Multi-Core Architecture on FPGA for Large Dictionary String Matching *

Qingbo Wang, Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2562
qingbow, prasanna@usc.edu

Abstract

FPGA has long been considered an attractive platform for high performance implementations of string matching. However, as the size of pattern dictionaries continues to grow, such large dictionaries can be stored in external DRAM only. The increased memory latency and limited bandwidth pose new challenges to FPGA-based designs, and the lack of spatial and temporal locality in data access also leads to low utilization of memory bandwidth. In this paper, we propose a multi-core architecture on FPGA to address these challenges. We adopt the popular Aho-Corasick (AC-opt) algorithm for our string matching engine. Utilizing the data access feature in this algorithm, we design a specialized BRAM buffer for the cores to exploit a data reuse existing in such applications. Several design optimization techniques are utilized to realize a simple design with high clock rate for the string matching engine. An implementation of a 2-core system with one shared BRAM buffer on a Virtex-5 LX155 achieves up to 3.2 Gbps throughput on a 64 MB state transition table stored in DRAM. Performance of systems with more cores is also evaluated for this architecture, and a throughput of over 5.5 Gbps can be obtained for some application scenarios.

1 Introduction

String matching looks for all occurrences of a pattern dictionary, in a steam of input data. It is the key operation in search engines, and is a core function of network monitoring, intrusion detection systems (IDS), virus scanners, and spam/content filters [3, 4, 15]. For example, the open-source IDS Snort [15] has thousands of content-based rules, many of which require string matching against entire network packets, i.e. deep packet inspection. To support heavy network traffic, high performance algorithms are required to prevent an IDS from becoming a network bottleneck.

FPGAs have been attractive for high performance implementations of string matching due to their high I/O bandwidth and computational parallelism. Application specific optimizations for string matching algorithms have been proposed for FPGA-based designs [18]. They typically use a small dictionary, on the order of a few thousand patterns (e.g., see [3, 4]). Thus the state transition table (**STT**) generated from a Deterministic Finite Automaton (DFA) representation of the pattern dictionary, or the pattern signatures themselves, can be stored in the on-chip memory or in the logic of FPGAs.

However, the size of dictionaries has increased greatly. A dictionary can have 10,000 patterns or more [14, 15] now, resulting in an STT table tens of megabytes in size. Such large tables can be stored only in external memory and incur long access latency. Since every character searched requires a memory reference, this latency increase degrades the string matching performance. The problem is worsened by the fact that string matching presents little memory access locality and that access to the STT is irregular.

In this paper, we propose a multi-core architecture on FPGA for large dictionary string matching. We use the Aho-Corasick algorithm (AC-opt) for design verification, but the architecture can be applied to any such algorithms that employ a DFA stored in DRAM for pattern matching [16]. Our study shows, using AC-opt algorithm, that a small number of frequently visited states exist in the process of string matching, and the majority of memory references during string matching go to these "hot" states. When we allocate these states on FPGA to enable on-chip access to them, not only can the traffic to external memory be significantly reduced, but the throughput for the string matching engine is also improved due to fast on-chip access. Our major contributions are:

- To the best of our knowledge, our architecture is the first multi-core architecture on FPGA for large dictionary string matching to address the challenge of

*Supported by the United States National Science Foundation under grant No. CCR-0702784. Equipment grant from Xilinx Inc. is gratefully acknowledged.

DRAM access latency. The BRAM buffer scheme in this architecture is an application of a data usage feature in the AC-opt algorithm, where a set of states are visited more often than the others.

- Several design optimizations are proposed to improve the performance of this architecture, such as DFA remapping, pipelined BRAM buffers, and thread virtualization with shift registers for thread scheduling and synchronization, etc. The schemes result in a simple design and high clock rate implementation on FPGA, and DRAM access latency can also be partially hidden.
- An implementation of two core system on a Xilinx Virtex-5 LX155 demonstrates the high performance of the proposed architecture. The design employs BRAM buffers of 1K states to serve a 64K state STT. Based on the Place & Route results, it can run at over 200 MHz using less than 2% of logic resources on the chip. It can achieve up to 3.2 Gbps in throughput for some input streams. We evaluate systems with more cores based on the implementation experience.

