Heterogeneous Latch-based Asynchronous Pipelines

Girish Venkataramani
Carnegie Mellon University
Pittsburgh, PA 15213
girish@cs.cmu.edu

Tiberiu Chelcea
Carnegie Mellon University
Pittsburgh, PA 15213
tibi@cs.cmu.edu

Seth C. Goldstein
Carnegie Mellon University
Pittsburgh, PA 15213
seth@cs.cmu.edu

Abstract

We present a technique to automatically synthesize heterogeneous asynchronous pipelines by combining two different latching styles: normally open D-latches [19] for high performance and self-resetting D-latches [5] for low power. The former is fast but results in high power consumption due to data glitches that leak through the latch when it is open. The latter is normally closed and is opened just before data stabilizes. Thus, it is more power-efficient but slower than normally open D-latches.

We propose a module selection optimization that assigns each pipeline stage to one of these two latching styles. This is performed by an automated algorithm that uses two types of heuristics: (1) it uses the Global Critical Path (GCP) [26], to assign D-latches to stages that are sequentially critical, and (2) it estimates potential datapath glitching to make SR-latch assignment decisions. The algorithm has quadratic-time complexity and experiments that apply the algorithm on several media processing kernels indicate that, on average, the heterogeneous pipelining algorithm achieves higher performance and is more energy efficient than either the homogeneous D-latch or SR-latch pipeline styles.

1. Introduction

As technology shrinks and problems of clock distribution and timing closure become increasingly difficult, asynchronous circuits become more attractive, since they offer a modular design paradigm. The importance of finding energy and performance efficient pipeline structures and seamlessly integrating them with standard toolflows is reflected in a wide body of literature on the subject. One integral aspect of these pipeline structures is the choice of the pipeline data storage unit, which can dramatically influence both performance and energy efficiency.

In this paper, we examine and optimize the performance and power properties of commonly used pipeline storage units, which we refer to as pipeline latches (or just latches). Particularly, we identify two interesting pipeline latching styles, D-latches [19] for their high-performance and Self-Resetting Latches (SR-Latches) [5] for their energy efficiency, and propose an algorithm that automatically mixes and matches these styles in the same asynchronous system. Such an algorithm leverages on the modularity of asynchronous pipeline designs to deliver heterogeneous pipelines with superior performance and energy efficiency properties.

Latch-based pipeline styles are very common in asynchronous pipelines [19, 9, 7, 21] due to their high-performance properties. Latches are normally open, and thus eliminate the control overheads associated with opening them. However, precisely because of this, data glitches are allowed to pass through the latch to downstream stages causing useless switching activity in the datapath. While these data glitches do not cause correctness concerns, they result in an increase in datapath power consumption.

Recently, Chelcea et. al. [5] proposed Self-Resetting latches (SR-latches) to solve the power consumption problem. The SR-latches are normally closed; a dedicated latch controller opens them just before the output data stabilizes, allowing the final result to pass downstream. This behavior prevents a majority of the data glitches from passing through thereby reducing datapath power consumption compared to the D-latch implementations [5]. However, they have two major drawbacks: (a) they are slower compared to a D-latch implementation and (b) they require a dedicated SR-latch controller [5] which increases area and control-path power.

In this paper, we present an optimization to manage this trade-off between performance and power consumption. It is described as a module selection problem that builds a heterogeneous latch-based pipeline by assigning each pipeline stage in the design to use either a D-latch or an SR-latch style, with the objective of sustaining both the energy efficiency benefits of a homogeneous SR-latch pipeline and the performance benefits of a homogeneous D-latch pipeline. The algorithm is founded on a set of simple and intuitive heuristics that assess the power and performance costs of assigning an SR-latch or a D-latch to a given pipeline stage.

To assess the cost of performance, the algorithm leverages knowledge of the Global Critical Path (GCP) [26], which was shown to represent the sequential critical path of execution and thus principal bottleneck in the system. The pipeline stages that fall on the GCP are globally critical in that their latency of execution directly contributes to the system-wide cycle time [3, 6]. Derived from the GCP is the notion of *global slack* [24], which describes how much a given stage can be slowed down without affecting the GCP. If a stage falls on the GCP, its global slack

1522-8681/08 $25.00 © 2008 IEEE
DOI 10.1109/ASYNC.2008.21

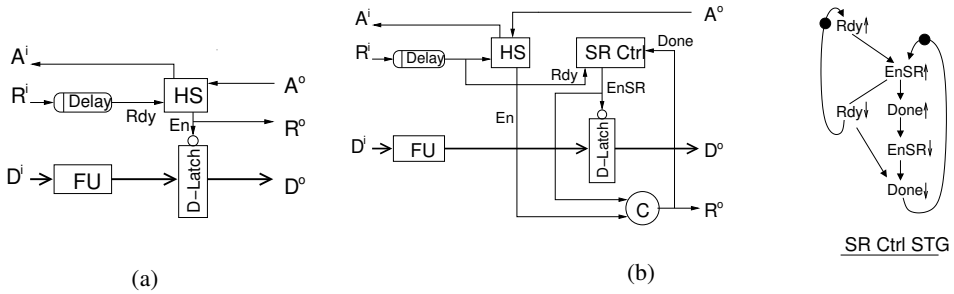

Figure 1. The two pipeline implementation styles: (a) D-latch based pipeline style for high-performance and (b) SR-latch style for better energy efficiency.

value is zero, otherwise it is the minimum available slack along any topological path leading to the GCP. Thus, global slack is essentially the timing budget representing how much a pipeline stage can be slowed down without degrading the system-wide cycle time. The algorithm assigns an SR-latch to a pipeline stage which has sufficient global slack.

