160 (160)	17475 (17395)	1.00 (-)	4.14 (2.16)	1.91 (-)	2.62 (1.32)	1.98 (-)
	Maelstrom2 (2-phase) (sized)	160 (160)	17832 (17523)	1.02 (1.01)	3.72 (1.79)	1.72 (0.82)	2.68 (1.39)	2.03 (1.05)
	Synchronous	131	22740	1.30	1.75	0.81	1.90	1.43
SerDes	Maelstrom (2-phase)	149 (149)	4555 (7205)	0.75 (1.19)	57.30 (29.70)	1.99 (1.03)	40.00 (37.67)	1.14 (1.07)
	MaelstromL (2-phase)	105 (105)	4031 (6649)	0.72 (1.09)	54.12 (29.13)	1.89 (1.02)	36.41 (36.23)	1.04 (1.04)
	Maelstrom2 (2-phase)	105 (105)	3733 (6050)	0.62 (-)	53.61 (28.67)	1.86 (-)	34.20 (35.00)	0.97 (-)
	Synchronous	62	3260	0.53	28.32	0.98	30.10	0.86
Shared Function Block	Maelstrom (2-phase)	92 (92)	7794 (10586)	0.77 (1.05)	4.61 (3.04)	1.54 (1.02)	1.01 (1.71)	0.65 (1.10)
	MaelstromL (2-phase)	68 (68)	7532 (10252)	0.75 (1.02)	4.51 (3.00)	1.51 (1.01)	0.93 (1.61)	0.60 (1.04)
	Maelstrom2 (2-phase)	68 (68)	7257 (9999)	0.72 (-)	4.46 (2.98)	1.49 (-)	0.84 (1.55)	0.54 (-)
	Synchronous	23	9898	0.99	2.65	0.89	5.12	3.30
Skipping Multiplier	Maelstrom (2-phase)	64 (64)	6773 (7378)	0.98 (1.07)	1.37 (0.95)	1.95 (1.35)	0.99 (0.83)	1.80 (1.51)
	MaelstromL (2-phase)	32 (32)	6489 (7074)	0.95 (1.03)	1.28 (0.77)	1.82 (1.10)	0.82 (0.62)	1.49 (1.13)
	Maelstrom2 (2-phase)	32 (32)	6256 (6856)	0.91 (-)	1.23 (0.70)	1.75 (-)	0.73 (0.55)	1.32 (-)
	Synchronous	16	6789	0.99	0.85	1.21	1.01	1.83
L2 Norm	Maelstrom (2-phase)	66 (66)	7257 (7938)	0.93 (1.02)	2.18 (1.20)	2.50 (1.37)	0.36 (0.28)	1.56 (1.21)
	MaelstromL (2-phase)	50 (50)	7012 (7892)	0.89 (1.01)	1.85 (0.96)	2.12 (1.10)	0.30 (0.25)	1.30 (1.09)
	Maelstrom2 (2-phase)	50 (50)	6845 (7812)	0.87 (-)	1.63 (0.87)	1.87 (-)	0.27 (0.23)	1.17 (-)
	Synchronous	23	5958	0.76	1.16	1.33	0.18	0.78
Geometric Mean	Maelstrom 2-phase vs. Maelstrom2 2-phase			1.09		1.16		1.23
	MaelstromL 2-phase vs. Maelstrom2 2-phase			1.04		1.05		1.09
	Synchronous vs. Maelstrom2 2-phase			0.90		1.11		1.30

- **Send**: $C!e$ sends the value of e over the channel C.
- **Receive**: $C?x$ receives a value over channel C and stores it in the variable x.
- **Probe**: $\overline{X}$ is used to determine if the channel is ready for a receive action, returning the value waiting on the request $X.r$ without executing the receive. For dataless channels, the syntax is simplified to X.
- **Sequential Composition**: $S; T$ executes the programs S followed by T.
- **Parallel Composition**: $S || T$ (or) S, T executes the programs S and T in any order.
- **Deterministic Selection**: $[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ where G_i, called a guard, is a dataless expression and S_i is a program. The selection waits until one of the guards, G_i, evaluates to $true$, then executes the corresponding program, S_i. The guards must be stable and mutually exclusive. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, which corresponds to waiting for G to become true.
- **Non-Deterministic Selection**: $[| G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n |]$ is the same as Deterministic Selection except that the guards do not have to be stable or mutually exclusive. If two or more evaluate to $true$ simultaneously, then one is picked arbitrarily (not necessarily random).
- **Loop**: $*[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ is similar to the selection statements. However, once a branch is completed, the guards are re-evaluated and another branch is chosen. The process is repeated until no guard evaluates to $true$, in which case the loop terminates. $*[S]$ is shorthand for $[true \rightarrow S]$.
- **Do-Loop**: $*[S_1 \leftarrow G_1]$ is similar to the loop, but can have only one branch. The branch is executed before evaluating the guard, similar to a do-while loop in software programming languages.

References

[1] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," *IRE Transactions on Circuit Theory*, vol. 6, no. 1, pp. 12–25, 1959.

[2] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," in *IEEE International Conference on Computer-Aided Design*, pp. 318–319, IEEE Computer Society, 1991.

[3] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 102–109, 2017.

[4] A. J. Martin, "Synthesis of asynchronous vlsi circuits," Tech. Rep. CS-TR-93-28, California Institute of Technology, 1991.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on information and Systems*, vol. 80, no. 3, pp. 315–325, 1997.

[6] K. Srinivasan and R. Manohar, “Maelstrom: A logic synthesis technique for asynchronous circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[7] M. Nyström, R. Manohar, and A. J. Martin, “Method and apparatus for a failure-free synchronizer,” Feb. 10 2004. US Patent 6,690,203.

[8] X. Wen, R. Li, and R. Manohar, “Translating general slack elastic programs into dataflow circuits,” in *2025 29th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 80–88, IEEE, 2025.

[9] S. Tugsinavisut, Y. Hong, D. Kim, K. Kim, and P. Beerel, “Efficient asynchronous bundled-data pipelines for dct matrix-vector multiplication,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 448–461, 2005.

[10] R. Tadros, N. Dasari, and P. Beerel, “Ultra-low power pass-transistor-logic-based delay line design for sub-threshold applications,” *Electronics Letters*, vol. 52, no. 23, pp. 1910–1912, 2016.

[11] S. Ataei, W. Hua, Y. Yang, R. Manohar, Y.-S. Lu, J. He, S. Maleki, and K. Pingali, “An open-source eda flow for asynchronous logic,” *IEEE Design & Test*, vol. 38, no. 2, pp. 27–37, 2021.

[12] C. Wolf, J. Glaser, and J. Kepler, “Yosys-a free verilog synthesis suite,” in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[13] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in *Computer Aided Verification: 22nd International Conference*, (Germany), pp. 24–40, Springer, Berlin, 2010.

[14] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[15] N. Bingham and R. Manohar, “A systematic approach for arbitration expressions,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4960–4969, 2020.

[16] R. Manohar, “An analysis of reshuffled handshaking expansions,” in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pp. 96–105, IEEE, 2001.