The rest of the paper is organized as follows. In Section 2, we introduce related work and background on Aho-Corasick algorithm. The feature of AC-opt algorithm is introduced in Section 3, and the FPGA-based architecture for large dictionary string matching is also presented. In Section 4, design optimization techniques are introduced. The implementation experience and performance evaluation are presented in Section 5. We conclude with a summary and discussion on future work.

2 Related Work and Background

2.1 Related Work

Many schemes for string matching on FPGA have been proposed. In [5], a novel implementation using a Bloom filter was introduced. The hash-table lookup uses only a moderate amount of logic and memory, but searches thousands of strings for matches in one pass. Also, a change in the rule set does not require FPGA reconfiguration. However, the tradeoff between the false positive rate and the number of rules stored in the memory leads to performance degradation for large dictionary string matching.

A search engine using the Knuth-Morris-Pratt (KMP) Algorithm [7] on FPGA was presented in [4]. The authors adopt a systolic array architecture for multiple pattern matching, where each unit is responsible for one pattern. The unit architecture uses a modified KMP algorithm with two comparators and a buffered input to guarantee that one character can be accepted into the unit at every clock cycle. The pattern and its pre-computed jump table are stored in BRAMs. This design results in highly efficient area consumption on FPGAs, but is limited by the available BRAM blocks on-chip.

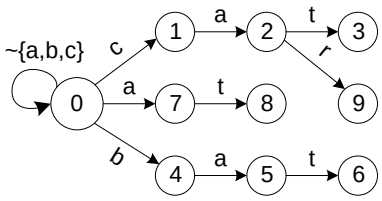
Chip multiprocessors (CMP) present new opportunities for fast string matching with their unprecedented computing power. In [19], researchers proposed new regular expression rewriting techniques to reduce memory usage on general purpose multi-core processors, and used grouping of regular expressions to enable processing on multiple threads or multiple hardware cores. Scarpazza et al. studied the optimization of Aho-Corasick algorithm on the Cell B.E. for both small and large pattern dictionary string matching [14]. When the patterns are in the range of a few hundred, one Synergistic Processing Element (SPE) using local store can obtain 5 Gbps throughput. However, when the dictionary includes hundreds of thousands of patterns, they must be stored in the external XDR DRAM, and the throughput can only reach 3.15 Gbps for 2 processors with 16 SPEs.

Recently, soft processors on FPGAs have gained lots of interest in the research community. Due to the demand for high performance in network security, bioinformatics and other applications [9, 12], FPGA and ASIC solutions have become more attractive. Using soft processors on FPGA, the engineering time can be reduced and software engineers can program the high performance hardware platform. In [13], a simplified IPv4 packet forwarding engine was implemented on an FPGA using an array of MicroBlazes. The softcore architecture exploited both spatial and temporal parallelism to reach a comparable performance to designs on an application-specific multi-core processor, i.e. Intel IXP2400. These studies motivate us to explore an multi-core architecture on FPGA to achieve high performance for large dictionary string matching.

2.2 Aho-Corasick Algorithm

A class of algorithms using automata have become more attractive [8] for string matching. From the classic Aho-Corasick algorithm [1] and its many variants, we selected the AC-opt for our design, since its theoretical performance is independent of dictionary size and keyword length [16].

The Aho-Corasick algorithm and its variants perform efficient string matching of dictionary patterns on an input stream S. It finds instances of the pattern keywords $P = (P1, P2, ..., Pn)$ in S, even when keywords may overlap with one another. All variants of Aho-Corasick function by constructing a finite state transition table (STT) and processing the input text character-by-character in a single pass. Once a state transition is made based on the current character, that character of the input text no longer needs to be considered. The construction of the STT needs to take place only once, and the STT can be reused as long as the pattern dictionary does not change. Each state also contains an output function. If the output function is defined on a given state, that state is considered to be a final state, and the output function gives the keyword or keywords that have been found.



(a) Goto function

i	1	2	3	4	5	6	7	8	9
f(i)	0	7	8	0	7	8	0	0	0

(b) Failure function

i	*output(i)*
3	cat,at
6	bat,at
8	at
9	car

(c) Output function

Figure 1. AC-Failure Example

The most basic variant of Aho-Corasick, known as AC-fail, requires the construction and usage of two functions in addition to the standard output function: goto and failure. The goto function contains the basic STT discussed above. A state transition is made using the goto function based on the current state and the current character of input. The failure function supplements the goto function. If no transition can be made with the current state and input character, the failure function is consulted so that an alternate state can be chosen and text processing may continue. This example illustrates usage of the AC-fail variant of Aho-Corasick. Figure 1 shows the goto, failure and output functions for a dictionary of the following keywords: cat, bat, at, car.

