Pseudo-Constant Logic Optimization

Aaron Landy and Greg Stitt
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL, USA
landy@hcs.ufl.edu, gstitt@ece.ufl.edu

***Abstract*—Constant folding reduces area and enables greater parallelism, but requires circuits with constant inputs. In this work, we extend constant folding to support *pseudo-constants*, which are values that change with low frequency. We present a method of pseudo-constant logic optimization based on dynamically reconfigurable capabilities of FPGAs, which optimizes logic for different pseudo-constant values and then reconfigures the logic whenever the pseudo-constant changes. Although not beneficial for all logic, we show this optimization achieves up to a 1.25x increase in functional density on Xilinx Virtex 5 FPGAs.**

Keywords—FPGA; pseudo-constants; logic minimization; dynamic reconfigurability; run-time reconfiguration

I. Introduction

Constant folding [5] is a widely studied logic-minimization strategy for FPGAs. Unfortunately, circuit designers often avoid constants to enable as many use cases as possible, limiting constant-folding applicability.

However, circuits often use signals that exhibit near-constant behavior where the value is rarely changed, which we define as *pseudo-constant*. For example, many signal-processing applications initially set a pseudo-constant convolution kernel, which remains the same for the duration of the application. Alternatively, each frame of a low frame-rate video may also be considered pseudo-constant. These pseudo-constants are often inputs to highly replicated logic components such as adders, multipliers, comparators, and muxes (e.g., [2][8]), which could benefit from constant folding to reduce area and/or increase replication.

In this paper, we introduce *pseudo-constant logic optimization.* To account for invalidated logic resulting from changes in a pseudo-constant value, we exploit lookup-table (LUT) reconfigurability to dynamically modify the logic. We show that for common types of logic, pseudo-constant logic optimizations can achieve area savings from 27%-50% on Xilinx Virtex 5 FPGAs. Additionally, we show that pseudo-constant optimized multiplexers match the functional density of traditional synthesis in as few as 128 operations per invalidation, and approach up to 1.25x greater functional density for infrequent invalidations.

II. Previous Work

Previous studies have demonstrated a concept similar to pseudo-constants by using partial reconfiguration for run-time logic minimization [4][7][9][11]. Previous work also showed that partial reconfiguration can have prohibitive reconfiguration times, implementation complexity, and limitations on reconfiguration granularity [3][10][11]. This past work examined trade-offs between area and reconfiguration time when using run-time logic optimization, and included a functional density metric to quantify the trade-offs. We extend past work by reducing reconfiguration times and implementation complexity via the LUT-based RAM primitives provided by most FPGAs.

III. Pseudo-Constant Logic Optimization

A. Overview

Pseudo-constant and traditionally optimized circuits are identical after elaboration but differ significantly after technology mapping. Consider a 4:1 multiplexer with a constant or pseudo-constant select input. Traditional constant folding would replace the multiplexer with a direct connection. However, pseudo-constant mapping requires more resources to enable changes due to invalidations.

To allow for changes, technology mapping for pseudo-constant logic is restricted to FPGA primitives that support runtime reconfiguration. In this paper we focus on common primitives in existing Xilinx Virtex 5 devices: *LUT RAM* and *LUT shift registers*. Pseudo-constants are also possible on Altera devices but are outside the scope of this paper.

After technology mapping, the resulting circuit requires a small bitfile that implements the logic for each pseudo-constant value. In the case of LUT primitives, this bitfile is simply the truth table stored in the LUT. We currently focus on offline bitfile creation as overcoming the complexity and overhead of online creation is beyond the scope of this paper. Offline creation is possible when a synthesis tool can pre-compute the bitfiles for all possible pseudo-constant values. At runtime, the circuit loads the correct bitfile upon a pseudo-constant invalidation. For a 4:1 mux with a pseudo-constant select, a synthesis tool could statically determine four separate bitfiles and store them in a block RAM. As another example, one input to a 32-bit comparator may only have two different possible values (e.g., runtime-specified thresholds), requiring only two separate bitfiles.

