

GlitchLess: An Active Glitch Minimization Technique for FPGAs

Julien Lamoureux, Guy G. Lemieux, Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada
{julienl, lemieux, stevew}@ece.ubc.ca

ABSTRACT

This paper describes a technique that reduces dynamic power in FPGAs by reducing the number of glitches in the global routing resources. The technique involves adding programmable delay elements within the logic blocks of an FPGA to programmably align the arrival times of early-arriving signals to the inputs of the lookup tables and to filter out glitches generated by earlier circuitry. On average, the proposed technique eliminates 91% of the glitching, which reduces overall FPGA power by 18%. The added circuitry increases overall area by 5% and critical-path delay by less than 1%. Furthermore, since it is applied after routing, the proposed technique requires no modifications to the existing FPGA routing architecture or CAD flow.

Categories and Subject Descriptors

B.7.1 [Integrated Circuits]: Types and Design Styles – *Gate Arrays*

General Terms

Design.

Keywords

Field-Programmable Gate Arrays, Power Minimization.

1. INTRODUCTION

Advancements in process technologies, programmable logic architectures, and CAD tools are allowing increasingly larger and faster systems to be implemented on Field-Programmable Gate Arrays (FPGAs). These large systems, however, consume increasing amounts of power. Reducing the power of FPGA implementations is important, not only to reduce packaging costs, but to open FPGAs to many more applications.

There are two types of power dissipation in integrated circuits: static and dynamic. Static power is dissipated when current leaks between the various terminals of a transistor, while dynamic power is dissipated when individual circuit nodes toggle. Although static power is increasing relative to dynamic power for newer process technologies, dynamic power remains the dominant source of power dissipation in FPGAs. A study that examined power dissipation in a commercial 90nm FPGA found that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'07, February 18-20, Monterey, California, USA.

Copyright 2007 ACM 978-59593-600-4/07/0002...\$5.00.

dynamic power accounted for 62% of total power [1].

This paper introduces a technique that reduces dynamic power in FPGAs by actively minimizing the number of unnecessary transitions called glitches or hazards. The technique involves adding programmable delay elements within the logic blocks of an FPGA to programmably align the arrival times of early-arriving signals to the inputs of the lookup tables (LUTs) and to filter out glitches generated by earlier circuitry.

Theoretically, the proposed technique can be used to eliminate all the glitching within FPGAs and therefore significantly reduce power. In practice, however, we must trade-off the amount of glitch reduction with area and speed overhead. Since we only delay the early-arriving signals, there is no significant impact on circuit speed (other than increased parasitic capacitances). However, the programmable delay elements consume chip area, so we should expect a modest increase in the area of a configurable logic block. This tradeoff between glitch reduction (and hence power), area, and delay will be quantified in this paper. Specifically, this paper examines the following questions:

1. How should the programmable delay elements be connected within the logic blocks? The programmable delay elements could conceivably be connected to the logic block inputs, LUT inputs, logic block outputs, or combinations of these.
2. How many programmable delay elements are needed within each logic block? Intuitively, adding more programmable delay elements to the logic blocks eliminates more glitches since more signals can be aligned; however, it also increases the area overhead.
3. How flexible should the programmable delay elements be? The more flexible each delay element is, the better it will be able to align the arrival times of signals. However, there is a tradeoff between this flexibility and the area overhead of the added circuits.

This paper is organized as follows. Section 2 defines glitching and summarizes existing techniques that can be used to minimize glitching. Section 3 then examines glitching for circuits that are implemented on FPGAs. Section 4 presents the delay insertion schemes that are proposed in this paper. Section 5 then describes the experimental framework used in Section 6, which compares each scheme. Finally, Section 7 summarizes the results and presents our conclusions.

2. BACKGROUND

2.1 Terminology

There are two types of transitions that can occur on a signal. The first type is a *functional* transition, which is necessary in order to

perform a computation. A functional transition causes the value of the signal to be different at the end of the clock cycle than at the beginning of the clock cycle. In each cycle, a functional transition occurs either once or the signal remains unchanged. The second type of transition is called a *glitch* or a *hazard*, which is not necessary in order to perform a computation. These transitions can occur multiple times during a clock cycle.

2.2 Glitch Minimization

Several techniques have been proposed to minimize glitching. CAD techniques including logic decomposition [2], loop folding [3], high-level compiler optimization [4], technology mapping [5,6], and clustering [5] have been proposed to minimize switching activity. These techniques can eliminate some of the glitching, but typically incur area and delay penalties as they reorganize the structure of the circuit. Other approaches involve relocating flip-flops [7] or inserting additional flip-flops (pipelining) [8] to reduce the combinational path length. These techniques can also eliminate some of the glitching, however, significant power savings require additional flip-flops which increases the latency of the circuit. The *gate freezing* technique described in [9] eliminates glitching by suppressing transitions until the freeze gate is enabled. This technique is suitable for fixed implementations since it can be applied to selected gates with high glitch counts. However, the technique is less suitable for FPGAs since the applications implemented on FPGAs are not known until after fabrication, meaning it is difficult to determine, at fabrication time, where the extra circuitry should be added. Finally, the delay insertion technique described in [10] minimizes glitching in fixed logic implementations by aligning the input arrival times of gates using fixed delay elements. In this paper, we propose a similar technique that targets FPGAs. Aligning edges in an FPGA is considerably more complex than doing so in an ASIC, since in an FPGA, the required delay times are not known when the chip is fabricated. This means the delays must be programmable; if not managed carefully, the overhead in these programmable delay elements can overwhelm any power savings obtained by removing glitches.

3. GLITCHING IN FPGAS

This section presents statistics regarding glitching for circuits implemented on FPGAs. This section begins with a breakdown of functional vs. glitching activity to determine how much glitching is common in FPGA implementations. It then examines the width of typical glitches and determines how much power is dissipated by a single glitch. Finally, it indicates how much power could be saved if glitching could be completely eliminated. These statistics are important, not only because they help motivate our work, but also because they provide key numbers (such as typical pulse widths) that will be needed to calibrate the architectures proposed in Section 4.

3.1 Switching Activity Breakdown

Table 1 reports the switching activities for a suite benchmark circuits implemented on FPGAs. These activities are gathered using gate-level simulation of a post-place and route implementation for a set of benchmark circuits (see Section 5 for more details). Gate-level simulations provide the functional and total activity; the glitching activity is computed as the difference between these two quantities. In general, the amount glitching is greater in circuits with many levels of logic, circuits with uneven

routing delays, and circuits with exclusive-or logic. As an example, an unpipelined 16-bit array multiplier (C6288) implemented on an FPGA has five times more glitch transitions than functional transitions.