States	a	b	c	r	t
0	7	4	1		
1	2	4	1		
2	7	4	1	3	9
4	5	4	1		
5	7	4	1		6
7	7	4	1		8
3,6,8,9	7	4	1		

Table 1. DFA state transition table

A more optimized version of Aho-Corasick is also presented in [1], known as AC-opt. We use this algorithm in our study. AC-opt eliminates the failure function by combining it with the goto function to obtain a next-move function. The result is a true DFA, which is capable of string matching by making only one state transition per input character. Searching is therefore simplified and more efficient. However, since the construction of the next-move function requires both the goto and failure functions, it is less efficient than AC-fail when processing the dictionary initially.

Table 1 illustrates an AC-opt DFA. The DFA is constructed from the dictionary consisting of the keywords used before. Figure 2 shows a search on the input string "caricature."

input		c	a	r	i	c	a	t	u	r	e
state	0	1	2	9	0	1	2	3	0	0	0

Figure 2. A String Search Example

3 Algorithm Analysis and Multi-Core Architecture

3.1 Analysis of AC-opt

In our study, the STTs usually have more than 60K states, which constitute the rows of a 2-dimensional array. The number of columns is the same as the size of the alphabet. Thus, a large dictionary needs 60 MB storage or more, which can be stored on DRAM only. We believe that even for a significantly large input stream, the states traversed during a string matching are concentrated on a small part of that STT only. Furthermore, the few states visited most by that string matching engine satisfy the majority of state transitions during the process of matching.

	% States in levels 0,1	% Hits
Full-text search	0.114%	46.79%
Network monitoring	0.114%	81.46%
Intrusion detection	0.506%	89.84%
Virus scanning	0.506%	88.39%

Table 2. A few states in levels 0 and 1 are responsible for the vast majority of the hits

The four representative application scenarios in our study are (1) a full-text search system, (2) a network content monitor, (3) a network intrusion detection system, and (4) an anti-virus scanner. These are the same as in [14]. In scenario (1), a text file (the King James Bible) is searched against a dictionary containing the 20,000 most used words in the English language. In scenario (2), network traffic is captured at the transport control layer with "wireshark" [6] while a user is browsing multiple popular news websites. This capture is searched against the same English dictionary as before. In scenario (3), the same network capture is searched against a dictionary of about 10,000 randomly generated binary patterns, whose length is uniformly distributed between 4 and 10 characters. In scenario (4), a randomly generated binary file is searched against the randomly generated dictionary.

The authors in [14] showed that the DFA states at levels 0 and 1, whose distances to the initial state are 0 and 1 respectively, attract a vast majority of the hits to memory, as

listed in Table 2. We further recorded the number of hits to each state during the search processes in the same four scenarios, and ranked the states in descending order of visits. From the results presented in Figure 3, which shows the number of top states and the percentage of memory accesses that go to these states, we can see that more than 85% of accesses by the string matching engine go to the top 1000 states. We define the **hot state**s as the set of states that are visited most during a string matching process.

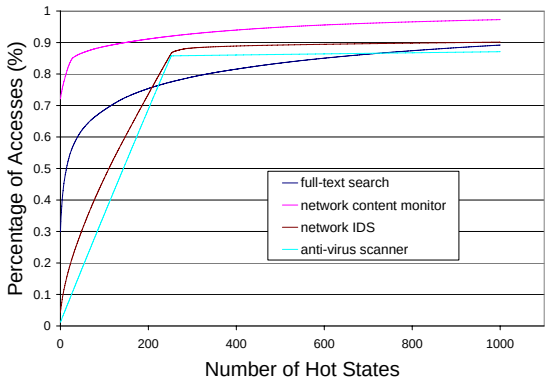


Figure 3. Memory Accesses to Hot States

3.2 Architecture Overview

Our proposed architecture is shown in Figure 4. There are p cores sharing a STT through an interface to DRAM(s). Each core, e.g. C_i in the figure, is equipped with a copy of on-chip buffer B_i, serving an input stream S_i, and giving an output O_i when available.

