Don’t Forget Memories – A Case Study Redesigning a Pattern Counting ASIC Circuit for FPGAs

David Sheldon
Department of Computer Science and Engineering, UC Riverside
dsheldon@cs.ucr.edu

Frank Vahid
Department of Computer Science and Engineering, UC Riverside
Also with the Center for Embedded Computer Systems, UC Irvine
vahid@cs.ucr.edu

Abstract

Modern embedded compute platforms increasingly contain both microprocessors and field-programmable gate arrays (FPGAs). The FPGAs may implement accelerators or other circuits to speedup performance. Many such circuits have been previously designed for acceleration via application-specific integrated circuits (ASICs). Redesigning an ASIC circuit for FPGA implementation involves several challenges. We describe a case study that highlights a common challenge related to memories. The study involves converting a pattern counting circuit architecture, based on a pipelined binary tree and originally designed for ASIC implementation, into a circuit suitable for FPGAs. The original ASIC-oriented circuit, when mapped to a Spartan 3e FPGA, could process 10 million patterns per second and handle up to 4,096 patterns. The redesigned circuit could instead process 100 million patterns per second and handle up to 32,768 patterns, representing a 10x performance improvement and a 4x utilization improvement. The redesign involved partitioning large memories into smaller ones at the expense of redundant control logic. Through this and other case studies, design patterns may emerge that aid designers in redesigning ASIC circuits for FPGAs as well as in building new high-performance and efficient circuits for FPGAs.

Categories and Subject Descriptors
C.4 [**PERFORMANCE OF SYSTEMS**]: Design studies and Performance attributes

General Terms
Performance, Design.

Keywords
ASIC, FPGA, redesigning circuit, memory, BRAM, pattern counting, design patterns, high-throughput design, stream.

1. Introduction

Implementing applications as circuits on ASICs, and increasingly on FPGAs, is widely known to provide substantial speedups versus implementation on microprocessors for a wide variety of applications. However, designing circuit architectures for FPGAs involves some important differences from ASICs. One well-known difference involves the different off-chip/on-chip memory access time ratio, being large for ASICs, but often near 1 (or even less) for FPGAs, thus dramatically changing key architecture design criteria that typically involve going to great lengths to minimize off-chip memory accesses.

Another difference involves the pre-existence of block RAMs on FPGAs versus synthesizing custom-sized hard-core RAMs on ASICs. Such pre-existence, coupled with limited numbers of ports on block RAMs, suggests that circuit architectures for FPGAs should divide on-chip data in a more equal and distributed manner than for ASICs, to enable the best utilization of block RAMs as well as of distributed RAM (RAM implemented using FPGA configurable logic blocks). Yet another difference, related to the previous one, is the lack of placement freedom when using FPGA hard-core units like block RAMs or multipliers. In ASICs, RAM and multiplier cores can generally be placed near the components that use those cores. In FPGAs, however, RAM and multiplier cores have fixed placements. While mapping a circuit to FPGA typically involves placing components using cores, near to the cores the component is using, the distribution of those cores throughout the FPGA often makes such close placement impossible. Distant placement in turn results in long connections that must be routed across the FPGA, quickly consuming switch matrix capacity. Thus, routing from FPGA hard cores can quickly lead to a congestion problem that slows a circuit due to long routes, or that result in excessively long synthesis runs that may not complete due to the difficulty or inability to route the circuit.

We encountered the above problems in a project that involved mapping a previously-designed ASIC pattern counting circuit to an FPGA. While the circuit worked superbly as an ASIC implementation, the circuit could not be scaled to handle large numbers of patterns on an FPGA – the block RAM resources were quickly consumed, and the circuit’s performance slowed dramatically as sizes were increased to desired quantities, with the circuit eventually failing to map.

Much work has been done on technology mapping problems specific to FPGAs as opposed to ASICs, e.g., [3][4][6][10][14][15]. Beraudo [3] replicates parts of circuits to improve performance – our approach also involves a form of replication, but at a higher-level.