3.2 Pulse Width Distribution

In FPGAs, glitches are generated at the output of a LUT when the input signals transition at different times. The *pulse width* of these glitches depends on how uneven the input arrival times are. Intuitively, we would expect FPGA glitches to be wider than ASIC glitches, since signals are often routed using non-direct paths due to the limited connectivity of FPGA routing resources. Figure 1 plots the pulse width distribution of the C6288 benchmark circuit. The distribution was obtained using event-driven simulation and delays from VPR as described in Section 5. The graph shows that the majority of glitches have a pulse width between 0 and approximately 10 ns. Although this range varies across our benchmark circuits, we have found that the shape of the distribution is similar for every circuit.

3.3 Power Dissipation of Glitches

The parasitic resistance and capacitances of the routing resources filter out very short glitches. To measure the impact of this, HSPICE was used to build a profile of power with respect to pulse width. Figure 2 illustrated the relative power dissipated when pulses with widths ranging from 0 to 1ns are applied to an FPGA routing track that spans four logic blocks. A 180nm process was assumed.

The graph illustrates that pulses less than or equal to 200 ps in duration are mostly filtered out the routing resources. Pulses that are longer than 300 ps in duration dissipate approximately the same amount of power as longer pulses. Thus, if the input signals of a gate arrive within a 200 ps window, the glitching of that gate is effectively eliminated.

Table 1. Breakdown of Switching Activity

Circuit	Logic Depth	Activity	Func. Activity	Glitch Activity	% Glitch
C1355	4	0.32	0.23	0.09	27.5
C1908	10	0.26	0.17	0.09	34.6
C2670	7	0.27	0.21	0.06	22.2
C3540	12	0.42	0.23	0.19	45.2
C432	11	0.26	0.18	0.08	29.3
C499	4	0.34	0.23	0.11	31.9
C5315	10	0.40	0.25	0.15	36.7
C6288	28	1.56	0.29	1.27	81.1
C7552	9	0.39	0.23	0.16	42.0
C880	9	0.23	0.19	0.05	19.8
alu4	7	0.08	0.07	0.01	13.1
apex2	8	0.05	0.04	0.01	13.7
apex4	6	0.04	0.03	0.01	32.3
des	6	0.27	0.17	0.10	36.8
ex1010	8	0.03	0.01	0.02	52.9
ex5p	7	0.17	0.08	0.09	51.0
misex3	7	0.06	0.05	0.01	20.9
pdcc	9	0.03	0.02	0.01	31.8
seq	7	0.05	0.04	0.01	16.0
spla	8	0.05	0.03	0.02	42.7
Geomean	8.1	0.024	0.019	0.047	30.8

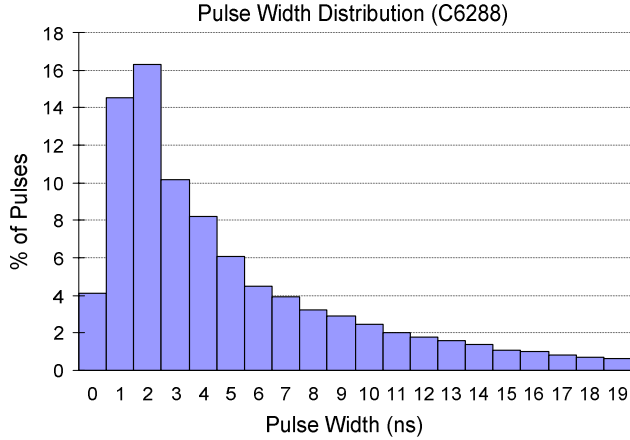


Figure 1. Pulse width distribution of glitches.

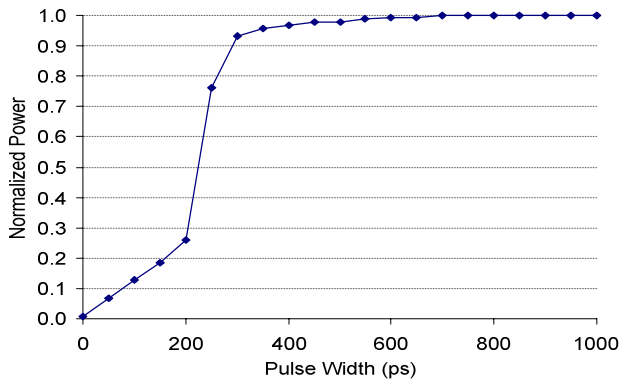


Figure 2. Normalized power vs. pulse width.

3.4 Potential Power Savings

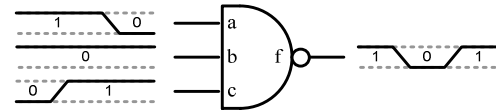
Table 2 reports the average total power dissipated by circuits when implemented on an FPGA. The first column reports the power of the circuits in the normal case, when glitching is allowed to occur. The third column reports the power in the ideal case, when glitching is eliminated with no overhead. The fourth column shows the percent difference between the two power estimates; this number indicates how much power could be saved if glitching was completely eliminated without any overhead. Depending on the circuit, the potential power saving ranges between 4% and 73%, with average savings of 22.6%. These numbers motivate a technique for reducing glitching in FPGAs.

Table 2. FPGA power with and without glitching

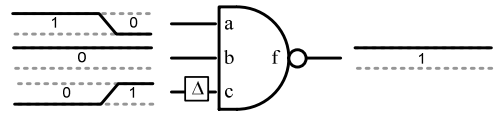
Power (mW) (glitching)	Power (mW) (no glitching)	% Difference
24.3	18.8	22.6

4. PROPOSED TECHNIQUE

Our proposed technique involves adding programmable delay elements within the logic blocks of the FPGA. Within each logic block, we delay early-arriving signals so as to align the edges on each LUT input, thereby reducing the number of glitches on the output of each LUT. The technique is shown in Figure 3; by delaying input *c*, the output glitch can be eliminated. Since only the early-arriving input(s) are delayed, the overall critical path of the circuit is not increased.



(a) Original circuit with glitch



(b) Glitch removed by delaying input *c*

Figure 3. Removing glitches by delaying early-arriving signals.

We consider four alternative schemes for implementing this technique; the schemes differ in the location of the delay elements within the configurable logic block. In this section, we first describe the programmable delay element that is common to all four schemes. Then we describe each scheme, showing how the delay elements are used to align edges. Finally, we describe the CAD algorithms that are used to determine the configuration of each programmable delay element after place and route.

4.1 Programmable Delay Element

Figure 4 illustrates the programmable delay elements used in each of the schemes. The circuit is composed of two inverters. The first inverter has programmable pull-up and pull-down resistors to control the delay of the circuit. The second inverter has large channel lengths to minimize short-circuit power.