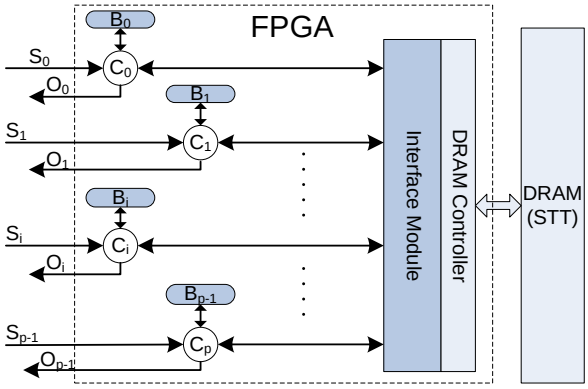


Figure 4. Architecture Overview

Utilizing the identified data usage feature from last section, the buffers are employed to store the hot states on-chip to reduce off-chip memory references, and to take advantage of fast on-chip memory access. When an address to STT arrives, the core logic makes a decision to direct it to either off-chip DRAM or on-chip storage. If a large number of hot states are stored on-chip, the buffer can resolve majority of references to STT, thus improving throughput performance of string matching.

The cores are connected to external DRAM through an on-chip interface module and the DRAM controller. The DRAM controller can be off-chip, but is usually implemented on FPGA.

3.3 On-Chip Buffer for Hot States

To populate on-chip buffers with hot states, we first run a search against the STT using a training input trace which exhibits statistical similarity to the incoming traffic. The number of visits to each state of the DFA is recorded and the list is sorted in descending order. The states represented by the top n entries in the list are hot states. The larger the n, the higher the hit rate to on-chip buffer by a string matching engine. However, the selection of n is also affected by the type of buffer device, and the available buffer size, etc.

A state corresponds to a row in the STT, with 256 next state entries for an 8-bit represented alphabet. When we use a 32-bit data to store a next state entry, including the next state ID and output functions, 1 KB storage is needed for each state. The on-chip storage can be implemented as a fully associative cache on FPGA, such as a CAM. However, a CAM of thousands of entries can lower the system's clock rate and can become a performance bottleneck. Due to its fast access speed and large volume, Block RAM (BRAM) is an ideal choice for implementing on-chip buffer to hold a large number of hot states.

3.4 Structure of a Core

A single core architecture is shown in Figure 5. An input character is received by the Address Generator, and combined with the current state to generate a new address for STT reference. The STT is organized as a 2-dimensional matrix, with the row indices representing the state IDs, and the column indices the 8-bit input characters. In the DRAM, as well in the BRAM, the data are stored physically as a one-dimensional array in row major order. Thus, the address generation can be achieved by simply concatenating the current state ID with the 8-bit input character. This address is then used to determine whether this memory reference should go to on-chip buffer or off-chip DRAM.

To exploit the available DRAM bandwidth, we add multiple virtual threads to a core to take advantage of temporal parallelism. Each of these virtual threads processes an input stream, which could be a segment from a long input stream, or an independent stream. A virtual thread is a DFA engine traversing the STT with its input characters. The thread manager, along with thread context storage, is added to provide scheduling and synchronization among the virtual threads.

The thread context stores the status of a virtual thread, including core ID, virtual thread ID, the address to STT, the

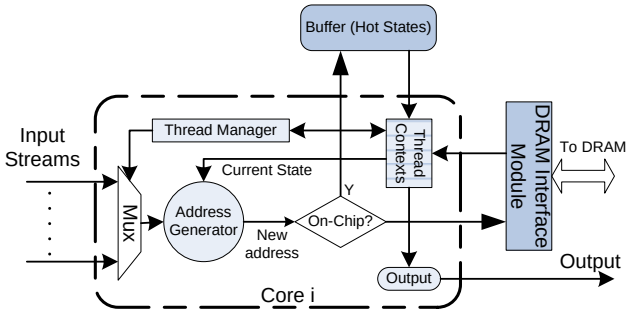


Figure 5. Structure of a Core

returned current state after a reference to STT is resolved, etc. It also keeps track of whether the reference to STT is on-chip or not. The thread manager chooses a ready thread, and picks its input stream through the Mux, for the address generator to process. The Output unit checks thread context registers to output the match.