Work has also been done on creating custom computing circuits for FPGAs, often with knowledge of the FPGA’s

CODES/ISSS’08, October, 2008, Atlanta, GA, USA.
Copyright 2008 ACM 1-58113-000-0/00/0004…$5.00.

physical resources. Metzgen [12] constructed a high-performance ALU for the NIOS processor. Patterson [13] created a DES encryption block targeted to particular FPGAs. Numerous other circuits for FPGAs have been developed, e.g., [2][7][8][9]. Many of these circuits target a specific FPGA device, using knowledge of available physical resources when creating the circuit.

However, there has been little work on how to redesign existing ASIC circuits to FPGAs. Most research has focused on creating new circuits for FPGAs, with little work done in trying to understand the differences between designing for ASICs versus FPGAs.

In this paper, we provide a case study describing our efforts to redesign a pattern counting circuit for fast, efficient FPGA implementation. The contributions are twofold. First, we describe a fast and efficient pattern counting circuit that can process 100 million patterns per second and that scales well to tens of thousands of target patterns. Counting patterns is a fundamental computing problem with a wide variety of applications, including networking, computer profiling, bioinformatics, and more, for which FPGAs provide outstanding speedups over microprocessors. Second, we describe how we improved performance and utilization by redesigning an ASIC circuit for an FPGA, which, along with future or other case studies, may lead to design patterns [5] to help guide circuit designers who target FPGAs.

2. Pattern Counting and the ASIC-Oriented Pipelined Binary Tree Circuit

Lysecky [11] introduced a high-throughput circuit for the pattern counting problem. The problem involves a bus over which unique bit patterns may appear. One example of such a bus is the address bus between a microprocessor and a memory, as shown in Figure 1. Given a set of pre-specified patterns of interest, known as *target patterns*, the problem is to count the number of times each target pattern appears on the bus. A goal is to have the pattern counter support the highest throughput possible. Possible applications of pattern counting include accurate profiling of an executing program, accurate profiling of network traffic in a network router, tallying huge databases of business transactions (e.g., counting the number of each item sold), counting word frequencies in large numbers of phone conversations, counting occurrences of particular sequences in biological data, and much more.

Lysecky's solution is based on a pipelined binary tree, illustrated in Figure 2. The target patterns are stored in the tree in breadth-first order. Thus, the first level (root) contains only one pattern, the second level contains two patterns, the third contains four patterns, the fourth contains eight patterns, and so on. Each level consists of control logic and a memory to store the patterns, and another memory of the same size (not shown in the figure) to maintain pattern counts. Each level operates concurrently, taking information from the previous level, and sending information to the next level.

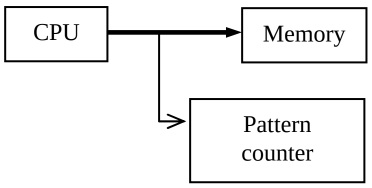

Figure 1: Example pattern counting scenario: Counting occurrences of an address on a CPU bus.

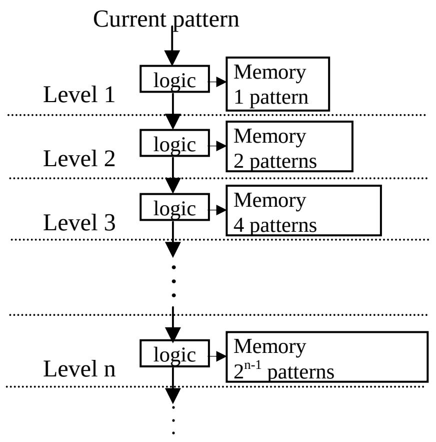

Figure 2: Pattern counting with a pipelined binary tree. Each level operates concurrently, taking the pattern and address information from the previous level, and passing information to the next level.

Level 1 receives the current pattern and compares with the target pattern. If equal, level 1's logic increments the count associated with that target pattern. If less, the logic passes the pattern to level 2, informing level 2 to look in its left node (because in a binary tree, if the pattern is less than the root, then search proceeds down the left subtree) – in particular, by telling level 2 to look at address 0. If greater, level 1 tells level 2 to look in address 1. Level 2 then compares the pattern with the target pattern located in the address it received from level 1 (while level 1 meanwhile processes the next incoming pattern). If equal, level 2's logic increments the count associated with that target pattern. If less, level 2 appends a 0 to the address, so if the address was 0, the new address is 00; if it was 1, the new address is 10. If greater, level 2 appends a 1 to the address, yielding either 01 or 11. Subsequent levels operate similarly, either incrementing their count, or appending 0 or 1 to the address as they pass the address to the next level.