The pull-up and pull-down resistors each have n stages with a resistor and a bypass transistor controlled by an SRAM bit. The first stage has a resistance of R and the resistance of the subsequent stages is doubled for each stage. Using the control bits, this circuit can be programmed to produce any delay $\Delta \in \{k, \tau + k, 2\tau + k, 3\tau + k, \dots, (2^n - 1)\tau + k\}$, where τ is the delay produced by a resistance R to charge or discharge the capacitor C and k is the delay produced by the bypass resistances and the inverters.

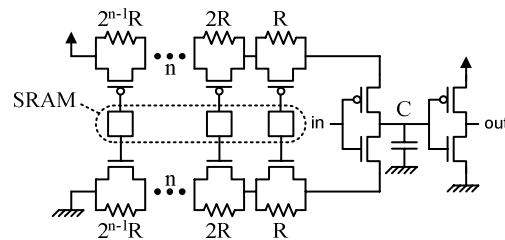


Figure 4. Programmable delay element.

Figure 5 illustrates the pull-up and pull-down resistor circuits. The pull-up circuit is a PMOS pass-transistor and the pull-down circuit is a NMOS pass-transistor. Bias circuits are used to control the gate voltage of the pass-transistors to produce a large resistance. One pull-up and one pull-down bias circuit are shared by all the pass-transistors in a programmable delay element. The different resistances needed by the different stages are obtained by changing the length of the pass-transistors.

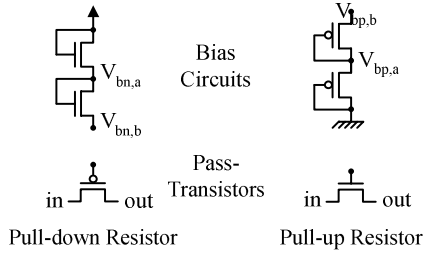


Figure 5. Resistor circuits.

The delay of the programmable delay element is affected by temperature, supply noise, and process variation. Although not addressed in this paper, these factors are important since adding more delay than necessary may affect the critical-path delay of the implementation and not adding enough delay will reduce the amount of glitching that can be eliminated. Ideally, the delay variation of the programmable delay element will scale with the delay variation of the FPGA routing resources.

4.2 Architectural Alternatives

Figure 6(a) illustrates the baseline configurable logic block (CLB). A CLB consists of LUTs, flip-flops, and local interconnect. The LUTs and FFs are paired together into Basic Logic Elements (BLEs). Three parameters are used to describe a CLB: I specifies the number of input pins, N specifies the number of BLEs and output pins, and K specifies the size of the LUTs. The local interconnect allows each BLE input to choose from any of the I CLB inputs and N BLE outputs. Each BLE output drives

a CLB output. The four schemes we consider for adding delay elements to a configurable logic block are illustrated in Figure 6(b) to 6(e). Each of are described below.

In Scheme 1, the programmable delay elements are added at the input of each LUT, as shown in Figure 6(b). This architecture allows each LUT input to be delayed independently. We describe the architecture using three parameters: min_in , max_in , and num_in . The min_in parameter specifies the precision of the delay element connected to each LUT input. Intuitively, more glitching can be eliminated when min_in is small since the arrival times can be aligned more precisely. On the other hand, there is more overhead when min_in is small since each programmable delay element requires more stages to provide the extra precision. The max_in parameter specifies the maximum delay that can be added to each LUT input. Intuitively, more glitching can be eliminated when max_in is large since wider glitches can be eliminated. However, there is more overhead when max_in is large. Finally, the num_in parameter specifies how many LUT inputs have a programmable delay element, between 1 and K (the number of inputs in each LUT). Increasing num_in reduces glitching but increases the overhead. In Section 6, we quantify the impact of these parameters on the power, area, and delay of this scheme.

The disadvantage of Scheme 1 is that, since some inputs need very long delays for alignment, large programmable delay elements area required. Since num_in delay elements are needed for every LUT, this technique has a high area overhead if num_in

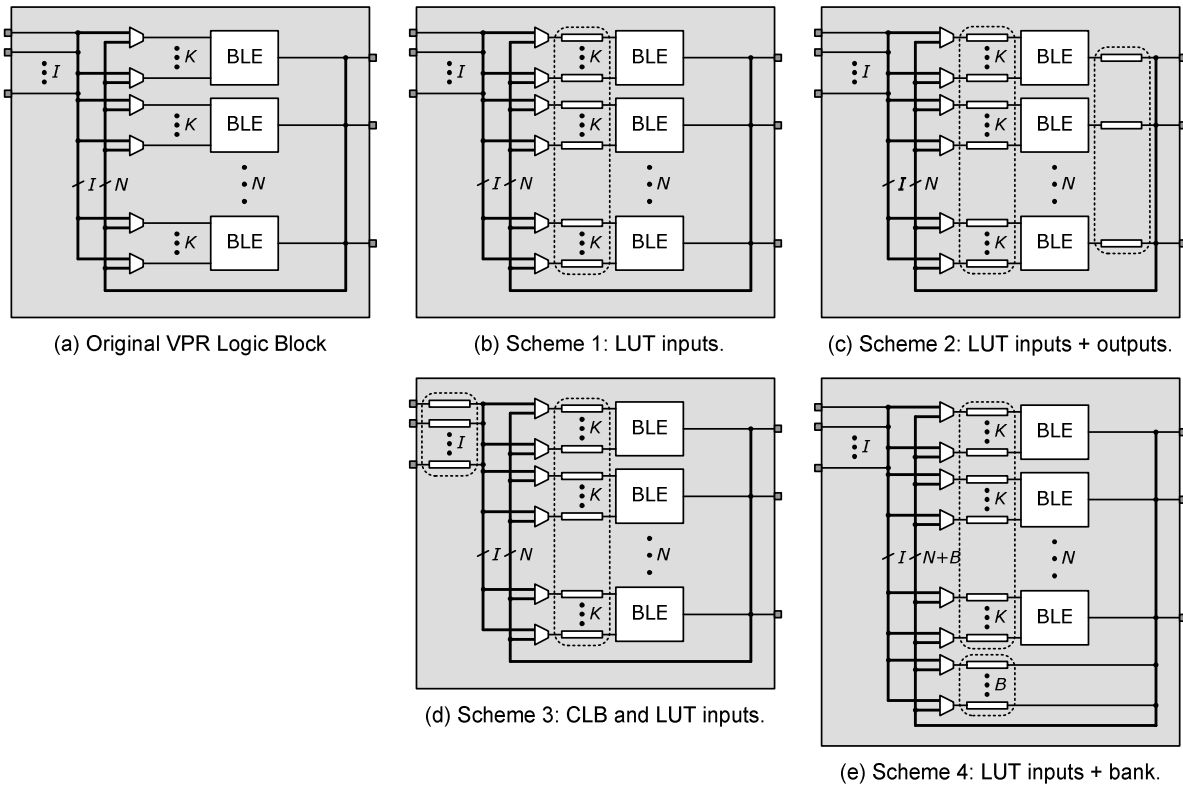


Figure 6. Delay insertion schemes

is large. In Scheme 2, shown in Figure 6(b), additional programmable delay elements are added to the outputs of LUTs (we refer to these new delay elements as *LUT output delay elements*). With this architecture, a single LUT output delay element could be used to delay a signal that fans out to several sinks, potentially reducing the size and the number of delay elements required at each LUT input. We describe the LUT output delay elements using two parameters, *min_out* and *max_out*, which specify the minimum and maximum delay of the output delay elements. The LUT input delay elements are described using the same parameters as Scheme 1.