The cost of power is most accurately assessed by a formal model that estimates the possibility of data glitches in a given datapath component. However, it is difficult to model glitching in a dynamic asynchronous circuit. The amount of glitches is closely tied to the input data vectors and the topology and speed of a given datapath component. Instead of modeling this power consumption, we estimate the potential for datapath glitches if a given stage were assigned a D-latch. For example, if the consumer of a stage contains a complex functional unit (e.g., a multiplier), then the potential for datapath glitches is higher. This will prompt the algorithm to assign an SR-latch to the current stage. Similarly, we have devised a set of heuristics to roughly assess the potential for data glitches in a given circuit sub-system.

Based on these heuristics, we formulate an algorithm that first analyzes the GCP and global slack properties of a pipeline in which all stages are uniformly assigned D-latches. Then, we iteratively select candidate stages that exhibit benefits when converted to an SR-latch implementation. Between iterations, the timing properties like GCP and global slack are updated. The result is a quadratic-time algorithm that synthesizes a heterogeneous, latch-based asynchronous pipeline circuit. In this paper, we focus the algorithm on optimizing four-phase bundled data asynchronous circuits, but we believe that the concepts presented are extensible to other methodologies as well.

We have incorporated this optimization within CASH [2, 25], a hardware compiler that synthesizes asynchronous four-phase bundled data handshake circuits from C programs. Experimental results from synthesizing several media processing kernels from the Mediabench [12] suite indicate that the proposed algorithm achieves the best energy-delay and energy-delay-area properties when compared to homogeneous D-latch and homogeneous SR-latch implementations. Moreover, the performance of the heterogeneous pipelines are at par with or superior to that of homogeneous D-latch pipelines, which are known to be well suited for high-performance.

The rest of the paper is organized as follows. Section 2 examines previous work on pipeline latching styles. Section 3 motivates the problem and defines the optimization. Section 4 describes the set of heuristics used to assess performance and power costs and Section 5 describes the overall algorithm. Section 6 presents experimental results and we conclude in Section 7.

2. Related Work

Since the seminal work of Sutherland [21], a primary focus of asynchronous circuits research has been on pipeline style implementations. These pipeline implementations can typically be classified as fine-grained or coarse-grained structures. The former operates at the bit level while the latter operates at the level of word-length operations or "functional units".

Fine-grained pipeline structures were introduced by Williams [30], and improved on by many others [18, 15, 17, 13, 32, 16, 22]. These pipeline templates are usually implemented using dynamic logic, which provides intrinsic storage capability, but are not amenable to automatic synthesis with commercial standard-cell CAD tools.

Coarse-grained pipeline structures, which is the focus of this work, can be implemented with standard-cell gates [21, 19, 9, 7]. Several latch and handshake protocol implementations have been proposed in the past. The Sutherland [21] micropipelines communicate with two-phase handshaking and use pass-capture logic for the latches. Singh and Nowick [19] proposed Mousetrap as a latch-based implementation for two-phase handshake protocols. These latches are similar to the D-latches we consider in this paper in that they are normally open during the passive phase of the handshake and are closed during the active phase. Furber and Day [9] have proposed several different styles of latch implementations that could be used in conjunction with strongly coupled and fully decoupled handshake protocols. More recently, SR-latches [5] were proposed as an energy-efficient version of normally transparent latches like Mousetrap [19].

Most CAD tools designed for synthesizing asynchronous systems from high-level languages [8, 23, 29, 11, 1] may target different implementation styles (e.g. choice of data encoding, choice of data validity); however, these systems cannot synthesize heterogeneous systems, which mix a variety of pipeline implementation styles.

There are two approaches closer to our work. The TAST [20, 31] compiler allows the user to specify a desired data encoding for each variable, which may result in a system which contains a heterogeneous mix of data storage units in pipelines. The designer can use profiling [20] or formal methods [31] to decide how to assign encoding styles to different variables so that power and performance improvements are achieved. However, this process is not automated.

Chelcea et al [4] have introduced some peephole and resynthesis transformations for the Balsa synthesis system [8]. Some of these transformations may change the data validity of data items, which results in changes to the data storage units, thus making the system heterogeneous. However, unlike our approach, theirs is not automated and is guided by the experience of the designer.

To our knowledge, there exists no automated synthesis flow for constructing asynchronous systems with heterogeneous pipelines. Our work takes a step in this direction — we show that it is possible to mix and match the benefits from different pipeline implementation styles in order to deliver a well-rounded solution. The main contribution of this paper is an automated algorithm to build heterogeneous pipelines that is designed to reap the benefits of modularity in asynchronous circuits.

3. Problem Description

In this section, we will motivate the proposed optimization by briefly describing and comparing two latching styles. Then, we will clearly state the optimization problem and its objectives. We restrict our attention to asynchronous pipelines implementing a four-phase, bundled data handshake protocol and examine the trade-offs involved in implementing different latching styles in these pipelines.

3.1. Motivation

Fig. 1 shows the schematics of the two latch-based pipeline styles implementing a four-phase bundled data handshake protocol. The request, acknowledge and data signals of the input channel are represented by R^i, A^i and D^i respectively, while the equivalent signals in the output channel are represented by R^o, A^o and D^o respectively. The *HS* block implements the handshake protocol.