B. Pseudo-Constants Primitives for Xilinx Virtex 5

General-purpose logic resources in Xilinx Virtex 5 devices are composed of columns of configurable logic blocks

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

(CLB). Each CLB is composed of two SLICEs, each of which contains four LUTs (A, B, C, and D). Paired with each LUT is dedicated carry-chain logic and a flip-flop. Figure 1 shows the simplified functional architecture of the Virtex 5's six-input, two-output LUT.

1) Distributed LUT RAM

To implement the LUT RAM pseudo-constant primitive, we use Xilinx Distributed RAM. Each Xilinx LUT allows read and write access to the 64 SRAM bits in either 64x1-bit or 32x2-bit configurations. Multiple LUTs per slice can be grouped together to create wider or deeper memories. The write addresses for all four LUTs are driven by LUT D's six logic and read inputs, placing significant limitations on the efficiency of LUT RAM structures. For example, a dual-ported 64x1 RAM requires two LUTs (50% area penalty).

To achieve maximum area efficiency, a LUT RAM primitive using Virtex 5 distributed RAM should use all four LUTs in a single SLICE. Inputs **D[1:6]** drive the common write address and are used to configure LUTS A, B and C, which can then be used as three independent LUTs, while LUT D's inputs are consumed by serving as the write-address for LUTs A, B, and C. Using LUT RAM, each SLICE yields either three 6-input, 1-output functions, or three 5-input, 2-output functions. If inputs **D[1:6]** could be driven by both logic during normal operation and configuration hardware during reconfiguration, then four 6:1 or 5:2 functions could be realized per SLICE.

2) Shift Register

LUT shift-register primitives can be implemented using Xilinx SRL primitives. When configuring LUTs as shift registers, configuration bits for many LUTs can be shifted serially in a single configuration chain.

Using the SRL32, a single LUT can be configured as a five-input, one-output function. Configured as two SRL16s, each LUT can be configured as a four-input, two-output function. Unlike SRL32, each SRL16 must be driven by an independent configuration input; multiple SRL16 primitives cannot be combined to form a longer configuration chain.

C. Architectural Extensions

The pseudo-constant primitives for the Virtex 5, described above, show that the number of inputs and outputs to an FPGA's LUTs are a key limitation of pseudo-constant logic packing and place an upper bound on the achievable area reduction.

For example, in the design of an adder circuit, described in the next section, the key design limitation was the number of outputs from a LUT. While all inputs needed to produce up to five sum outputs and a carry could drive a single LUT, at most two outputs per LUT could be generated. The availability of only one set of fast carry logic and flip-flop per LUT limits the achievable maximum clock speed when using two outputs per LUT. Using LUT RAM primitives, one LUT per SLICE is consumed solely by the use of its

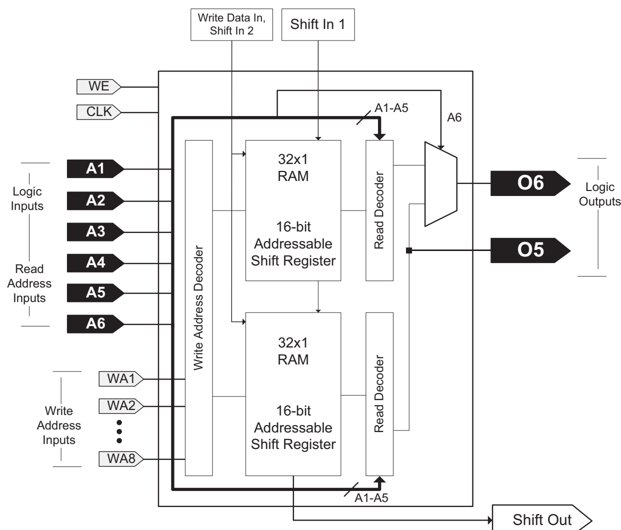

Figure 1: Functional architecture of a Xilinx Virtex 5 LUT. Each LUT can be configured as a 64x1 dual-ported RAM, a single variable-length shift register up to 32-bits long, or two independent variable-length shift registers up to 16-bits long each.

address pins by the RAM write address, and cannot be used for logic.