3.5 Reconfigurable Multi-Core Architecture

In our architecture, a core executes on its own, even though they may affect other's performance through shared global devices, such as interconnect and DRAM. The p cores can be abstracted as p hardware threads, which add spatial parallelism to the architecture by multiplying execution from the single cores. As a result, this massive parallelism collectively exploit the DRAM bandwidth .

The multi-core architecture on FPGA can be described by the parameters, such as the number of cores, the number of virtual threads per core, the bandwidth and latency of external DRAM, the size, latency and access bandwidth of the on-chip buffer, etc. Constrained by the resources of FPGA, these parameters can be chosen to achieve high performance for string matching.

4 Design Optimizations

4.1 DFA Re-mapping

The IDs of the identified hot states were initially assigned during the building of the DFA, and are unlikely to be contiguous. As the state IDs are used by hardware logic to decide which memory to reference, the discontinuity of the hot state IDs complicates the hardware design. We adopted an ID re-mapping scheme to ease this by shifting the IDs of hot states to the beginning of the STT, making them top states. The state space is divided into two domains after the re-mapping, where a state with an ID number lower than n, the number of the selected hot states, goes to access on-chip buffer, and others external DRAM. Hence, the design for this decision-making becomes a comparator.

4.2 Simplified Thread Synchronization by Input Interleaving

For thread scheduling and synchronization, there are different ways to design the thread manager and thread context store shown in Figure 5. To reduce implementation complexity, we used an input interleaving scheme for thread scheduling and synchronization. In this scheme, every virtual thread within a single core, identified with a thread ID from 0 to $m-1$, is assigned with an string segment, where m is the number of virtual threads for each core. These threads are polled in a round-robin fashion by the thread manager, therefore the input streams are essentially interleaved. Using this scheme, the thread contexts can be maintained by a shift register with help from the BRAM buffer design, which is introduced in next section. However, a drawback of this design is that a stall of execution can occur when the head thread's reference to STT, specifically to DRAM, has not returned, thus losing performance. Our evaluation shows that when m is reasonably large, our design does not suffer significantly from the stalls.

When the interleaved input streams are the segments from one input trace, it can happen that matchable patterns across the boundary of two consecutive segments are missed. To avoid this hazard, an overlap, equal to the length of the longest pattern minus one, is preserved between the neighboring segments when partitioning [14]. This method slightly decreases the overall throughput, but guarantees the correctness of the string matching engine.

4.3 Shared and Pipelined Buffer Access Module

We utilized a shared and pipelined design for the BRAM buffer access module. BRAMs on FPGA can be naturally configured for dual-port access without loss of performance, so two cores can share one buffer, where a set of hot states is stored. To increase the clock frequency of buffer access, we adopt a pipeline architecture inside this module.

As illustrated in Figure 6, the BRAMs in the buffer module are divided into k even partitions, denoted $M_0, M_1, \cdots M_{k-1}$. The selected hot states are also divided evenly into k groups, and each group is stored on one of the BRAM partitions. The accessing elements, denoted AE, are separated by the pipeline registers. An AE is responsible for accessing its local BRAM and relays data from stage to stage.

As illustrated in Figure 6, a core sends a thread's context with address information into the BRAM buffer module. For a thread accessing DRAM, its thread context is simply passed through stage registers without any processing. However, if a thread needs to access the BRAM partitions, an AE, getting data from both the BRAM and the pipeline register of the previous stage, must do the following:

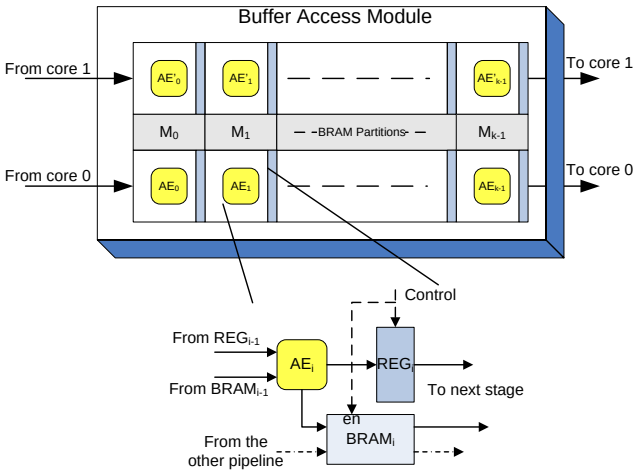


Figure 6. BRAM Buffer Access Module

- If the arriving thread needs access to BRAM buffer, and the STT address falls in the range of its local buffer, send the address into the local BRAM.
- If the output of the previous BRAM partition is the resulting current state for this thread, pass it to the next pipeline register.
- Otherwise, the data from the previous pipeline register is passed along to the next stage.