The pipelined binary tree achieves single-clock-cycle throughput. The cycle length is mostly due to memory access. Wires between levels can be extremely compact using simple folding approaches that abut each level with the next. The original design, in UMC's 0.18 technology, achieved GHz frequencies, and hence billion-patterns-per-second throughput. Size is efficient due to only minimal logic being required per level (an adder, a comparator, and a few gates), which is dwarfed by the memory size for large target pattern sets.

In seeking to perform pattern counting on a Xilinx Spartan 3e 1600 FPGA using Xilinx ISE tools [16], we used Lysecky's binary tree circuit by coding it in structural VHDL. However, we found that the circuit, while working superbly for its target device of ASICs, failed to work as well on FPGAs. Figure 3 shows that the clock frequency of the original binary tree design drops precipitously as the number of target patterns (i.e., binary tree size) is increased above 512, from nearly 100 MHz for the smaller trees, to below 10 MHz for the 4,096 pattern tree. The result is a large throughput decrease shown Figure 4 (note that the Y-axis is a log scale). The large drops are likely due to routing congestion caused by trying to connect the larger levels'

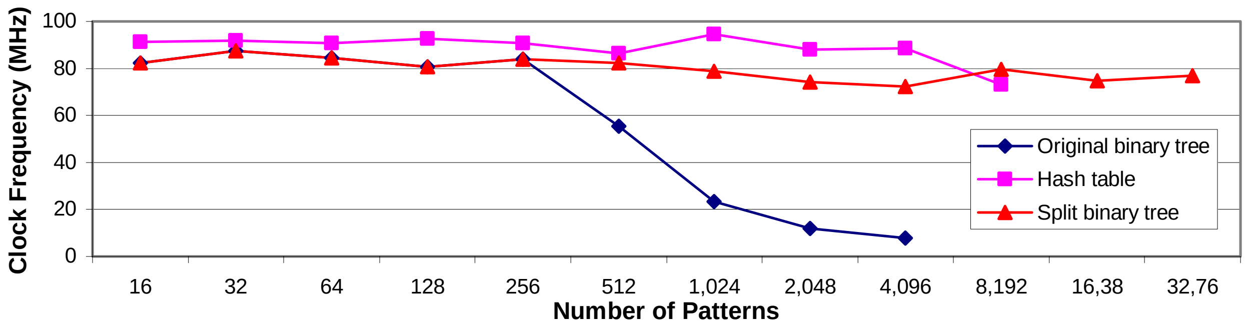

Figure 3: Clock frequency as a function of number of patterns, for the three different designs considered.

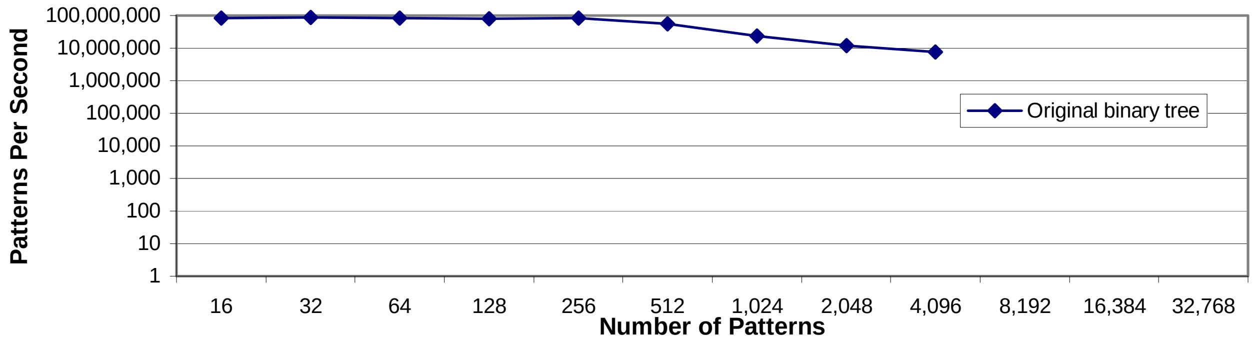

Figure 4: Throughput for the original binary tree. Note that Y-axis is a log scale, so throughput reduction is more than 10x.