In the D-latch style in Fig. 1a, the handshake controller raises *En*, when input data has been processed. This will close the D-latch and stabilize the data at the output. The critical path to raise the R^o signal is summation of the latencies of the *Delay* and *HS* blocks.

The SR-latch style [5], illustrated in Fig. 1b, includes a dedicated latch controller, *SR Ctrl*, which determines when the D-latch is opened and closed. The State Transition Graph (STG) for the controller is also shown in Fig. 1b. The D-latch is normally closed. When *Rdy* is raised, the *SR Ctrl* opens the latch by raising *EnSR*. Once the handshake controller determines that the R^o signal can be raised, the C-element at the bottom fires, which in turn closes the latch. Thus, the latch is open for a small window of time, a little bit wider than the latency through the handshake controller, *HS*. This style introduces more timing constraints that must be met; a detailed discussion of these topics can be found in [5].

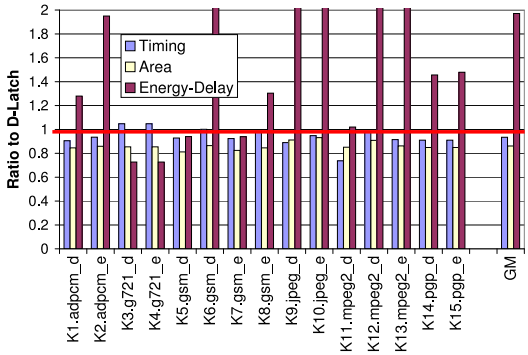

Figure 2. A comparison between homogeneous D-latch and homogeneous SR-latch styles. A value higher than 1 indicates improvement over the base case.

Notice that compared to the D-latch, the SR-latch latency to raise R^o includes the additional latency of the C-element, in comparison to the critical path of the D-latch in Fig. 1a. On the other hand, since the latch is normally closed, the potential for data glitches to leak through the latch is minimized. These differences are reflected in the performance results presented in Fig. 2. A set of benchmarks from the Mediabench suite [12] were processed by the CASH compiler [2, 25] and synthesized as both a homogeneous D-latch implementation as well as a homogeneous SR-latch implementation. The kernels are listed in Table 1. The C function in the table was the input to the compiler, which translates the program to a predicated dataflow graph representation [2]. Each node in this graph is synthesized as a handshaking pipeline stage that includes a pipeline latch. While the CASH compiler synthesizes the control-path, the datapath is synthesized by Synopsys Design Compiler. Post-layout simulations were performed in Modelsim and using the switching activity file (.saif) from simulation, power is estimated by Synopsys. The experiments were performed with the [180nm/2V] ST Microelectronics standard cell library.

The results in Fig. 2 show a comparison of end-to-end execution time, area and energy-delay for the two styles. A value greater than one implies that the SR-latch pipeline is superior to the D-latch pipeline. The results indicate that the homogeneous D-latch pipeline is superior to the homogeneous SR-latch pipeline by about 7% and 18% in terms of performance and area respectively. This is expected since the critical path delay is longer through the SR-latch style (see Fig. 1) and the additional controller (*SR Ctrl*) increases its area. However, in

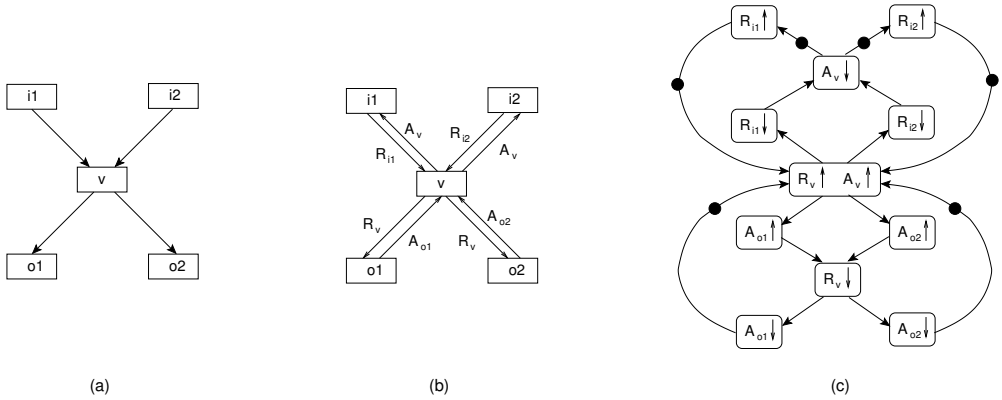

Figure 3. Understanding the pipeline's control-path. We show an (a) example dataflow sub-graph, (b) its asynchronous pipeline control-path implementation and (c) the execution model implementing the underlying protocol.

terms of energy-delay, the SR-latch implementations are close to 2x better than the D-latch ones. This is due to excessive datapath glitching activity in the D-latch which is virtually eliminated by SR-latches.

The goal of the heterogeneous latch synthesis algorithm is to mix and match the two styles in the same pipeline design. Thus, critical stages can use high-performance (and low area) D-latches while non-critical stages may use low-power SR-latches. We want to leverage the benefits of both styles in a heterogeneous pipeline.