Scheme 3, shown in Figure 6(c), is another way to reduce the area required for the LUT input delay elements. In this scheme, additional delay elements, which we call *CLB input delay elements*, are added to each of the *I* CLB inputs. Since there are typically fewer CLB inputs than there are LUT inputs in a CLB, this could potentially result in an overall area savings. The parameters *min_c* and *max_c* specify the minimum and maximum delay of the CLB input delay elements. We assume every CLB input has a delay element, in order to maintain the equivalence of each CLB input.

Finally, Scheme 4, shown in Figure 6(d), reduces the size of the LUT input delay elements by adding a bank of delay elements which can programmably be used by all LUTs in a CLB. We refer to these delay elements as *bank delay elements*. Signals that need large delays can be delayed by the bank delay elements, while signals that need only small delays can be delayed by the LUT input delay elements. In this way, the LUT input delay elements can be smaller than they are in Scheme 1. These bank delay elements can be described using two additional parameters: *max_b* and *num_b*. The *max_b* parameters specify the maximum delay of the bank delay elements and the *num_b* parameter specifies the number of programmable delay elements in the bank. Note that we assume that the minimum delay of the bank delay element is equal to the maximum delay of the LUT input delay element since only one of delay elements needs to add precision.

The parameters used to describe each scheme are summarized in Table 3 below. The area and delay overhead for each scheme, as well as their ability to reduce glitches, will be quantified in Section 6.

Table 3: Architectural parameters

Scheme	Parameter	Meaning
All	<i>min_in</i>	Min delay of LUT input delay element
	<i>max_in</i>	Max delay of LUT input delay element
	<i>num_in</i>	# of LUT input delay elements / LUT
2	<i>min_out</i>	Min delay of LUT output delay element
	<i>max_out</i>	Max delay of LUT output delay element
3	<i>min_c</i>	Min delay of CLB input delay element
	<i>max_c</i>	Max delay of CLB input delay element
4	<i>max_b</i>	Max delay of bank delay element
	<i>num_b</i>	# of bank delay elements / CLB

4.3 CAD Algorithms

This section describes the algorithms used to determine the configuration of each programmable delay element. This configuration occurs after placement and routing, when accurate delay information is available.

Regardless of the architecture scheme used, a quantity *Needed_Delay* is first calculated for each LUT input. This quantity, which indicates how much delay should be added to the LUT input so that all LUT inputs transition at the same time, is calculated using the algorithm in Figure 7.

```

calc_needed_delays (circuit) {
    calc_arrival_times (circuit);
    foreach node n ∈ circuit {
        foreach fanin f ∈ n
            Needed_Delay(n, f) = Arrival_Time(n) - Arrival_Time(f) -
                                Fanin_Delay(n, f);
    }
}

```

Figure 7. Pseudo-code for calculating the delay needed to align the inputs

The next step is to implement a delay as close to *Needed_Delay* as possible for each LUT input. Since, in all but the first scheme, signals can be delayed in more than one way, there is more than one way to implement the needed delay. The technique used is different for each scheme.

The algorithm used to calculate the configuration of each LUT input delay element in Scheme 1 is shown in Figure 8. In this case, there is only one way to insert delays, so the algorithm is straightforward. Note that the granularity of the delay elements (*min_in*) and the number of delay elements attached to each LUT (*num_in*) will affect how closely the inserted delays match the desired values (found in the algorithm of Figure 7).

The algorithm for Scheme 2 is shown in Figure 9. This algorithm first visits each LUT in topological order from the inputs to the outputs and determines the minimum delay needed by all the fanouts of that LUT. It then configures the output delay element to match this delay and then updates the needed delay value of each fanout. It then configures the input delays as in Scheme 1.

The third scheme, which incorporates programmable delay elements at the CLB inputs and LUT inputs, uses the algorithm described in Figure 10 to the configure the CLB input delay elements and then uses the algorithm described in 8 to configure the LUT input delays. The algorithm visits each CLB input and determines the minimum delay needed by the LUT inputs that are driven by that input. It then configures the CLB input delay element to match the minimum delay and updates the needed delay of the affected LUT inputs to reflect the change

Finally, the fourth scheme, which incorporates a bank of programmable delay elements in addition to those at the LUT inputs, uses the algorithm described in Figure 11 to configure the bank of delay elements. The algorithm visits each CLB in the circuit and configures the bank circuits to delay signals that need to be delayed by more than *max_in* and smaller or equal to *max_b*. When the algorithm finds a signal that needs a delay that is greater than *max_in*, it calculates the amount of delay that it can add to a signal (by a delay element in the bank) and then updates the needed delay to reflect the change for the subsequent LUT input algorithm. The count variable is used to limit the number bank delay elements that are used for each CLB. After the

configuration for each bank delay element is found, the algorithm from Figure 8 is used to calculate the configuration for each LUT input delay element.

```

scheme1 (circuit, min_in, max_in, num_inl)
{
  config_LUT_input_delays (circuit, min_in, max_in, num_in);
}
config_LUT_input_delays (circuit, min_in, max_in,
                          num_in) {
  foreach LUT n ∈ circuit {
    count = 0;
    foreach fanin f ∈ n {
      if (Needed_Delay(n, f) > min_in &&
          Needed_Delay(n, f) ≤ max_in &&
          count < num_in)
      {
        Added_Delay(n, f) = min_in *
          floor(Needed_Delay(n, f) / min_in);
        Needed_Delay(n, f) =
          Needed_Delay(n, f) - Added_Delay(n, f);
        count++;
      }
    }
  }
}

```

Figure 8. Pseudo-code for assigning delays in Scheme 1.

```

scheme2 (circuit, min_in, max_in, num_in, min_out, max_outl)
{
  config_output_delays (circuit, min_out, max_out);
  config_LUT_input_delays (circuit, min_in, max_in, num_in);
}
config_output_delays (circuit, min_out, max_out)
{
  foreach LUT n ∈ circuit {
    min = max_out;
    foreach fanout f ∈ n {
      if (Needed_Delay(f, n) < min) {
        min = Needed_Delay(f, n);
      }
    }
    if (min ≥ min_out) {
      foreach fanout f ∈ n {
        Added_Delay(f, n) = min_out * floor(min / min_out);
        Needed_Delay(f, n) =
          Needed_Delay(f, n) - Added_Delay(f, n);
      }
    }
  }
}