The "control" signal is used by the thread manager to hold the pipeline when a stall is necessary. Note that all BRAM partitions have output registers, and do not need pipeline registers in between.

4.4 Interface between Cores and DRAM Controller

Our design uses a multiplexing FIFO to interface the cores and the DRAM controller. A simple time-slotted round-robin scheduling is used to serve incoming requests from each core. By "time-slotted," the addresses coming at a clock cycle go through the FIFO according to a predetermined order, e.g. the IDs of the cores, before the requests from the next clock cycle. This scheme is consistent with the virtual thread management introduced in Section 4.2. It performs because our input streams to the cores are independent of each other, and they all have equal priorities.

We adopt a design from Le, et al. [10], and adapt it to a 4-to-1 basic unit of synchronous FIFO with conversion. Higher ratio FIFO can be formed using multiple basic units. The FIFOs are implemented using registers and logic only, to save BRAM for hot state buffer. A common implementation of a FIFO is a circular buffer, with read and write address. The size of the queue sets the maximum number of entries can be stored, and is bounded by $m \times p$ in our design. This is due to the fact that a thread does not send a new request until its last request is served, therefore the maximum number of the active threads, i.e. the STT references, is $m \times p$.

5 Implementation and Evaluation

5.1 Implementation on FPGA Platforms

We implemented our architecture on a Virtex-5 XL155 for a 64K state STT with 2 cores. A STT of 64K states needs a 16-bit representation of its state IDs. According to the analysis in Section 3.1, the larger the buffer size, the better the throughput can be. We target a buffer of 1K states, which needs at least 4 Mb BRAM on FPGA. The Virtex-5 LX155 has 192 BRAM blocks of 36 Kb, i.e. 6912 Kb in total, which is sufficient to buffer 1K states in our design.

Our implementation can run at over 200 MHz using the design optimizations introduced in Section 4. While the BRAM usage is at 66% of the chip, the logic resource consumption for the two cores is less than 2%. We can place dozens of cores on a mid-sized FPGA.

Our design works with a customized DRAM controller connected by a FIFO queue that can have different clock frequencies for write and read. The controller is based on a DDR2 controller generated by the Memory Interface Generator (MIG) tool in Xilinx ISE design suite 10.1.

5.2 DRAM Access Module

To evaluate system performance for string matching with large dictionary and long input trace, the behavior of DRAM modules must be studied. DDR SDRAMs have gained popularity, by increasing operating frequency, to become the standard of DRAMs currently. In Table 3, we identify some critical timing specifications for DDR SDRAM from the published technical documentation of DRAM vendors [11].

SDRAM	DDR	DDR2	DDR3
tRTP	12	7.5	10
tRC	55	55	50
tRRD	10	10	10
tRAS	40	45	37.5
tRP	15	15	15
tRCD	15	15	15
tCL	15	15	15
Clock period	5	3	2.5
Banks	4	4/8	8

Table 3. DDR SDRAM key electrical timing specifications. All units are **ns** except for the banks. Clock period is when working with FPGA. **tRTP**: Read-to-Precharge delay. **tRC**: Active-to-Active delay in the same bank. **tRRD**: Active-to-Active delay between different banks. **tRAS**: Active-to-Precharge delay **tRP**: Precharge latency. **tRCD**: Activate latency. **tCL**: Read Latency.

FPGA vendors support DDR3 with a clock rate of 400 MHz and DDR2 with 333 MHz on their development platforms [2, 17]. However, when the data access is irregular and spatial locality can not be utilized, DRAM delays and latencies become more important than their peak data rates. The parameters in Table 3 can be divided into two classes. The first includes such timing requirements as $tRTP$, tRC, $tRRD$, $tRAS$, etc., which specify the delays to be satisfied between consecutive operations on DRAM. The second is latencies, including tRP, $tRCD$, tCL, which are the time needed for an operation to complete. These parameters are used in our simulation program to estimate the performance for our design. Note that the numbers for the parameters may vary slightly for different DRAM modules.