3.2. Problem Definition

We formulate heterogeneous latch synthesis as a module selection problem. The goal of the optimization is performance-sustaining power optimization—in other words, we want to improve power and energy properties of a given design without degrading its performance as compared to a D-latch implementation. The inputs to this optimization problem are:

- A dataflow graph, $G = (V,E)$ representing the pipeline stages in the design. Each node $v \in V$ will contain a latch on its output. Let $Type(v)$ represent the latch style (D-latch or SR-latch) deployed in v. Additionally, we define op(v) to describe the ALU operation performed by stage v.
- The difference in latency between a D-latch and an SR-latch is given as Δ_{sr}, i.e., the D-latch is faster than the SR-latch by Δ_{sr} time units.

A solution to the problem is to define a mapping $Type : V \mapsto \{\text{D-Latch}, \text{SR-latch}\}$ that defines the latch implementation used in stage $v \in V$. The objective of the problem is to minimize datapath glitch power with the constraint that the system-wide cycle time [6, 26] of the solution is less than or equal to the cycle time of the D-latch solution, i.e., a solution where $Type(v) =$ D-latch for all nodes.

4. Heuristic Cost Functions

Finding the optimal solution to this problem necessitates solving two separate problems: one, finding all stages that can afford to be a bit slower for using SR-latches and, two, estimating the potential switching activity. While the former is a reasonably well understood problem, the latter is an undecidable problem since it depends on the input vectors to the circuit, which are statically unknown. Further, switching activity is extremely sensitive to the delay of gates [10]. Thus, in the presence of variation and unknown input vectors, it is very difficult to accurately estimate switching activity. Due to these reasons, we rely on heuristic cost functions to guide our algorithm toward better quality solutions.

Before presenting our algorithm, we will first describe the set of heuristic cost functions that the algorithm is based on. We use these heuristics to guide the algorithm toward higher quality solutions. We use two classes of heuristics for this problem: first, a system-level timing based cost function to guage the impact on performance and, second, a set of heuristics to guage the impact on datapath power.

4.1. Global Slack

The main constraint of the problem is that cycle time of the new solution must not be worse than cycle time of the homogeneous D-latch solution. To enforce this constraint, we use system-level timing analysis results, *slack* and the *Global Critical Path (GCP)* [26, 27].

We illustrate the concept of the GCP using Fig. 3. In (a), it shows an example sub-graph of a dataflow graph, G, representing the application. Each edge, $(u,v) \in E$, of the sub-graph becomes a handshake channel and each node, $v \in V$, becomes a pipeline stage. Fig. 3b shows the schematic for the RTL micro-architecture. Communication between pipeline stages are controlled by local handshaking based on a pre-defined protocol. Fig. 3c shows the execution model (e.g., as an STG) of the protocol implemented by the pipeline.

```
1   heterogeneous_Latch_selection(G, Slack) {
2       ∀ v ∈ V, Type(v) = D-latch;
3       Done = pre_assign(G);
4       change = true; BestSlack = 0; BestOp = undef;
5       while (change) {
6           change = false;
7           foreach (v ∈ V) {
8               if (v ∈ Done) continue;
9               if (GSlack(v) > BestSlack) {
10                  BestOp = v; BestSlack = GSlack(v);
11                  change = true;
12              }
13          }
14          if (change) {
15              Type(BestOp) = SR-latch;
16              update_slack(BestOp, −Δsr);
17              Done.add(BestOp);
18          }
19      }
20  }
```

```
21  pre_assign(G) {
22      Done = ∅;
23      foreach (v ∈ V) {
24          if (|Fo(v)| ≥ 2) {
25              Type(v) = SR-latch; Done.add(v);
26              update_slack(v, −Δsr);
27              continue;
28          }
29          u = single fanout destination of v;
30          if (BitOps(u) ≥ BitOps) {
31              Type(v) = SR-latch; Done.add(v);
32              update_slack(v, −Δsr);
33              continue;
34          }
35      }
36      return Done;
37  }
```

Figure 4. High-level overview of the heterogeneous latch selection algorithm.

This execution model can be analyzed to determine the cycle time of the system [14, 26, 3]. In steady state, this is the time difference between successive firings of a given event (i.e., signal transition) in the model. Based on this analysis, we can also determine other aspects of system-level timing. In particular, the model induces a partial firing order on the events. In Fig. 3c, for example, event $A_v\downarrow$ can fire only after both $R_{i1}\downarrow$ and $R_{i2}\downarrow$ fire. Such ordering relationships are encoded in *behaviors* [26]; every behavior, b, has a set of input events, $In(b)$, which must fire before any output event in $Out(b)$ can fire.

Based on the results of system-level timing analysis, we infer three important timing properties:

- $Slack(e_i,b)$: In steady state, this represents how early an input event, $e_i \in In(b)$, fires before all the inputs in $In(b)$ have fired. There exists a critical input, e_c, that is the last input event to fire. If each input, e_i, fires at time, T_i, then $Slack(e_i,b) = T_c - T_i$, implying that slack for the critical input is zero. If the delay to fire the behavior b is $d(b)$, then the output events in $Out(b)$, fire at time $(T_c + d(b))$.
- Global Critical Path (GCP): is the longest sequence of zero-slack events. It is a path, GCP = $\langle e_1, \cdots, e_i, e_{i+1}, \cdots, e_{last}\rangle$, such that e_{last} is the last event to fire in the execution, and for every two consecutive events, e_i and e_{i+1} on the GCP, there exists a behavior, b_{i+1}, such that $e_i \in In(b_{i+1})$ and $e_{i+1} \in Out(b_{i+1})$. Further, for every event on the GCP, $Slack(e_i,b_{i+1}) = 0$. The latency of the GCP is equivalent to the cycle time [26].
- $GSlack(e)$: represents *global slack* of event e. It specifies how long event e can be delayed without affecting the cycle time or the GCP. It is defined as the minimum cumulative slack along any output path to the GCP. It is recursively defined as follows:

$$GSlack(b) = \min_{e_o \in Out(b)} GSlack(e_o)$$

$$GSlack(e) = \begin{cases} 0, & \text{if } e \in GCP, \\ \min\limits_{\forall\ b \mid e \in In(b)} \left(Slack(e,b) + d(b) + GSlack(b)\right), & \text{otherwise.} \end{cases}$$