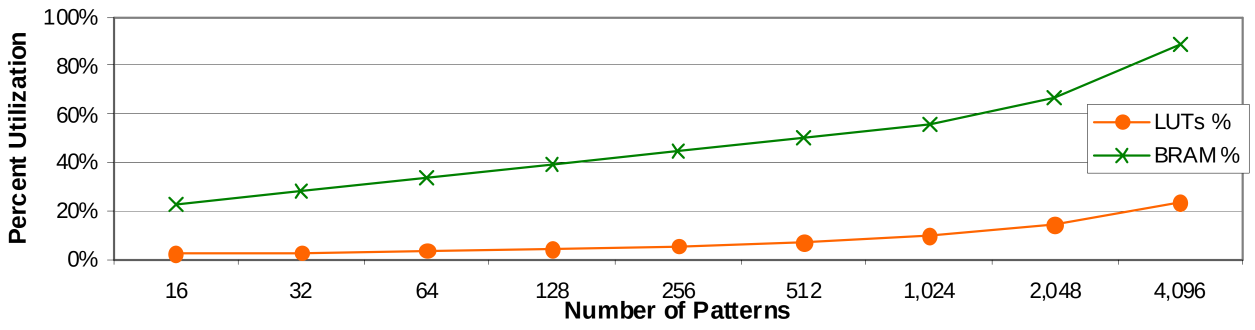

Figure 5: Original pipelined binary tree's block RAM and LUT utilizations.

logic to multiple block RAMs (each Spartan block RAM can hold 512 patterns). Beyond 4,096 patterns, the circuits failed to map to the FPGA. Figure 5 provides some insight into why, showing that the 4,096 pattern design utilizes nearly all the available block RAMs, but leaves most LUTs (lookup tables) unutilized.

In the above design, every level uses a block RAM. Thus, there is underutilization within the block RAMs themselves for all levels smaller than 512 (levels 1 through 8), which is the size of a block RAM. We tried to have the smaller levels instead use distributed RAM, but doing so decreased the clock cycle by 50% or more.

3. Hash Table Approach for Pattern Counting

As the pipelined binary tree failed to synthesize well to FPGAs, we investigated an entirely different approach for pattern counting on FPGAs. We created a new design implementing a hash table. We implemented a custom hashing function to convert a pattern to an address via simple bit selection. In order to get a best-case analysis for the hash table, we used a perfect hash in our experiments, meaning the patterns used for each experimental run had a one-to-one mapping into the hash table. Upon finding a match, the corresponding count is incremented.

This simple circuit is comprised mainly of a memory equal to the size of the number of patterns, with very little logic required for the hash function, incrementing, and conflict logic. While the circuit's clock frequency is high, as shown in Figure 3, Figure 6 shows that this circuit does not achieve the same throughput as the pipelined binary tree, starting below 10 million patterns per second. Furthermore, the circuit failed to map for more than 8,192 patterns, due to block RAMs being consumed, as shown in Figure 7. The figure shows that block RAMs are exhausted at 8,192 patterns while LUTs are almost entirely unutilized.