```

Figure 9. Pseudo-code for assigning additional delays in Scheme 2.

```

scheme3 (circuit, min_in, max_in, num_in, min_clb, max_clb)
{
  config_CLB_input_delays (circuit, min_clb, max_clb);
  config_LUT_input_delays (circuit, min_in, max_in, num_in);
}
config_CLB_input_delays (circuit, min_clb, max_clb) {
  foreach CLB c ∈ circuit {
    foreach input i ∈ c {
      min = max_clb;
      foreach fanout f ∈ i {
        if (f ∈ c && Needed_Delay(f, i) < min) {
          min = Needed_Delay(f, i);
        }
      }
    }
  }
}

```

```

if (min ≥ min_clb) {
  foreach fanout f ∈ i {
    Added_Delay(f, i) = min_clb * floor(min / min_clb);
    Needed_Delay(f, i) =
      Needed_Delay(f, i) - Added_Delay(f, i);
  }
}
}
}
}
}

```

Figure 10. Pseudo-code for assigning additional delays in Scheme 3.

```

scheme4 (circuit, min_in, max_in, num_in, max_b, num_b)
{
  config_bank_delays (circuit, max_in, max_b, num_b);
  config_LUT_input_delays (circuit, min_in, max_in, num_in);
}
config_bank_delays (circuit, max_in, max_b, num_b) {
  foreach CLB c ∈ circuit {
    count = 0;
    foreach LUT n ∈ c {
      foreach fanin f ∈ n {
        /* Note: min_b == max_in */
        if (Needed_Delay(n, f) > max_in &&
            Needed_Delay(n, f) ≤ max_in + max_b &&
            count < num_b)
        {
          Added_Delay(n, f) = max_in *
            floor(Needed_Delay(n, f) / max_in);
          Needed_Delay(n, f) =
            Needed_Delay(n, f) - Added_Delay(n, f);
          count++;
        }
      }
    }
  }
}

```

Figure 11. Pseudo-code for assigning additional delays in Scheme 4.

5. EXPERIMENTAL FRAMEWORK

This section describes the experimental framework that is used to obtain the switching activity information and the FPGA area, delay, and power estimates that are presented in this paper.

5.1 Switching Activity Estimation

The switching activities are obtained by simulating circuits at the gate level and counting the toggles of each wire. The simulations are driven by pseudo-random input vectors and circuit delay information from the VPR place and route tool [11]. To capture the filtering effect of the routing FPGA routing resources and of the programmable delay elements, the simulator uses the *inertial delay* model. Furthermore, to replicate an FPGA routing architecture consisting of length 4 routing segments, the VPR delays are divided into chains of 300ps delay.

5.2 Area, Delay, and Power Estimation

Area, delay, and power estimates are obtained from the Versatile Place and Route (VPR) tool [11]. VPR models an FPGA at a low-level, taking into account specific switch patterns, wire lengths, and transistor sizes. After generating a specified FPGA architecture, VPR places and routes a circuit on the FPGA and then models the area, delay, and power of that circuit.

VPR models area by summing the area of every transistor in the FPGA, including the routing, logic blocks, clock network, and configuration memory. The area of each transistor is approximated using the Minimum Transistor Equivalents (MTE), as described in [11]. Delay and power are modeled after routing, when detailed resistance and capacitance information can be extracted for each net in the circuit. The Elmore delay model is used to produce delay estimates and the FPGA power model described in [12] is used to produce power estimates. The FPGA power uses the VPR capacitance information and externally generated switching activities to estimate dynamic, short-circuit, and leakage power.

5.3 Architecture Assumptions and Benchmark Circuits

We gathered results for three LUT sizes: 4 inputs, 5 inputs, and 6 inputs. In call cases, we assumed each CLB contains 10 LUTs. To maintain routability, we assume that the architecture with 4-input LUTs has CLBs with 22 inputs, the architecture with 5-input LUTs has CLBs with 27 inputs, and the architecture with 6-input LUTs has CLBs with 33 inputs. We further assume a routing fabric containing buffered length-4 routing tracks. In each experiment, we used 20 combinational benchmark including the 10 largest combinational circuits from the MCNC and ISCAS89 benchmark suites. Before placement and routing, each circuit is mapped to lookup-tables using the Emap technology mapper [5] and packed into clusters using the T-VPack clusterer [11].

6. RESULTS

This section begins by calibrating the parameters of the four delay insertion schemes described in Section 4. Each scheme is calibrated to eliminate most of the glitching while minimizing the area and delay overhead. After finding suitable values for each parameter, the four schemes are compared to determine which scheme produces the best results.

6.1 Scheme 1 Calibration

We first consider the min_in parameter, which defines the minimum delay increment of the programmable delay element at the inputs of the LUTs. Intuitively, a smaller delay increment reduces glitching but increases area. Figure 12 shows how much glitching is eliminated for minimum delay increments ranging between 0.1 and 3.2ns. To isolate the impact of the min_in parameter, the graph assumes that every LUT input has a programmable delay element with an infinite maximum delay (max_in is ∞ and num_in is K).

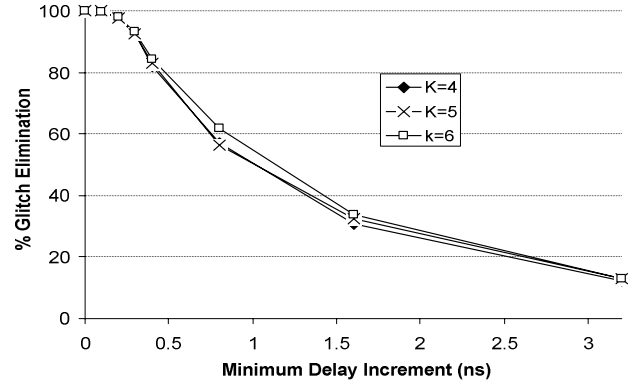


Figure 12. Glitch elimination vs. minimum LUT input delay for Scheme 1

The graph illustrates that most of the glitching can still be eliminated when the minimum delay increment is 0.25ns. This corresponds to the fact that narrow glitches are filtered away by the routing resources and that the majority of glitches have a width greater than 0.2ns, as described in Section 3. The same conclusion holds for FPGAs that use 4-input, 5-input, or 6-input LUTs.

The second parameter, denoted max_in , defines the maximum delay of the programmable delay element at the inputs of the LUTs. Intuitively, increasing the maximum delay reduces glitching but increases area. Figure 13 shows how much glitching is eliminated as a function of the maximum delay. The graph illustrates that over 90% of the glitching can be eliminated when the maximum delay of the programmable delay element is 8.0ns. This corresponds with Figure 1, which illustrates that the majority of glitches have a width that is less than 10.0ns.