The key timing property we use for our optimization is global slack. From its definition, we know that if we were to slow down a given event by a value less than or equal to its global slack value, then we are guaranteed that the GCP will not change and thus the system-wide cycle time will not be affected. We use global slack as the timing budget available to a given stage for transforming its latch style to an SR-latch based implementation. Essentially, a given pipeline stage, $v \in V$ is assigned an SR-latch if the following holds:

$$GSlack(R_v\uparrow) \geq \Delta_{sr} \quad \bigwedge \quad GSlack(R_v\downarrow) \geq \Delta_{sr} \qquad (2)$$

The choice of the latching style only affects the timing of events downstream, i.e., $R\uparrow$ and $R\downarrow$ events. Thus, we ensure that there is sufficient global slack at v to tolerate a delay of Δ_{sr} at the output of both $R_v\uparrow$ and $R_v\downarrow$. As we show in Section 5, we will iteratively examine stages, $v \in V$ to determine if its type should be set to SR-latch. If $Type(v)$ is set to SR-latch, then we update slack to propagate the change throughout the execution model.

4.2. Power Costs

Now we will describe the power heuristics that help in estimating the extent of data glitches that may be saved using SR-latches:

1. Fanout: Let $Fo(v) = \{u \mid (v,u) \in E\}$. Our first observation is that the fanout, $|Fo(v)|$, of a node $v \in V$, is a factor in the

amount of data glitches that may propagate to downstream nodes if $Type(v) =$ D-latch. If $|Fo(v)|$ is large than every data glitch from v is propagated to every fanout destination of v, thus increasing the probability of glitches. Thus, we want to use SR-latches when we encounter any node with a fanout. We assign an SR-latch to every stage with a fanout greater than one:

$$|Fo(v)| \quad \geq \quad 2 \qquad (3)$$

2. *BitOps*: Next, we observe that the amount of datapath glitches in downstream stages depends on the complexity of the datapath in these stages. For example, if a datapath glitch were to leak through to a downstream stage containing a multiplier, then the leak will cause much more unnecessary switching as compared to a stage whose datapath contains just a simple logic AND gate. Thus, to estimate the potential for datapath glitches in downstream stages, we introduce the concept of $BitOps(v)$ for each stage. This value represents the number of 1-bit operations performed in the stage's datapath—this is roughly equivalent to the number of standard cell logic gates used in the datapath. The larger the value of *BitOps*, the larger is the potential for data glitches. We assign an SR-latch to a stage, $v \in V$, if any output stages from v contain a complex datapath; thus, the SR-latch prevents glitches from leaking through to these complex downstream stages. We use an upper-bound, $\overline{BitOps}$, to represent the maximum allowable bit operations to tolerate glitches due to D-latches. Specifically, an SR-latch is used in stage $v \in V$, if the number of bit-ops of a destination is larger than $\overline{BitOps}$.

$$\exists\,(v,w) \in E,\ s.t. \quad BitOps(w) \quad \geq \quad \overline{BitOps} \qquad (4)$$

5. The Algorithm

Using these heuristics, we have formulated an iterative heterogeneous latch selection algorithm, as shown in Fig. 4. The top-level function, *heterogeneous_Latch_selection()* takes as input a graph G representing the asynchronous pipeline and the performance analysis results (as described in Section 4.1) from analyzing a homogeneous D-latch based pipeline. The algorithm first performs a pre-assignment pass (Lines 21–37) that sets the types of several nodes based on the heuristics above: (a) multiple fanout nodes always contain SR-latches (Lines 24–28) and (b) if the output of the node is computationally heavy-weight, as per (4), then we also pre-assign SR-latches (Lines 29–34).

For every node, v, that is assigned an SR-latch, we must update slack (see Line 16, 26 and 32). Without having to re-compute slack through a complete timing re-analysis, it is possible to update values of slack by adjusting the effect of the timing change in the local area and propagating the effect globally [28, 24]. This technique involves an analysis of the local re-convergent paths surrounding the area of change. Once identified, the slack at the join points of these re-convergent

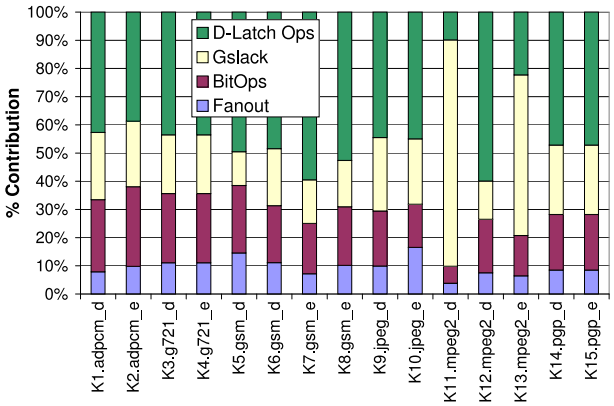

Figure 5. Contribution of each heuristic in selecting nodes that will use an SR-latch.

paths must be updated to reflect the change in timing along the re-convergent paths. Then, any change in timing at the join point is similarly propagated to the next join point and so on. This slack update algorithm is linear in the number of edges [28, 24]. In the presence of conditional behavior (choice), the update is an approximation since it focuses only on the most frequently executed paths. Still, for the set of benchmarks reported here, the algorithm was shown to be fully accurate in estimating new values of slack for about 75% of all system events, consistently across all benchmarks [28], and within an error of $\pm 10\%$ for more than 90% of all system events. Once slack is updated, the GCP and global slack can also be updated by a linear pass through the execution model.