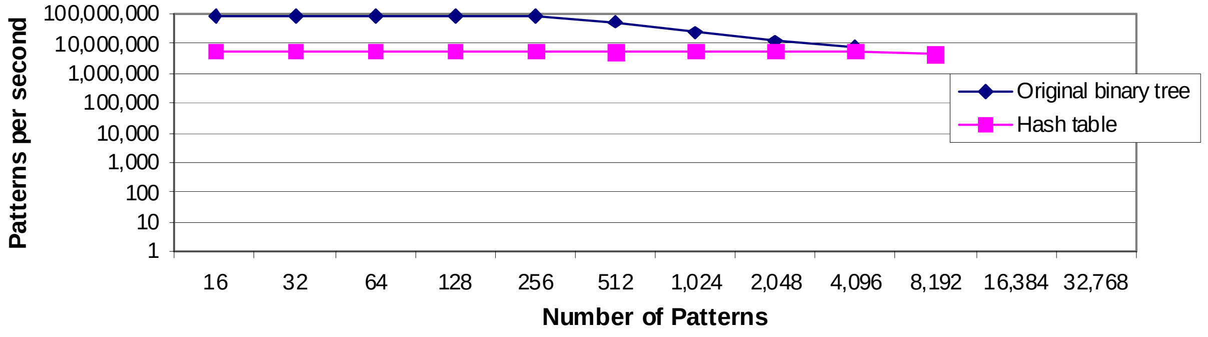

Figure 6: Throughput for a circuit hash table implementation.

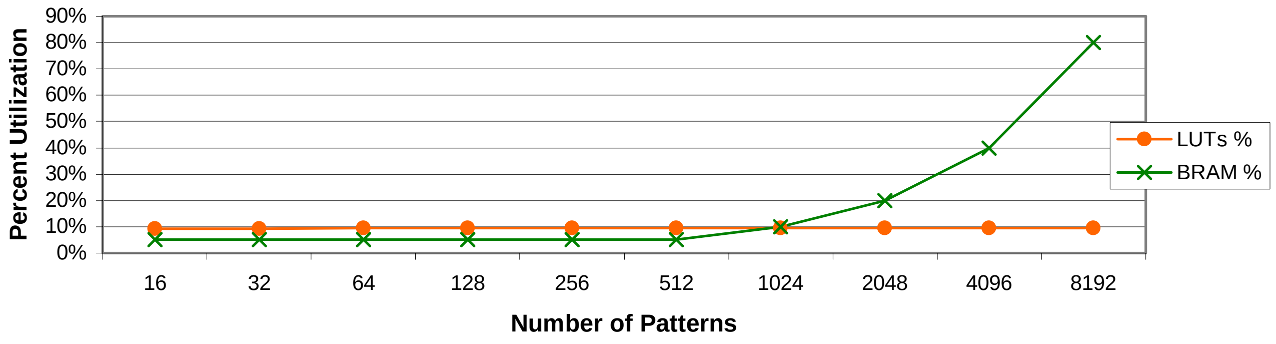

Figure 7: Hash table circuit's block RAM and LUT utilizations.

4. Split Pipelined Binary Tree for Pattern Counting

We thus re-examined the binary tree approach to determine if the tree structure could have better scaling and efficiency on FPGAs. We noted that the large memories seemed to be the source of the FPGA problem in both the original binary tree and the hash table circuits, causing clock frequency reductions in the former, and block RAM exhaustion in both. We thus sought to reduce the size of the largest single memory needed in the circuit.

In the original binary tree circuit, each level required a memory twice the size of the previous level. Beyond a size of 512, which is the size of a block RAM on the FPGA we were using, performance dropped for the binary tree approach.

In the original binary tree circuit, after 512 patterns, the performance begins to slow. We concluded that this drop in performance was largely due to increased wire lengths for a level's logic to access the level's block RAMs, e.g., the logic for a level with 2,048 patterns would be connected to 4 BRAMs. In the binary tree circuit, the logic associated with each level is small, while the BRAMs are spread over the entire FPGA. The overall performance is further slowed by the fact that BRAMs must have single cycle access to the data.

Our solution was to divide any level with a memory larger than 512 into sub-circuits consisting of logic having sub-memories of 512 each. To avoid the situation of one block of logic connecting to multiple BRAMs, we replicated the logic at each level for each sub-memory, forming a sub-module. Each sub-module connects with the appropriate two sub-modules of the next level. Figure 8 shows the sub-module structure that is