5.3 Performance Evaluation

Performance of string matching is measured by throughput in $Gbps$. We first study the performance of our design implementation for a 2-core system. The two cores share a BRAM buffer of 1K states and an external DRAM, the Micron MT47H64M16 DDR2 SDRAM. The DRAM has a 16-bit data bus, 8 internal banks with a selected burst length of 4. The number of pipeline stages in the BRAM buffer is set to 8. The number of virtual threads was varied to see the impact on performance.

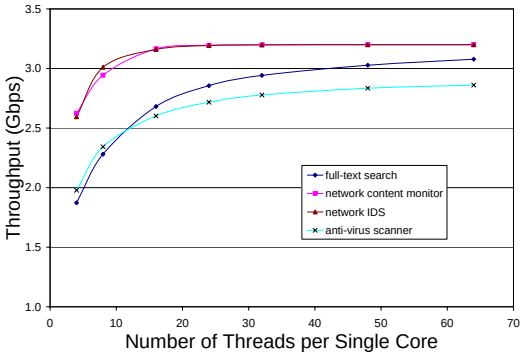


Figure 7. Throughput for a 2-Core System

Our simulation program set a 200 MHz clock rate for the cores while allowing the DRAM module to runs at 333 MHz. The DRAM refresh cycles were ignored due to its little effect on the performance. When we have 2-way 8-bit input at 200 MHz, the maximum throughput achievable by the design is 3.2 Gbps. The results for the four application scenarios from Section 3.1 are shown in Figure 7. As the number of virtual threads per core increases, the throughput generated by the two cores also grows to about 3 Gbps. For application scenarios such as network content monitor and network IDS, the optimal throughput of 3.2 Gbps is approached when the number of threads is large. This means all accesses to DRAM are returned before a requesting thread gets its turn to run again, thus eliminating the stalls of execution.

We then fixed the total size of BRAM buffer on chip, and varied the number of cores to study the performance of our architecture under memory constraints. As the number of cores increases, the buffer size of each core decreases, and so is the number of hot states, which reduces the hit rate to the on-chip buffers.

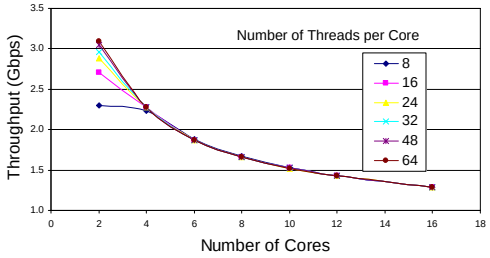


Figure 8. Performance of Multi-Core Architecture for Full-Text Search

Figure 8 shows that for a full-text search, our design with two cores can yield best performance for all cases with different number of threads per core. The throughput for all designs is then decreasing even though the number of cores keeps increasing. For the network content monitor experiment, the performance peaks at about 5.5 Gbps for a four core design, as shown in Figure 9. And then the throughput degrades similarly as the last one.

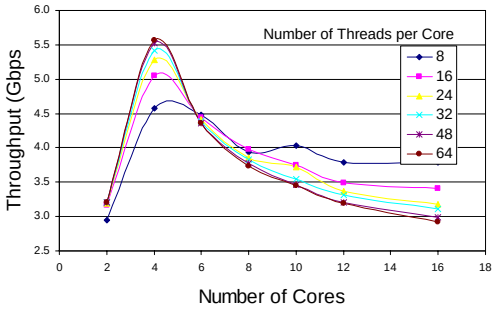


Figure 9. Performance of Multi-Core Architecture for Network Content Monitor

As the design changes to have more cores, the hit rate to on-chip buffer will drop, due to reduced buffer size, to a level at which the references to DRAM deplete the available bandwidth, and no more benefits come from increasing threads. At that time, Adding more cores only intensifies the contention for DRAM, therefore deteriorating the overall performance of the design. The reason for the different peaking points between the two experiments is that the hit rate curves shown in Figure 3 are different for the two. the full-text search's hit rate to on-chip buffer is lower than the one for the network content monitor, given the same number of hot states, so its performance should saturate earlier than that of the network content monitor. The difference in peak throughputs for the two experiments also shows the performance of our proposed architecture is contingent on input stream characteristics.

5.4 Performance Comparison

While there is no published research on FPGA with which to compare our work, we studied large dictionary string matching on a Dell XPS 410 with an Intel Core 2 Quad Q6600 processor for the full-text search scenario. The Intel C/C++ 10.1 compiler applies system-specific optimizations, which can allow Cycle-Per-Instruction (CPI) for a program to reach close to $1/4$. Similar to our scheme in the FPGA design, a "training" pre-fetch is used to boost the performance. The "training" technique conducts search on training input statistically similar to the real input. The intention is to load the cache with the most visited states. The results for 4 cores are presented in Table 4.

Table 4. Performance of String Matching on a Multicore System

Measured Throughput (Gbps)	Best	Average
5,000 patterns	5.5	3.2
5,000 patterns, Trained	10.0	5.8
50,000 patterns	3.3	2.3
50,000 patterns, Trained	4.7	3.4

Another thorough study of large dictionary string matching on Cell B.E., is presented in [14]. In this study, the authors found the XDR DRAM in the system performs best in random access when an SPE performs 16 concurrent transfers of 64-bit blocks, and proposed a 16 input interleaving scheme for one SPE. Assisted by other optimization techniques on memory system and local stores, the design achieves a theoretical aggregate pattern search throughput of 3.15 Gbps on a 2 Cell processor system with 16 SPEs. Our proposed architecture with only 2 to 4 cores exhibits competitive performance to these highly advanced multicore CMPs, without resorting to a high performance proprietary DRAM system as Cell B.E. does.

6 Conclusions

This paper presented a multi-core architecture on FPGA for large dictionary string matching. By buffering "hot states" on on-chip BRAM, the off-chip DRAM references were significantly reduced. Using proposed optimization techniques, our architecture can be realized with high operating clock rate for designs on FPGA. The achieved throughput performance is comparable to the solutions on other state-of-the-art multicore systems, while consuming only a small fraction of logic resources on FPGA. Our future work will study the effects of deploying more DRAM modules on the system performance, and explore the possibility of adding external SRAMs to the design.

7 Acknowledgments

We are grateful to Yi-Hua E. Yang, Weirong Jiang, Danko Krajisnik and Ju-wook Jang for helpful discussions and comments on an early draft of the paper.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] Altera Corporation. http://www.altera.com.

[3] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis. Performance analysis of content matching intrusion detection systems. In *Proc. of the International Symposium on Applications and the Internet*, January 2004.

[4] Z. Baker and V. Prasanna. A computationally efficient engine for flexible intrusion detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1179–1189, October 2005.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Hot Interconnects*, pages 44–51, August 2003.

[6] Gerald Combs. http://www.wireshark.org.

[7] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, 2006.

[9] M. Labrecque, P. Yiannacouras, and J. G. Steffan. Scaling soft processor systems. In *Proceedings of FCCM*, pages 99–110, 2008.

[10] H. Le, W. Jiang, and V. K. Prasanna. A sram-based architecture for trie-based ip lookup using fpga. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:33–42, 2008.

[11] Micron Technology, Inc. http://www.micron.com.

[12] G.-G. Mplemenos and I. Papaefstathiou. Soft multicore system on FPGAs. In *Proceedings of FCCM*, pages 199–201, 2008.

[13] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An FPGA-based soft multiprocessor system for ipv4 packet forwarding. In *Proceedings of FPL*, pages 487–492. IEEE, 2005.

[14] D. P. Scarpazza, O. Villa, and F. Petrini. Exact multi-pattern string matching on the Cell/B.E. processor. In *Conf. Computing Frontiers*, pages 33–42, 2008.

[15] SNORT: The Open Source Network Intrusion Prevention and Detection System. http://www.snort.org.

[16] B. W. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. *Technical Report, Eindhoven University of Technology*, 19(10):1179–1189, October 1994.

[17] Xilinx Incorporated. http://www.xilinx.com.

[18] S. Yi, B.-K. Kim, J. Oh, J. Jang, G. Kesidis, and C. R. Das. Memory-efficient content filtering hardware for high-speed intrusion detection systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 264–269, 2007.

[19] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, New York, NY, USA, 2006. ACM.