After pre-assigning these nodes, the top-level algorithm iterates over the remaining nodes looking for candidates that can use an SR-latch if they satisfy (2). In each iteration of the `while` loop in Lines 5–19, we find the node, v, which has the largest global slack on both its $R_v\uparrow$ and $R_v\downarrow$ events. Once, we have evaluated all candidates, the best candidate is assigned to use an SR-latch and slack is updated to reflect the change (Lines 14–18). When no candidates are found in a given iteration, the algorithm stops.

5.1. Computational Complexity

The algorithm has a quadratic complexity in the size of G. This complexity is inferred as follows:

- *pre_assign()*: The initial call to this function simply iterates over all nodes; thus its complexity is $\omega_{pre} = O(|V|)$.
- `while`: For a single iteration of the `while`, we again evaluate all nodes in the graph, in the worst case; $\omega_{while} = O(|V|)$.
- iterations: In the worst-case, we may perform as many `while` loop iterations as there are nodes in G; $\omega_{iter} = O(|V|)$.
- update: the complexity of updating slack is $O(|E|)$ [28, 24]. In the worst-case, we would call update for every node in the graph. Thus, the complexity during iteration for the update is $\omega_{up} = O(|V|\,|E|)$.

Putting this together, the overall complexity of the algorithm is quadratic in the size of the dataflow graph and is derived as

follows:

$$\begin{aligned}\omega &= \omega_{pre} + \omega_{up} + \omega_{while}\,\omega_{iter} \\ &= O(|V|) + O(|V|\,|E|) + O(|V|^2) \\ &= O(|V|\,|E| + O(|V|^2))\end{aligned}$$

6. Experimental Results

We applied the heterogeneous latch selection algorithm on the Mediabench kernels [12] listed in Table 1. These kernels were synthesized using the CASH compiler [25, 2] to target a [180nm/2V] ST Microelectronics standard cell library. After physical design, the circuit's timing was extracted using Synopsys Design Compiler (SDC) and simulated using Modelsim. This gate-level simulation generates a switching activity file (.saif), which is fed back into SDC to estimate power consumption.

Table 1 reports on how many nodes in V were chosen for using an SR-latch. The runtime of the algorithm is in the last column. Of these nodes selected for SR-latches, Fig. 5 shows the reasons (or heuristics) due to which a given node was chosen for using SR-latches. The *D-Latch* segment of the bars in the graph show the fraction of stages that were not changed to SR-latches since they fall on the GCP. On average, the algorithm selects about half of the total nodes to use SR-latches. In some cases, K11.mpeg2_d and K13.mpeg2_e, for example, the number of nodes chosen for using SR-latches is much larger than the average case. This is because these kernels have certain sub-circuits that were never executed (due to conditional behavior) in these testbenches. The timing analysis technique used for extracting slack and GCP is profiling driven [26]. Thus, for these nodes, global slack is infinite.

Quantitative evaluations of the algorithm is shown in Fig. 6. These graphs show the performance comparisons between a homogeneous SR-latch implementation and the heterogeneous latch implementations relative to the homogeneous D-latch implementation, which is the base version. Thus, a value of one on the Y-axis indicates that the performance is equivalent to that of a homogeneous D-latch pipeline, while a value greater than one shows an improvement over the performance of a homogeneous D-latch pipeline. The timing graph in Fig. 6A shows that there is virtually no performance penalty to selectively assigning SR-latches to different nodes. In fact, in some cases, performance seems to improve over a purely D-latch implementation. Careful inspection of these kernels suggests no fundamental reasons for this improvement; they are, rather, side-effects of physical design since the post-layout datapath latency (and thus the size of the matched delay) in some nodes were reduced compared to the equivalent node in the homogeneous D-latch pipeline. On the other hand, the homogeneous SR-latch implementation is, on average, about 7% slower than homogeneous D-latches.

Energy delay, on the other hand, vastly improves in comparison with D-latches (see Fig. 6C). This is especially true for the large benchmarks where the probability of data glitches is much higher. On average, both the homogeneous SR-latch implementation and the hybrid implementations achieve an improvement of roughly 2x over a D-latch based implementation, with the hybrid algorithm performing marginally better.

In terms of area, a D-latch implementation is the best since SR-latches require a separate SR-latch controller as shown in Fig. 1. On average, the SR-latch and hybrid styles take up roughly 15% and 12% more area than the D-latches respectively. On the other hand, energy-delay-area improves substantially compared to the D-latch pipelines. On average, the energy-delay-area products of the hybrid and SR-latch pipelines are 1.78x and 1.69x better than the D-latch version.

In summary, we can draw the following the following conclusions from the experimental results:

- On average, across all benchmarks, each of the three heuristics, global slack, *BitOps* and fanout, pick 42%, 34% and 16% of the nodes selected to use SR-latches. This gives us an indication of their relative importance.

- It is not clear what the heuristic should be for nodes that are seldom executed in a circuit with conditional execution. While they will not generate any data glitches themselves, it is possible that glitches from upstream stages may leak into these sub-systems causing unnecessary switching. In this algorithm, we simply assign SR-latches to all these nodes. A more sophisticated pass may choose to assign SR-latches to the nodes at the input boundary of such regions and D-latches to nodes within the region.

- For a given performance bound, the hybrid version is superior to both a homogeneous D-latch and SR-latch implementation. In other words, the use of global slack enables us to accurately track performance and ensure timing-sustaining power optimization. Thus, it is able to achieve equivalent or better energy-delay properties compared to an SR-latch implementation without sacrificing the performance of D-latches. Further, since not all of the nodes are chosen to be implemented as SR-latches, the area penalty is also lesser than for a homogeneous SR-latch implementation.

- The proposed algorithm is an efficient and effective heuristic. The runtimes for the algorithm in the last column of Table 1 indicate that even the largest kernels take no more than a couple of minutes to process, thus ensuring the scalability of the algorithm. On average, the quality of circuits produced by the hybrid algorithm is superior to both a pure D-latch and a pure SR-latch implementation in terms of performance, energy-delay and energy-delay-area.

7. Conclusions

The choice of pipeline latching style plays a significant role in determining the timing, power and area properties of a given asynchronous bundled data pipeline design. D-latch pipeline stages are known to be extremely fast [19], but produce a lot

Kernel	C Function	Total Nodes	SR-Latch Ops	Runtime (secs)
K1.adpcm_d	adpcm_decoder	281	161	12
K2.adpcm_e	adpcm_coder	297	182	14
K3.g721_d	fmult+quan	163	92	3
K4.g721_e	fmult+quan	163	92	3
K5.gsm_d	LARp_to_rp	117	59	3
K6.gsm_d	Short_term_synthesis_filtering	198	102	5
K7.gsm_e	Coefficients_27_39	84	34	1
K8.gsm_e	Short_term_analysis_filtering	207	98	5
K9.jpeg_d	jpeg_idct_islow	972	539	161
K10.jpeg_e	jpeg_fdct_islow	418	230	20
K11.mpeg2_d	form_component_prediction	1132	1020	56
K12.mpeg2_d	idctcol	427	171	12
K13.mpeg2_e	dist1	1013	787	76
K14.pgp_d	mp_smul	142	75	3
K15.pgp_e	mp_smul	142	75	3

Table 1. Results of applying the hybrid ASU selection algorithm. Of the total number of nodes, "SR-Latch Ops" nodes were chosen to use an SR-latch ASU. The runtime is in the last column. These experiments were performed on Sparc machine with 4GB memory.

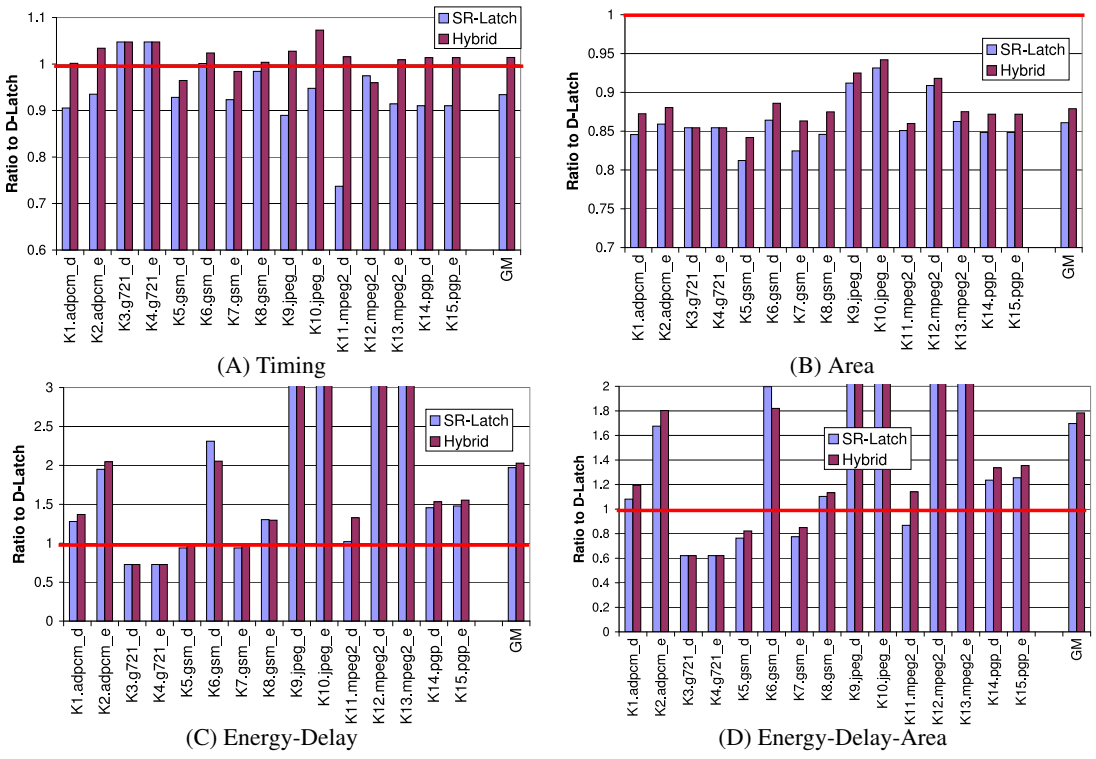

Figure 6. The improvements in overall performance, area, energy-delay and energy-delay-area due to op-chaining. Each of the graphs shows four series representing the four experiments and compares the performance, area and energy of the op-chained system to the base (initial) configuration. A bar higher than one indicates an improvement over the base run.

of data glitches leading to increase in dynamic power consumption. Self-Resetting latches (SR-latches) [5], on the other hand, reduce the number of glitches produced in latch-based pipelines but are slower and take up more area.