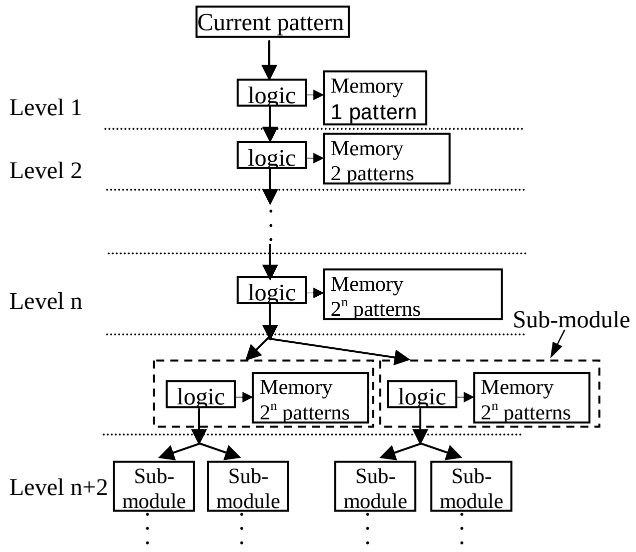

Figure 8: Pattern counting with a pipelined split binary tree. Each level operates concurrently, taking the pattern and address information from the previous level, and passing information to the next level. Levels after n are composed of multiple logic and memory blocks. In our experiments, n=8.

created in this *split* binary tree. For each additional level added to the tree, each sub-module will be connected to 2 sub-modules, each of 512 patterns.

The sub-module solution maintains the simple connectivity among levels and the fully-pipelined nature of the design. This structure also has the added benefit of easing the routing of the tools. Once a pattern has been passed to a sub-tree, the pattern will stay in that sub-tree. This means that no communication is needed between sub-modules in the same level. The only communication required is with the parent module in the previous level and the two children in the next level. This basic structure also gives the tool much greater flexibility in how to best to layout the circuit on the FPGA.

The solution's drawbacks include an increase in area due to redundant logic in each level, equaling approximately 15 slices per BRAM, and more power due to multiple sub-modules being active simultaneously in each level. We found neither drawback to be significant. Neither LUT utilization nor power was an issue in the earlier design, and thus those factors could tolerate increases without problem.

Figure 9 shows throughput results for the split pipelined binary tree, compared with the previous two approaches. The split binary tree maintains a nearly constant throughput beyond 512 patterns, to the maximum size that we experimented with, which was 32,786 patterns. Actually, the split binary tree experiences a 10% drop in throughput, which is not noticeable in the figure.

Figure 10 shows block RAM and LUT utilizations. We noticed a surprising *decrease* in block RAM utilization above 1,024 patterns. Upon investigation of the synthesis script outputs, we determined that this decrease was due to the synthesis tool making use of distributed RAM. We believe the use of distributed RAM (i.e., using CLBs for memory rather than block RAMs) was enabled by the smaller maximum memory size, which helped the synthesis tool place logic and memory near each other and thus to find acceptable routing solutions, even when using distributed RAM. It is not clear to us at this time why the tool could put the larger memories into distributed RAM without significant clock frequency reduction, but could not do so when we tried to use distributed RAM for the smaller memories of the original binary tree. However, of the different types implementations that we examined, the split-binary tree was the only implementation for which the synthesis tool used distributed memory. In all the tests, we directed the synthesis tool to attempt to use the BRAMs.

The largest tree we synthesized for the target Spartan device could hold 32,768 target patterns. Based on throughput and utilization, more patterns could likely have been successfully mapped (likely up to 65,536). However, as seen in Figure 11, the synthesis runtime for the 32,768 pattern circuit was 8 hours, which was the longest runtime we considered for this study.

Thus, splitting the memory into smaller devices enables more efficient synthesis, stemming largely from the synthesis requirement that a logical memory have single cycle access even when implemented on multiple BRAMs.

5. Other Memory Configurations

Ideally, one circuit could be automatically mapped to different FPGAs, but as seen in the extent of the redesign involved in earlier sections, such automation could be challenging. We have shown how to map a high-throughput pattern counter to the memory implementation architecture of a Xilinx Spartan FPGA. However, Altera's FPGAs use a different memory architecture.

Xilinx Virtex and Spartan FPGAs contain on-chip memory blocks (BRAMs) that are all the same size on a single device (though the sizes may vary between different devices). In contrast, Altera Stratix FPGAs [1] have on-chip memory (TriMatrix Memory) that is divided into three different types. MLAB is the smallest of the types with 640 bits per block, with up to 6,750 such blocks available on a device. M9K is a 9 kilobits memory block with up to 1,040 such blocks on a device.