To improve amenability for pseudo-constant optimization, we also evaluate a hypothetical FPGA architecture that is augmented to improve efficiency of wider-output functions, such as those found in many arithmetic operations. Additional outputs per LUT and fast carry logic and flip-flop pairs for each LUT output could greatly improve the efficiency of wide-output functions. An extra set of address pins per SLICE to serve as the common write address input would prevent the 25 percent loss of functional density in LUT RAM based designs.

IV. Experiments

To evaluate pseudo-constant logic optimization, we manually technology mapped commonly replicated functions onto pseudo-constant primitives for Xilinx Virtex 5 FPGAs. Because Virtex 5, Virtex 6, and Virtex 7 devices all employ an identical CLB architecture, the results also apply to those devices. To determine benefits, we also synthesize each circuit without the proposed optimization to a Xilinx Virtex 5 LX50 FPGA using Xilinx ISE 14.2.

We evaluate the same circuits on a theoretical device incorporating the modifications proposed in Section III.C. This device's SLICEs are composed of four six-input, two-output LUTs identical to those of a Virtex 5. We added carry-logic and flip-flop stages to both outputs of each LUT. An additional set of common write-address inputs for LUT RAM primitives was also added. We assume Virtex 5 timing and switching characteristics [12]. Further design tradeoffs of such a device are outside the scope of this study.

A. Case Studies

In this section, we evaluate logic that is commonly replicated in large numbers by many FPGA applications,

including an adder, a comparator, and a multiplexer. Figure 2 summarizes the results.

1) 32-bit Full Adder

When synthesized into FPGA LUTs, adder circuits are output-bound. Synthesis in Xilinx ISE for a Virtex 5 uses the dedicated fast carry logic to create ripple-carry adders. Each LUT adds the ith bit of each input A and B, generating a sum and carry output to drive the carry logic, which combines these signals with C_{i-1} to generate a S_i and C_i.

If the add operation had one pseudo-constant input and one variable input, the pseudo-constant value can be folded into each full adder. Suppose three bits of the non-constant input, $[A_i, A_{i-2}]$, along with a carry input C_{i-3}, were connected to two LUTs. The four available outputs from this structure can then implement outputs $[S_i, S_{i-2}]$ and C_i. This structure allows the internal carry values to be calculated without consuming LUT outputs and implements a 3-bit adder using only two LUTs, yielding a 33% area savings. Using the SRL16-based four-input, two-output pseudo-constant LUT primitive, many such pseudo-constant 3-bit adders can be chained together. When synthesized using the pseudo-constant based design, a 32-bit adder consumes only 22 LUTs—an area savings of 31%. Because the Virtex 5 CLB's fast carry logic is accessible by only one output from each LUT, the optimized design cannot benefit from the fast carry logic. Despite a shorter overall combinational path, 11 logic stages rather than 32, the longer path between neighboring LUTs increases the circuit's combinational delay by 5x, from 2.515 ns for traditional logic to 10.377 ns using the pseudo-constant design.

When the pseudo-constant design is instead mapped onto the modified architecture from Section III.C, a 32-bit ripple carry adder can be mapped to the modified architecture using 22 LUTs with a combinational delay of 1.343 ns. This delay for the pseudo-constant-optimized adder is 47% faster than a traditionally synthesized adder.

2) Multiplexer

Using traditional synthesis methods, a four-input mux requires one LUT on a Virtex 5. Multiple four-input muxes can be combined using dedicated SLICE mux hardware to create up to one 16-input mux per SLICE.

If the select input to a mux were found to be pseudo-constant, using the SRL32 five-input, one-output LUT primitive, a five-input mux consumes only one LUT, and a 20-input mux can be created in each SLICE, yielding a 25 percent increase in functional density. Additionally, a four-input, two-output mux can be designed using the SRL16 four-input, two-output LUT primitive consuming only one LUT, yielding up to 50 percent LUT savings.