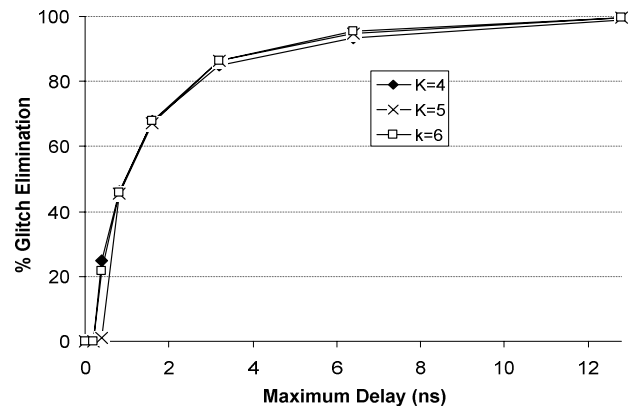


Figure 13. Glitch elimination vs maximum LUT input delay for Scheme 1

Finally, num_in defines the number of LUT inputs that have a programmable delay elements. Intuitively, increasing the number of inputs with delay elements reduces glitching since the arrival times of more inputs can be aligned. Figure 14 shows how much glitching is eliminated when the number of inputs with programmable delays is varied. The graph assumes that the minimum delay increment is $1/\infty$ and the maximum delay is ∞ .

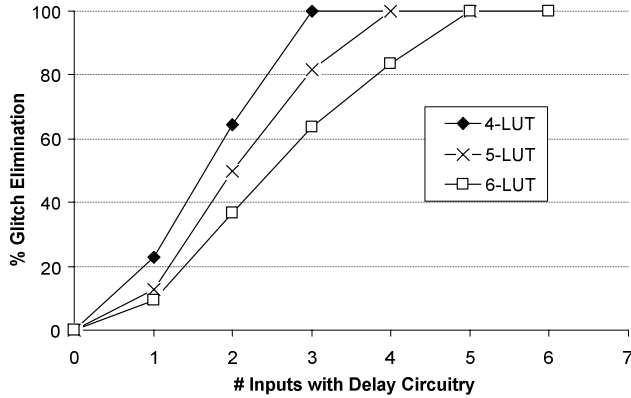


Figure 14. Glitch elimination vs. number of input delay elements per LUT for Scheme 1

The graph illustrates that each LUT should have a programmable delay element on every input minus one ($K-1$). Intuitively, adding delay circuitry to every input is not necessary since each LUT has at least one input that does not need to be delayed (the slowest input). However, adding fewer than $K-1$ delay elements significantly reduces the amount of glitching that can be eliminated.

6.2 Scheme 2 Calibration

The second delay insertion scheme has five parameters, namely: min_in , max_in , num_in , min_out , and max_out . The first three parameters control the delay elements and the inputs of the LUTs; the last two parameters control the delay elements at the output of the LUTs. Although the min_in , max_in , and num_in parameters were already calibrated for Scheme 1, they must be recalibrated for Scheme 2 since the output delay elements change how much delay is needed by LUT input delay elements. Intuitively, however, the value of the min_in parameter can be reused since the LUT input delays are still used to perform the final alignment of each signal.

The max_in and num_in are both recalibrated assuming min_out is infinitely precise ($1/\infty$) and max_out is ∞ . Figure 15 shows then glitch elimination for max_in from 0 to 12ns assuming again that min_in is $1/\infty$ and num_in is K . The results are similar to those in Scheme 1 except that some glitching is eliminated when max_in is 0 since the output delay elements are aligning some of the inputs. Again, most of the glitching can be eliminated when max_in is set to 8.0ns.

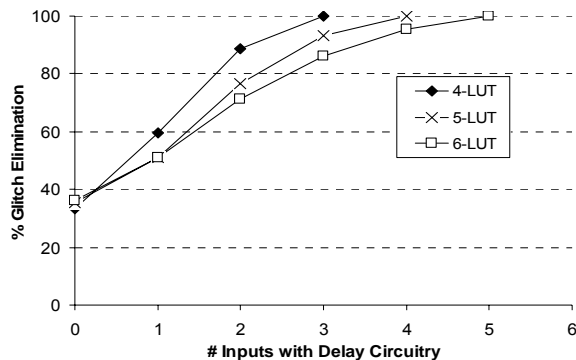


Figure 16 shows glitch elimination with respect to num_in . As

before, the graph assumes that min_in is $1/\infty$ and max_in is ∞ . The graph shows that more glitching is eliminated using fewer LUT input delay elements when the output delays are used. In Scheme 2, most of the glitching can be eliminated when num_in is $K-2$.

The remaining output delay element parameters are calibrated assuming min_in is 0.25, max_in is 8.0, and num_in is $K-2$. Figure 17 shows the glitch elimination for min_out from 0 to 3.2ns assuming that max_out is ∞ and Figure 18 shows the glitch elimination for max_out from 0 to 12ns assuming that min_out is $1/\infty$. The graphs illustrate that a 0.25 and 8.0 are also suitable for min_out and max_out , respectively.

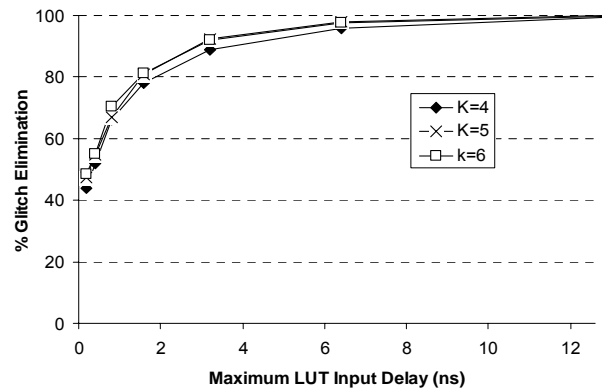


Figure 15. Glitch elimination vs. maximum LUT input delay for Scheme 2

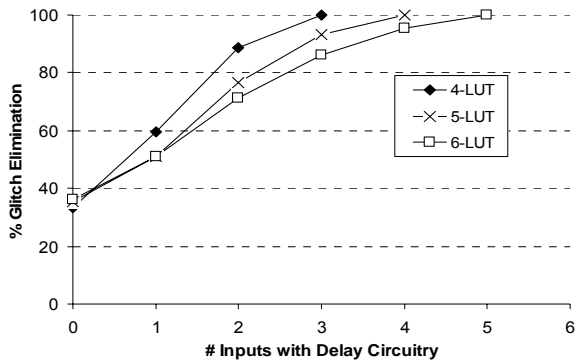


Figure 16. Glitch elimination vs. number of input delay elements per LUT for Scheme 2