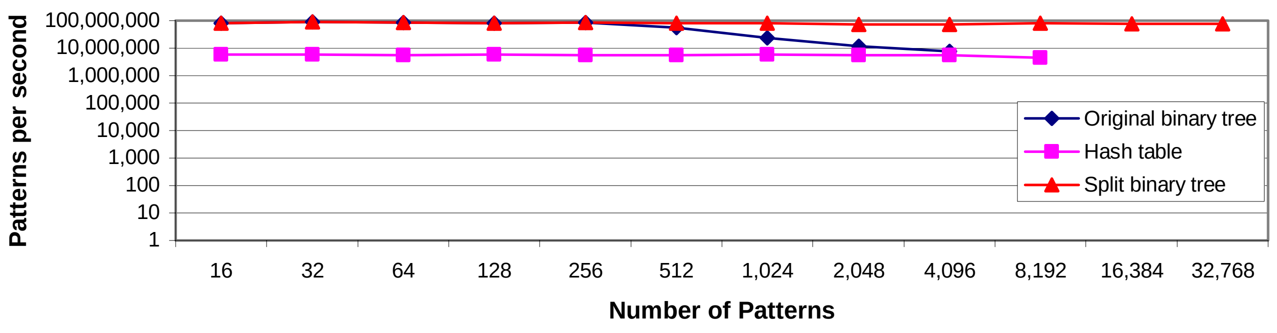

Figure 9: Throughput for the split pipelined binary tree.

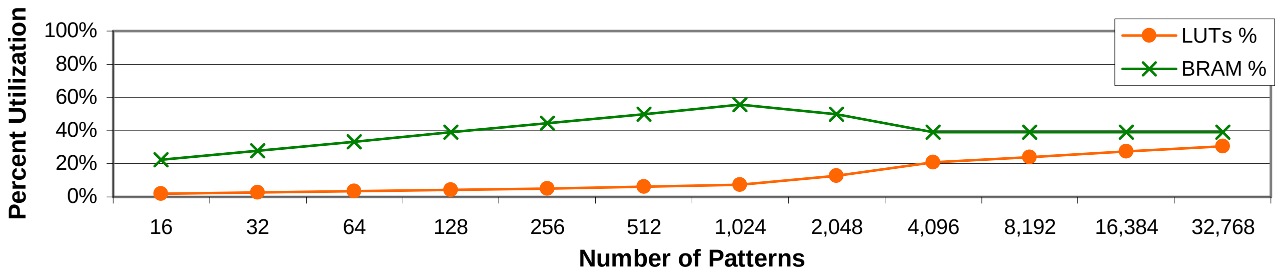

Figure 10: Split binary tree's utilizations.

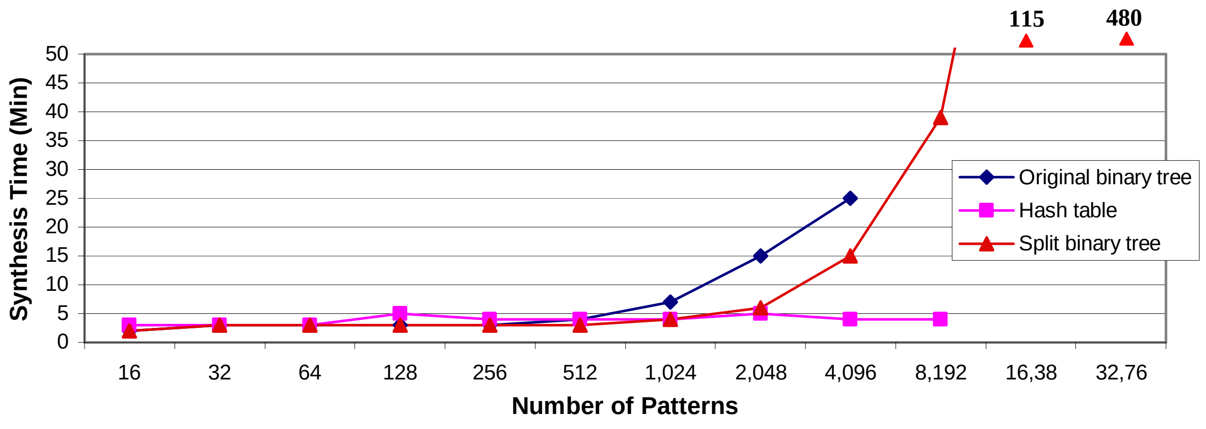

Figure 11: Synthesis times of the three designs.