Using the LUT RAM-based primitive in the modified architectures, a six-input, one-output mux uses just one LUT, with up to a 24-input mux per SLICE. There is no difference in timing performance between each design.

Circuit	Method	Results	
Adder		LUTs	Delay(ns)
	Traditional	32	2.515
	PC Virtex 5	22	10.377
	PC Modified Arch	22	1.343
Comp		LUTs	Delay(ns)
	Traditional	11	4.658
	PC Virtex 5	8	6.556
	PC Modified Arch	6	4.783
Mux N:1		Max N	
		Per LUT	Per Slice
	Traditional	4	16
	PC Virtex 5	5	20
	PC Modified Arch	6	24

Figure 2: Comparison of LUT utilization for each evaluated circuit with traditional synthesis and pseudo-constants (PC).

3) 32-bit Comparator

Suppose a circuit must compare two 32-bit numbers, A and B. When synthesized to the Virtex 5 architecture, this circuit requires 11 LUTs, with a delay of 4.658 ns.

If input B was pseudo-constant, its value can be folded into the function implemented by the circuit's LUTs using the SRL32-based five-input, one-output LUT primitive described above. The inputs to each LUT are comprised of four consecutive bits of the variable input, along with a "carry-out" from the previous group. The outputs from these groups are cascaded together to create a 32-bit wide comparator using only 8 LUTs for an area savings of 27%, with an increased in propagation delay of 6.556 ns.

B. Functional Density

In [11], Wirthlin et al. present a functional density metric, D, defined as the inverse of the product of a circuit's area, A, and operating time, T. Additionally, [11] presents a specialized form of this metric for use with run-time reconfigurable circuits. By adding reconfiguration time, t_{config}, divided by operations per reconfiguration, n, to the operating time term, the metric accounts for the performance effects of reconfiguration at a given invalidation frequency.

Figure 3 plots functional density for each of the three adder circuits as the number of operations between invalidations (i.e., the inverse of invalidation frequency) increases logarithmically. This figure shows that while the combinational delay overhead on the Virtex 5 architecture prevents the pseudo-constant circuit from matching the functional density of the traditional adder, on the modified architecture the pseudo-constant circuit surpasses the functional density of the traditional adder after only 19 operations between reconfigurations. Reconfiguration overhead per operation reaches nearly zero after only 2^{14} operations, a small figure considering FPGA clock frequencies in the hundreds of megahertz. For infrequent invalidations, the functional density of the pseudo-constant adder on the modified architecture approached 2.7x.

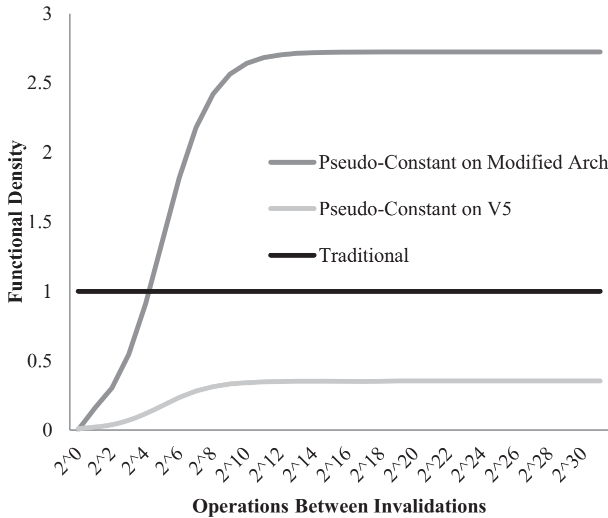

Figure 3: Functional density of a pseudo-constant adder compared to a traditional adder as the invalidation frequency increases. Results are shown for both the Virtex 5 and modified architectures.