We have formulated a module selection problem for assigning one of these two latching styles for each of the pipeline stages in a given design. The problem is solved by an automated algorithm that uses knowledge of system-level timing in order to assign D-latches to critical pipeline stages and SR-latches to stages where the probability of glitch production is greater. The algorithm uses a set of heuristics to estimate the effects on timing and power when making latching choices in different stages. The algorithm is effective and scalable. Experimental results from applying the algorithm on media processing kernels indicate that the proposed algorithm achieves the best performance, energy-delay and energy-delay-area properties compared to equivalent homogeneous D-latch and homogeneous SR-latch pipelines. We believe that the algorithm exploits the modularity properties of asynchronous pipelines to mix and match the pipelining needs with the power budgets.

8 Acknowledgments

This research has been funded by NSF grants CCF-0702640 and CCR0205523.

References

[1] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 149–158. IEEE Computer Society Press, Apr. 2004.

[2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *ASPLOS*, pages 14–26, Boston, MA, October 2004.

[3] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[4] T. Chelcea and S. M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *DAC*, pages 405–410, New York, NY, USA, 2002. ACM Press.

[5] T. Chelcea, G. Venkataramani, and S. C. Goldstein. Self-resetting latches for asynchronous micro-pipelines. In *DAC*, pages 986–989, New York, NY, USA, 2007. ACM Press.

[6] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *TODAES*, 9(4):385–418, 2004.

[7] P. Day and J. V. Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[8] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer J.*, 45(1):12–18, 2002.

[9] S. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 4-2:247–253, 1996.

[10] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *DAC*, pages 253–259, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[11] A. Kondratyev and K. Lwin. Design of asynchronous circuits using synchronous cad tools. *IEEE Transactions on Design and Test of Computers*, pages 2–12, 2002.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, 1997.

[13] A. M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, Computer Science Department, 1995. CS-TR-95-21.

[14] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *DAC*, pages 70–76, 1994.

[15] R. O. Ozdag and P. A. Beerel. High-speed QDI asynchronous pipelines. In *ASYNC*, pages 13–22. IEEE Computer Society, 2002.

[16] R. O. Ozdag and P. A. Beerel. A channel based asynchronous low power high performance standard-cell based sequential decoder implemented with QDI templates. In *ASYNC*, pages 187–197. IEEE Computer Society, 2004.

[17] R. O. Ozdag, P. A. Beerel, M. Singh, and S. M. Nowick. High-speed non-linear asynchronous pipelines. In *DATE*, pages 1000–1007. IEEE Computer Society, 2002.

[18] M. Singh and S. M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *ASYNC*, page 198. IEEE Computer Society, 2000.

[19] M. Singh and S. M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *ICCD*, pages 9–17. IEEE Computer Society, 2001.

[20] K. Slimani, Y. Remond, G. Sicard, and M. Renaudin. Tast profiler and low energy asynchronous design methodology. In *Proceedings of the 14th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2004)*, pages 268–277, 2004.

[21] I. Sutherland. Micropipelines: Turing award lecture. *CACM*, 32 (6):720–738, June 1989.

[22] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *ASYNC*, pages 46–53. IEEE Computer Society Press, March 2001.

[23] K. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, volume 5 of *Intl. Series on Parallel Computation*. Cambridge University Press, 1993. Tangram.

[24] G. Venkataramani. *System-level Timing Analysis and Optimizations for Hardware Compilation*. PhD thesis, Carnegie Mellon University, October 2007.

[25] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IWLS*, pages 501–508, Temecula, CA, June 2004.

[26] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. Global critical path: a tool for system-level timing analysis. In *DAC*, pages 783–786, New York, NY, USA, 2007. ACM Press.

[27] G. Venkataramani, T. Chelcea, M. Budiu, and S. C. Goldstein. Modeling the global critical path in concurrent systems. Technical Report CMU-CS-06-144, Carnegie Mellon University, August 2006.

[28] G. Venkataramani and S. C. Goldstein. Leveraging protocol knowledge in slack matching. In *ICCAD*, pages 724–729, New York, NY, USA, 2006. ACM Press.

[29] W. Wang, T. K. Tan, J. Luo, Y. Fei, L. Shang, K. S. Vallerio, L. Zhong, A. Raghunathan, and N. K. Jha. A comprehensive high-level synthesis system for control-flow intensive behaviors. In *GLSVLSI*, pages 11–14. ACM Press, 2003.

[30] T. E. Williams. *Self-timed rings and their application to division*. PhD thesis, Stanford University, Stanford, CA, USA, 1991.

[31] E. Yahya and M. Renaudin. Qdi latches characteristics and asynchronous linear-pipeline performance analysis. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'06)*, pages 583–592, 2006.

[32] K. Yun, P. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1996*, pages 17–28, March 8-21 1996.