The largest is the M144K block, which holds 144 kilobits and with up to 48 such blocks on a device.

Thus, if using an Altera device for the pattern counter circuit, less splitting of memories may be necessary than when using a Xilinx device. The levels closer to the root in the circuit's tree might use the smaller Altera memories, obviating the need for splitting of the circuit's memories and introducing the redundant control logic to larger tree levels. Therefore, the best circuit design can vary quite significantly depending on what the resources that a particular device has.

6. Conclusions

We described a high-throughput pattern counter targeted to an FPGA. The pattern counter used a pipelined binary tree architecture that was previously developed for ASICs, but which exhibited severe throughput reductions for larger trees, and which failed to map beyond 4,096 patterns. We determined the problem to be related to use of large memories, and redesigned the architecture such that the maximum memory size was 512. The redesign required replicating logic, but logic was not the constraining factor in the design, and increases in logic sizes were negligible, approximately 15 slices per memory.

This case study provides one example of how to design or redesign a circuit to account for specific FPGA-related issues, in this case the issue of memory size. Along with other case studies, design patterns may emerge to help guide designers who target FPGAs. We also show that the different FPGA designs will probably require different mapping strategies. An interesting avenue of future work is to develop methods to easily and effectively port circuits to different FPGAs the way that standard microprocessor binaries are easily ported to different microprocessors.

7. Acknowledgements

This research was supported in part by the National Science Foundation (CNS-0614957).

References

[1] Altera Corporation. www.altera.com, 2008

[2] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. Reconfigurable Molecular Dynamics Simulator, IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2004.

[3] Beraudo, B., J. Lillis. Timing Optimization of FPGA Placements by Logic Replication. Design Automation Conference (DAC), 2003.

[4] Cong J., Y. Hwang. Simultaneous Depth and Area Minimization in LUT-based FPGA Mapping. Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays (FPGA), 1995.

[5] DeHon A., J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design Patterns for Reconfigurable Computing. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2004.

[6] Eguro K., S. Hauck. Armada: Timing-Driven Pipeline-Aware Routing for FGPAs. Int. ACM Symp. on Field Programmable Gate Arrays (FPGA), 2006.

[7] He, C., Lu, M., and Sun, C. Accelerating seismic migration using FPGA-based coprocessor platform. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 207-216.

[8] Huang, Z. and Ercegovac, M. D. 2001. FPGA Implementation of Pipelined On-Line Scheme for 3-D Vector Normalization. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2001.

[9] Krueger, S. D. and Seidel, P. Design of an on-line IEEE floating-point addition unit for FPGAs. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 239-246.

[10] Li H., W. Mak, S. Katkoori. LUT-Based FPGA Technology Mapping for Power Minization with Optimal Depth. Proceedings of the Int. ACM Symposium on Field-Programmable Gate Arrays (FPGA), 2001.

[11] Lysecky R., S. Cotterell, F. Vahid. A Fast On-Chip Profiler Memory. IEEE/ACM Design Automation Conference (DAC), 2002, pp. 28-33.

[12] Metzgen P. A High Performance 32-bit ALU for Programmable Logic. Int. Symp. on Field Programmable Gate Arrays (FPGA), 2004.

[13] Patterson C. High Performance DES Encryption in Virtex FPGAs using Jbits. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), 2000.

[14] Singh A., M Marek-Sadowska. Efficient Circuit Clustering for Area and Power Reduction in FPGAs. Int. Symp. on Field Programmable Gate Arrays (FPGA), 2002.

[15] Singh D., S. Brown. Integrated Retiming and Placement for Field Programmable Gate Arrays. Int. Symp. on Field Programmable Gate Arrays (FPGA), 2002

[16] Xilinx, Inc. Spartan 3e 1600. http://www.xilinx.com, 2008.