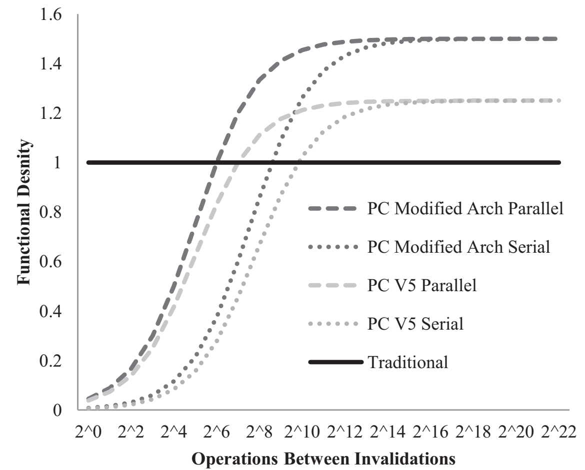

Figure 4: Functional density for each mux design is shown for both fully parallel and fully serial reconfiguration.

In any pseudo-constant design using LUT RAM or shift-register LUTs, reconfiguration can load the pseudo-constant bitfile into each LUT either in serial (one LUT at a time) or in parallel (all LUTs at once). Serial reconfiguration yields the largest performance penalty while parallel reconfiguration requires more reconfiguration resources. The degree of parallelism can be adjusted to find an appropriate Pareto-optimal design point for each design.

Figure 4 compares the functional density of each pseudo-constant 32-input mux to traditional muxes using either fully parallel or fully serial reconfiguration. The results show that pseudo-constant muxes approach a functional density of 1.25x on the Virtex 5 architecture, and 1.5x on the modified architecture, when compared to traditional synthesis. Additionally, the graph shows that the break-even point, at which functional density of the pseudo-constant optimized and traditional circuits are equal, is approximately 128 operations per invalidation using fully parallel reconfiguration, and fewer than 900 operations using fully serial reconfiguration.

V. Conclusions

In this paper, we showed that pseudo-constant optimizations can increase functional density of common logic structures. While initial results indicate up to 1.25x improvement in functional density on off-the-shelf FPGAs, the experiments also show the difficulty of implementing pseudo-constant designs on modern FPGAs. In particular, restrictions on dynamic reconfigurability and narrow-output functional units limit the effectiveness of pseudo-constant optimizations. If future FPGA designs address these concerns, pseudo-constant optimizations could be a viable method of increasing functional density in FPGA designs, with improvements as high as 2.7x density vs traditionally synthesized designs.

References

[1] Z. Baker, M. Gokhale, and J. Tripp. “Matched filter computation on FPGA, cell and GPU.” In IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 207-218, 2007.

[2] A. Brant and G. Lemieux. “ZUMA: An open FPGA overlay architecture.” In IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 93-96, 2012.

[3] S. Donthi and R. Haggard. “A survey of dynamically reconfigurable FPGAdevices.” In System Theory. Proceedings of the 35th Southeastern Symposium on, pp. 422-426, 2003.

[4] J. Eldredge and B. Hutchings. “Density enhancement of a neural network using FPGAs and run-time reconfiguration.” In FPGAs for Custom Computing Machines. Proceedings. IEEE Workshop on, pp. 180-188, 1994.

[5] P. Foulk. “Data-folding in SRAM configurable FPGAs.” In FPGAs for Custom Computing Machines. Proceedings. IEEE Workshop on, pp. 163-171, 1993.

[6] J. Fowers, G. Brown, P. Cooke, and G. Stitt. “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications.” In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 47-56, 2012.

[7] B. Gunther, G. Milne, and L. Narasimhan. “Assessing document relevance with run-time reconfigurable machines.” In FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on, pp. 10-17, 1996.

[8] A. Landy and G. Stitt. “A low-overhead interconnect architecture for virtual reconfigurable fabrics.” In Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems, pp. 111-120, 2012.

[9] E. Lemoine and D. Merceron. “Run time reconfiguration of FPGA for scanning genomic databases.” In FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on, pp. 90-98, 1995.

[10] E. McDonald. “Runtime FPGA partial reconfiguration.” In Aerospace Conference, IEEE, pp. 1-7, 2008.

[11] M. J. Wirthlin and B. L. Hutchings. “Improving functional density through run-time constant propagation.” In ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 86-92, 1997.

[12] Xilinx Virtex-5 FPGA Data Sheet: DC and Switching Characteristics, http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf