A Parameterized Automatic Cache Generator for FPGAs

Peter Yiannacouras and Jonathan Rose
Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada M5S 3G4
{yiannac@eecg.utoronto.ca, jayar@eecg.utoronto.ca}

Abstract

Caches in FPGAs can improve the performance of soft processors and other applications beset by slow storage components. In this paper we present a cache generator which can produce caches with a variety of associativities, latencies, and dimensions. This tool allows system designers to effortlessly create, and investigate different caches in order to better meet the needs of their target system. The effect of these three parameters on the area and speed of the caches is also examined and we show that the designs can meet a wide range of specifications and are in general fast and compact.

1. Introduction

Choosing the best cache for a system is very application-specific, but is often done subjectively. A particular cache is often selected based on general notions of its area, speed, and effectiveness inferred from the many studies performed on caches. This leads to a less than optimal cache solution. In order to more aggressively pursue good choices for cache parameters, a more precise understanding of a cache's area, speed, and effectiveness is required. Our goal is to facilitate such a pursuit by providing an automatic cache generator for FPGAs which, given a set of input parameters, will output an efficient cache implementation satisfying the given parameters. The generator is versatile as it allows for a number of different cache types to be generated. Users can select the appropriate size, latency, and associativity for their desired cache and use the proposed generator to create an implementation in Verilog. While The implementations are targeted for Altera's Stratix family of FPGAs, the code could be modified to emit primitives specific to other families of FPGAs.

The generator can be used to easily implement a desired cache, quickly evaluate performance for a number of caches, or to compile a catalogue of statistics for several cache variants. We will present statistics for the area and speed of caches with different associativities, latencies, and dimensions in an FPGA environment. Doing so will perhaps pave the way for a more exact and deterministic process for selecting cache parameters.

The remainder of this paper is organized as follows: Section 2 describes the Stratix FPGA architecture and necessary Cache nomenclature; Section 3 describes the various cache designs that can be generated; Section 4 examines the resource utilization and speed performance for caches of various types and sizes; Section 5 provides a link to the software developed; and Section 6 concludes.

2. Background

In this section we provide relevant background on the FPGA we use, our cache nomenclature and other working parts of our system. **The Stratix FPGA** from Altera is built in a 0.13um technology, and contains a heterogenous mixture of programmable logic, memory, and arithmetic blocks [1][4]. Its memory blocks come in three different sizes: 512 bits, 4 kilobits, and 512 kilobits (not including parity bits). These memory blocks are named M512, M4K, and MRAM (Mega RAM) according to their size and will be referred to as such throughout this document. The Stratix Logic Element (LE) consists of a 4-input lookup table and flip-flop, with the ability to implement 1 bit of add/subtract arithmetic in each cell.

The speed of a system is defined as its maximum operational clock frequency. This figure is reported by CAD tools, in our case Altera's Quartus II version 2.1 CAD package, which was used for all levels of synthesis.

A cache is a memory that stores a small subset of the data available in a processor's address space. If currently addressed data is in the cache, a *hit* is said to have occurred and the cache can satisfy the memory operation without involving the slower memory. Data values in the cache are identified using a *tag*. A tag is the subsection of the address required to uniquely identify the data. Tags are each stored in a tag store alongside the corresponding data stored in a data store.

There are three dimensions which define the size of a cache. These are its cache line, cache depth, and tag width. A *cache line* refers to the unit of data storage, in bits, used in the cache. The maximum number of cache lines that can be stored in the cache is known as the *cache depth*. The *tag width* is the number of bits in a tag.

Ideally, new data can be added to the cache as long as the cache has an unoccupied cache line available. This implies that a given data can map to any cache line. Such

a cache is known as a *fully associative cache*, or just associative cache. A more simplified cache such as the *direct-mapped* cache maps data to only one cache line determined by the low order bits of its address. Between these two extremes is the *set associative* cache. Instead of using the low order bits to select a single cache line, they are used to select n different cache lines. Such a cache is said to be *n-way* set associative. These caches generally provide the best compromise between circuit complexity and performance.

The strategy used to choose which data to evict from the cache is known as the r*eplacement policy*. Eviction occurs when space needs to be made for new data. The most common method, known as *LRU* (Least Recently Used), tracks how recently each piece of data was referenced, and evicts the one used furthest in the past.

A write operation can potentially cause the cache and memory to become unsynchronized if the cache contains a more recent value than the memory, and that value is evicted without being written to memory. In this design this is prevented by employing a *write-through policy*, which ensures the synchronization of the cache and memory by always writing to both.

A CAM (Content Addressable Memory) is the inverse of RAM. While a RAM is given an address and outputs the data stored at that address, a CAM receives data, often called a *pattern*, and returns the address where it is stored, or indicates that the pattern is not currently in the CAM. This makes CAMs ideal for searching through tags and detecting cache hits in associative caches.

3. Cache Design

Table 1 lists the set of characteristics that can be specified to our parameterized cache generator.

	Fully Associative	Direct-mapped	Two-way Set Assoc.
Read Latency	2,3	1,2	1,2
Write Latency	1	1	1,2
Depth	any	any	any
Addr Width	any	any	any
Data Width	any	any	any

Table 1. Characteristics of Each Cache Type

3.1 Fully Associative Cache

The design of a fully associative cache involves five components: A CAM, an encoder, a data store, a tag store, and a counter. Figure 1 illustrates a read operation.

A simple logical OR of the decoded CAM outputs is used to detect a hit. An encoder is used to generate the encoded address needed by the data store. The encoder is the largest and often slowest block of logic in this cache. Its size is determined solely by the depth of the cache but is minimized in this design by capitalizing on the CAM's one-hot decoded outputs.

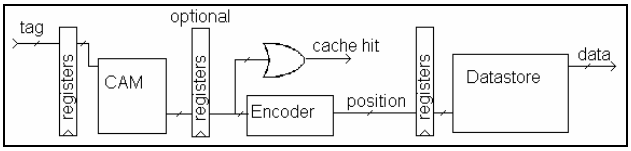

Figure 1. Associative Cache Read

A cached read completes in either two or three cycles after being issued, depending on the choice of the user. Writes require one cycle latency and use a counter-based replacement policy as was used in [2]. The CAM implementation is identical to that in [3]; the only modification is in the chosen dimensions of the CAM blocks which were customized to fit Stratix's M512s. Doing so required one fifth the memory [3] needed.

3.2 Direct-Mapped Cache

The direct-mapped cache has a simple design. It requires only a tag store, data store, and comparator, where the tag and data stores are both single-port RAM blocks. Since only one tag in the tag store can match the tag input, that tag is the only one read, and the only one that need be compared. Thus a single comparator is used to detect a hit. A schematic of the read circuitry is shown in Figure 2. A write operation involves simply writing the tag to the tagstore and the data to the data store overwriting any values already there.

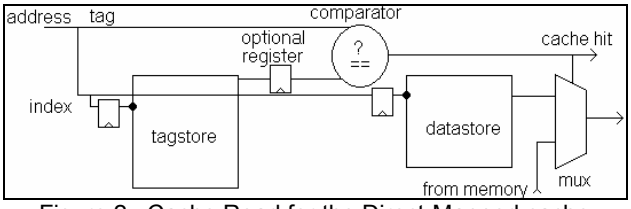

Figure 2. Cache Read for the Direct-Mapped cache

3.3 Two-Way Set Associative Cache

A two-way set associative cache has two cache lines in each set. Reading from the cache requires comparing tags from both cache lines and selecting which (if any) of the data to return. Figure 3 shows the implementation of such a system which is, in effect, two direct-mapped caches arranged in parallel.

The write operation is also identical to the direct-mapped cache except that a true LRU replacement policy is used to specify which of the two cache lines in a set are to be overwritten. The LRU circuitry is comprised of a single one bit-wide RAM block which indicates which of the two cache lines is to be evicted.

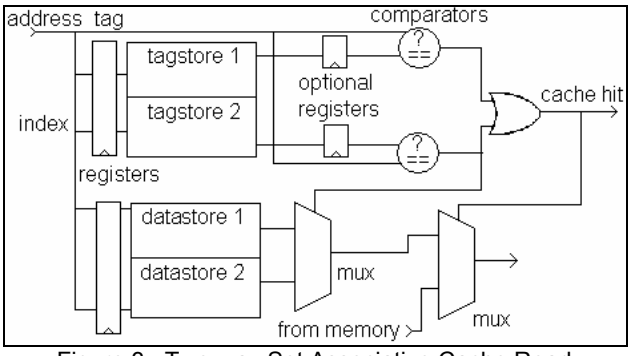

Figure 3. Two-way Set Associative Cache Read

4. Speed and Area Measurements

In this section we compare the different types of caches with respect to their maximum operating frequency, and area in terms of logic and memory usage. All measurements are made using a 32-word deep cache with a 32-bit address space and 32-bit data width as a base and varying only one of the dimensions.

We measure both logic (in terms of number of Stratix Logic Elements, or LEs) and memory (M512s, M4Ks, and Mega RAMs) consumption for the different caches. The effect of increased cache depth, address width, and data width will be examined for all cache types. Changes in latency were observed to have a negligible impact on the area of the circuits. Thus, the area measurements were made only for the different amounts of associativity.

For increases in cache depth, the number of Stratix LEs used for the direct-mapped and two-way set associative caches was fixed at 99 and 198 respectively. This independence is expected since those designs have no logic components which depend on the depth. Moreover, the number of logic elements is seen to be very small, respectively using 1% and 2% of the smallest Stratix chip, the S10. In the associative case, the increase in LE usage was linear with slope 3.5 LEs/word and zero offset. The contributing factors were the CAM, encoder, LRU circuitry, and wide logical OR, all of which grew with increased cache depth.

Figure 4 shows that all cache types suffer linear increases in LE usage with increases in address width. The direct-mapped and two-way cache designs both exhibit increases because of the comparators which need to compare larger tags. In the associative cache, the increase is due to the increasing size of the CAM. The two-way cache has two comparators while the direct mapped cache has one; thus, it increases at twice the rate. Still, LE usage is rather modest for all the caches.

The effect of increasing data width on the number of LEs used is also seen in Figure 4. This increase comes solely from growth in the data multiplexers. The two-way set associative cache contains three data muxes, whereas the direct and associative require only two. Thus it increases 150% faster than the direct and associative caches. Again these increases are rather moderate.

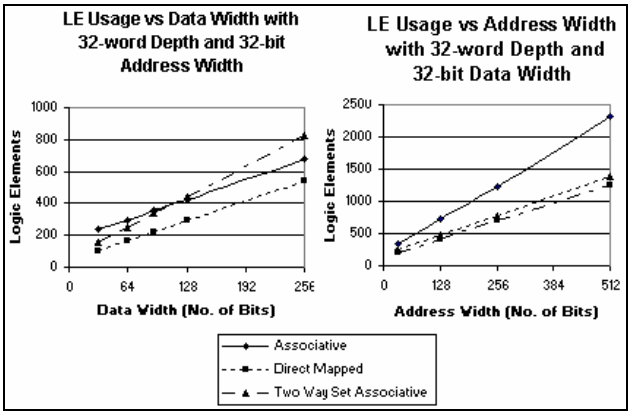

Figure 4. LE usage vs. Address Width and Data Width

Table 2. Memories used for different Cache Depths

	Number of RAM Blocks Used								
Depth	Associative			Direct Mapped			Two Way Set Ass.		
(words)	512	4K	Mega	512	4K	Mega	512	4K	Mega
32	14	2	0	0	2	0	1	4	0
128	56	2	0	0	2	0	1	4	0
256	112	4	0	0	4	0	1	4	0
1024	448	16	0	0	14	0	1	14	0
2048	NA	NA	NA	0	27	0	0	29	0
4096	NA	NA	NA	0	0	1	0	55	0

Table 3. Memories used for different Address Widths

Addr.	Number of RAM Blocks Used								
Width	Associative			Direct Mapped			Two Way Set Ass.		
(bits)	512	4K	Mega	512	4K	Mega	512	4K	Mega
32	14	2	0	0	2	0	1	4	0
128	53	4	0	1	4	0	3	8	0
256	105	8	0	0	8	0	3	16	0
512	207	15	0	0	15	0	3	30	0

Table 4. Memories used for different Data Width

Data	Number of RAM Blocks Used								
Width	Associative			Direct Mapped			Two Way Set Ass.		
(bits)	512	4K	Mega	512	4K	Mega	512	4K	Mega
32	14	2	0	0	2	0	1	4	0
64	14	3	0	0	3	0	1	6	0
92	14	4	0	1	3	0	1	8	0
128	14	5	0	1	4	0	1	10	0
256	15	8	0	0	8	0	1	17	0

While all the designs had relatively modest LE requirements, the opposite is seen in their memory requirements. The number of Stratix M512s, M4Ks, and Mega RAMs required for each cache design is shown in Tables 2, 3 and 4. Again the latencies are ignored since this did not impact the number of memories used. Note that the Altera software system, Quartus II, selects the

target physical memories used (except in the case where we explicitly used the small memories for the CAMs).

The growth of the M512s used for the CAMs in the associative cache is linear as expected from [3]. Changes in the data width have no effect on the CAM as can be seen in Table 4 column 8. An interesting observation occurs for the direct-mapped cache with depth of 4096 words. The Quartus II compiler merged the tag and data store into a single Mega RAM block. This technique reduces the number of memory blocks and simplifies routing which may speed up the circuit.

We now present the effect of different cache parameters and types on the post-routed speed of the cache. Data was gathered for all cache types including different latencies since latency plays a pivotal role in the system's overall speed. Figure 5 provides a graph generated from the measured data from increasing cache depth. For data and address width graphs see [5].

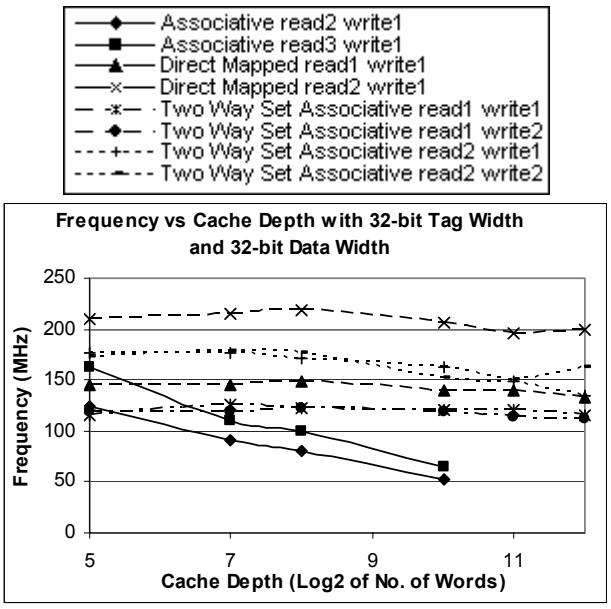

Figure 5. Graph of Frequency vs. Cache Depth

As expected, we noticed the direct-mapped cache is generally fastest and the associative cache is slowest. It is interesting to note that an associative cache with a three cycle read latency can outperform a single cycle read latency direct-mapped cache. Also, the direct-mapped cache which was merged into a MegaRAM performs faster than the direct-mapped cache half its size.

A summary of the effects of increasing each cache dimension on the speed of the cache is show in Table 5. These results follow intuitively from growth in their respective components as it is mostly larger CAMs and comparators that slow down the caches. Address widths affect both, data widths change neither, and depth influences only CAMs.

Table 5. Speed Reductions with Growth in Dimension

	Depth	Addr	Data
Assoc.	Yes	Yes	No
Direct	No	Yes	No
Two Way	No	Yes	No

5. Software

Software and source code for the cache generator is at: http://www.eecg.toronto.edu/~jayar/software/cachegen/cachegen.html. It consists of C-language code which generates parameterized Verilog.

6. Conclusion

We have presented an automatic parameterized cache generator that emits cache designs for an FPGA in a Verilog output file. The input to the generator is a set of parameters describing the desired cache, including associativity, latency, cache depth, address width, and data width. Cache designs were generated and evaluated for a wide variety of input parameters. From this analysis, a number of trends were established concerning their area and speed. The generator is very robust offering a wide variety of cache types, able to satisfy a wide variety of size and speed constraints. With this tool, and the presented statistics, a designer can identify cache parameters which satisfy the area and speed constraints of a system, and then choose a cache with optimal effectiveness. Progress in this direction will eventually yield a more precise method of selecting cache types for soft processors and other cache applications.

References

[1] Altera Corporation. "Altera Stratix FPGA Family Data Sheet, " December 2002. http://www.altera.com/

[2] Altera Corporation, "AN 119: Implementing High-Speed Search Applications with Altera CAM," in Altera Application Notes, July 2001. http://www.altera.com/

[3] J.L. Brelet. "Using Block RAM for High Performance Read/Write CAMs" in Xilinx Application Note xapp204. May 2000, http://www.xilinx.com/

[4] D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose, "The Stratix Routing and Logic Architecture" in FPGA '03, ACM. Symp. FPGAs, February 2003, pp. 15-20.

[5] P. Yiannacouras, "An Automatic Cache Generator for Stratix FPGAs," BASc Thesis. University of Toronto, 2003. http://www.eecg.utoronto.ca/~yiannac/docs/bascthesis.pdf