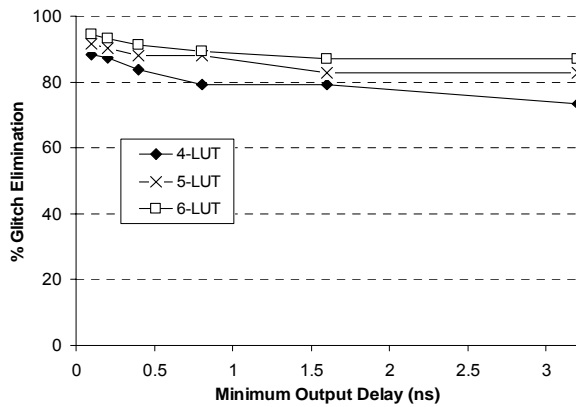


Figure 17: Glitch elimination vs. minimum LUT output delay for Scheme 2

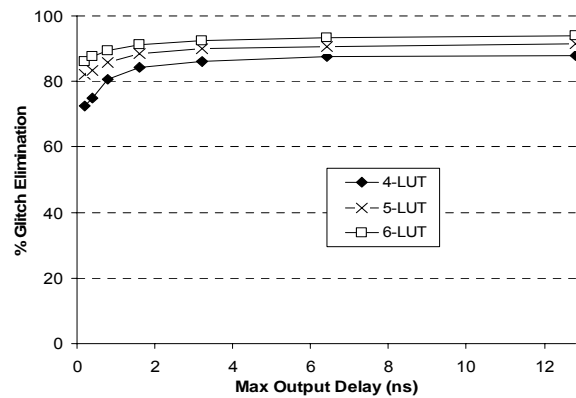


Figure 18: Glitch elimination vs. maximum LUT output delay for Scheme 2

6.3 Scheme 3 Calibration

The third delay insertion scheme has five parameters, namely: min_in , max_in , num_in , min_c , and max_c . The first three parameters control the delay elements at the inputs of the LUTs; the last two parameters control the delay elements at the input of the CLBs. The min_in , max_in , and num_in parameters were again recalibrate to account for the affect of the CLB input delay elements. The same procedure used in Scheme 2 was used. The

results for min_in and max_in were similar to the previous cases, which indicated that 0.25ns and 8.0ns were suitable, respectively.

The results for num_in , which are plotted in Figure 19, were different than in the previous cases. To isolate the impact of num_in , the graph assumes that min_in is $1/\infty$, max_in is ∞ , min_c is $1/\infty$, and max_c is ∞ . The results indicate that num_in should be 1, 2, and 2, for 4, 5, and 6-LUTs, respectively. Intuitively, fewer LUT input delay elements are needed since the CLB input delay elements account for most of the delay. Only in cases where the CLB inputs fanout to multiple LUTs within that CLB and those fanouts need different delays are the LUT input delay elements required.

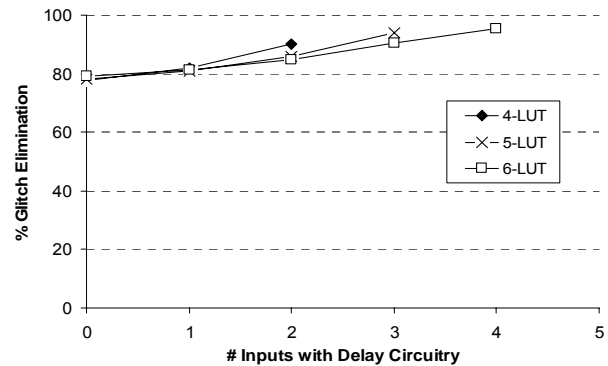


Figure 19: Glitch elimination vs. number of input delay elements per LUT for Scheme 3

6.4 Scheme 4 Calibration

The fourth delay insertion scheme has five parameters, namely: min_in , max_in , num_in , max_b , and num_b . The first three parameters control the delay elements and the inputs of the LUTs; the last two parameters control the bank of delay elements in the CLB. The bank of programmable delay elements are only used for signals that need more delay than can be added by the LUT input delay elements, therefore this scheme uses the same min_in and num_in values as Scheme 1: 0.25ns and K-1, respectively. Suitable values for max_in and max_b were found empirically to be 3.2ns and 8.0ns, respectively. Finally, Figure 20 shows glitch elimination with respect to the number of bank delay elements per CLB (num_b) assuming min_in is 0.25ns, num_in is K-1, max_in is 4.0ns, and max_b is 8.0ns.

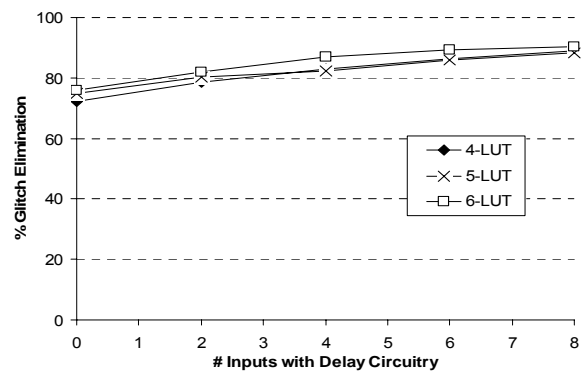


Figure 20: Glitch elimination vs. number of bank delay elements for Scheme 4

6.5 Overhead

The circuitry added to the CLBs to minimize glitching has an area, delay, and power overhead. The overhead for each scheme is examined below.

Area Overhead

The area overhead is determined by summing the area of the added delay circuitry in each logic block. This area includes the area of the delay elements and the added configuration memory. Table 4 reports how much area is needed in the logic blocks and Table 5 reports the percent area overhead taking logic block and routing area into account.

In general, Scheme 4 has a greater area overhead than Schemes 1, 2, and 3, which have similar area overheads. Scheme 4 requires more area because of the large multiplexers needed to select which CLB input or LUT output uses the bank delay elements. Moreover, the area overhead tends to decrease slightly as the LUT size increases since the area of the LUTs and multiplexers increases exponentially with K, while the area of the delay elements only increases linearly.

Table 4: Overhead area per CLB

LUT Size	Original CLB Area (MTE)	Overhead Area (MTE)			
		Scheme 1	Scheme 2	Scheme 3	Scheme 4
4	6938	1620	1620	1728	2344
5	10361	2160	2160	2538	2688
6	15228	2700	2700	2862	3256

Table 5: Average area overhead

LUT Size	Average Area Overhead (%)			
	Scheme 1	Scheme 2	Scheme 3	Scheme 4
4	5.3	5.3	5.7	7.7
5	5.0	5.0	5.9	6.3
6	4.4	4.4	4.6	5.3

Power Overhead

Even if all the glitches could be eliminated, the programmable delay elements still dissipate power. This overhead is modeled by summing the power dissipated by the added circuitry in each logic block of the FPGA using the expression below.

$$P(\text{circuit}) = \frac{\sum_{n \in \text{dnodes}} E_{\text{toggle}} \cdot \alpha(n)}{T_{\text{crit}}}$$

In the expression, dnodes is the set of nodes in the circuit that can be delayed, E_{toggle} is the energy dissipated by one programmable delay element during one transition, $\alpha(n)$ is the switching activity of the delayed node n , and T_{crit} is the critical path delay of the circuit. The energy of the programmable delay element is determined using HSPICE, the switching activity is determined using gate level simulation, and the critical path delay is determined using the VPR place and route tool. **Table 6** reports the average power dissipated by the added delay circuitry for each scheme. The power overhead is approximately 1% for all the schemes. Scheme 1; however, has the lowest power overhead.

Table 6: Average power overhead

LUT Size	Average Power Overhead (%)			
	Scheme 1	Scheme 2	Scheme 3	Scheme 4
4	0.89	0.99	1.15	0.95
5	0.94	1.12	1.25	0.98
6	0.98	1.09	1.07	0.87

Delay Overhead

Although the delay elements are programmed to only add delay to early arriving edges, a small delay penalty may be incurred even if the delay element is bypassed because of parasitic resistance and capacitance. To model delay overhead, HSPICE was used to determine the parasitic delay incurred by the delay element. The critical-path delay of each circuit was then recalculated, taking these parasitic delays into account. Finally, the overhead was calculated by comparing the new critical-path delay to the original critical-path delay.

Table 7 reports the average delay overhead for each scheme. Schemes 1 and 4 have the smallest overhead since both have *fast-paths* with no delay elements (no parasitics) to slow down the critical-path. Schemes 2 and 3 have a larger overhead, since neither scheme offer a *fast-path* for critical-path connections.

Table 7: Average delay overhead

LUT Size	Average Delay Overhead (%)			
	Scheme 1	Scheme 2	Scheme 3	Scheme 4
4	0.21	2.4	2.3	0.21
5	0.13	2.2	2.1	0.13
6	0.14	2.1	1.9	0.14

Table 8: % Glitch elimination of each scheme

Scheme 1	Scheme 2	Scheme 3	Scheme 4
91.8%	83.3%	81.8%	85.4%

Table 9: Overall power savings

Circuit	Power Saving (%)			
	Scheme 1	Scheme 2	Scheme 3	Scheme 4
C135	25.4	25.0	25.0	25.8
C1908	18.1	18.4	16.1	17.0
C2670	11.6	11.3	10.2	11.7
C3540	27.5	22.9	23.5	26.3
C432	13.0	10.7	10.6	10.6
C499	31.8	30.9	32.3	32.4
C5315	18.2	16.2	16.0	17.9
C6288	52.1	43.2	40.0	46.1
C7552	22.6	18.9	19.7	22.3
C880	7.2	6.5	8.0	7.1
alu4	2.5	2.4	3.3	2.7
apex2	3.6	3.2	3.8	3.6
apex4	9.5	9.1	9.4	9.3
des	15.1	12.1	14.2	14.4
ex1010	16.8	16.4	16.5	15.9
ex5p	23.8	23.4	21.5	25.0
misex3	7.6	7.3	7.3	7.2
pdc	11.1	10.1	10.7	11.3
seq	5.3	5.9	5.7	5.6
spla	20.3	19.8	20.0	20.2
Average	18.2	16.3	16.2	17.4

6.6 Overall Results

Finally, Table 8 and Table 9 present the overall glitch elimination and power savings for each scheme, respectively. Both tables report the results for 4-LUTs only since the results for 5 and 6 input LUTs were similar. Both tables indicate that Scheme 1 produces the best results, with 91.8% glitch elimination and overall power savings of 18.2%. The power savings are relatively close to the ideal savings of 22.6%.

7. CONCLUSIONS

This paper proposed an active glitch elimination technique to minimize dynamic power in FPGAs. The technique involves adding programmable delay elements within the logic blocks of the FPGA to align the edges on each LUT input and filter out existing glitches, thereby reducing the number of glitches on the output of each LUT. Four alternative schemes were considered for implementing this technique. Scheme 1, which involved adding programmable delay elements to $K-1$ inputs of each LUT produced the greatest power savings with the lowest overhead in terms of area and critical-path delay. On average, the proposed technique eliminates 91% of the glitching, which reduces overall FPGA power by 18.2%. The added circuitry increases overall area by 5.3% and critical-path delay by only 0.2%.

8. ACKNOWLEDGMENTS

This research was funded by Altera and the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

- [1] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger, A 90nm low-power FPGA for battery-powered applications, Intl. Symp. on Field-Programmable Gate Arrays (FPGA), pp. 3-11, 2006.
- [2] J. C. Monteiro and A. L. Oliveira, Finite state machine decomposition for low power, Proc. 35th Design Automation Conference (DAC), pp. 758-763, 1998.
- [3] D. Kim and K. Choi, Power conscious high-level synthesis using loop folding, Proc. 34th Design Automation Conference (DAC), pp. 441-445, 1997.
- [4] M. Kandemir et al, Influence of compiler optimizations on system power, IEEE Trans. VLSI, 9(6):801-804, 2001.
- [5] J. Lamoureux and S. Wilton, On the interaction between power-aware FPGA CAD algorithms, Proc. Intl. Conference on Computer-Aided Design (ICCAD), pp. 701-708, 2003.
- [6] D. Chen, J. Cong, F. Li, and L. He, Low-power technology mapping for FPGA architectures with dual supply voltages, Intl. Symp. on Field-Programmable Gate Arrays (FPGA), pp. 109-117, 2004.
- [7] J.C. Monteiro, S. Devadas and A. Ghosh, Retiming sequential circuits for low power, Proc. 35th Design Automation Conference (DAC), pp. 398-402, 1993.
- [8] S. Wilton, S.-S. Ang and W. Luk, , The impact of pipelining on Energy per operation in field-programmable gate arrays, Proc. Intl. Conf. on Field-Programmable Logic and its Applications, pp. 719-728, 2004.
- [9] L. Benini et al, Glitch power minimization by selective gate freezing, IEEE Trans. VLSI Systems, 8(3): 287-298, 2000.
- [10] A. Raghunathan, S. Dey and N. K. Jia, Register transfer level power optimization with emphasis on glitch analysis and reduction, IEEE Tras. CAD, 18(8): 114-1131, 1999.
- [11] V. Betz., J. Rose, and A. Marquardt, Architecture and CAD For Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.
- [12] K.K.W. Poon, S.J.E. Wilton, A. Yan, A Detailed Power Model for Field-Programmable Gate Arrays", in ACM Trans. on Design Automation of Electronic Systems (TODAES), Vol. 10, No. 2, pp. 279-302, April 2005.