DPICO: A High Speed Deep Packet Inspection Engine Using Compact Finite Automata

Christopher L. Hayes * and Yan Luo
Department of Electrical and Computer Engineering
University of Massachusetts Lowell
Lowell, MA, 01854 USA
hayesc3@rpi.edu, yan_luo@uml.edu

ABSTRACT

Deep Packet Inspection (DPI) has been widely adopted in detecting network threats such as intrusion, viruses and spam. It is challenging, however, to achieve high speed DPI due to the expanding rule sets and ever increasing line rates. A key issue is that the size of the finite automata falls beyond the capacity of on-chip memory thus incurring expensive off-chip accesses. In this paper we present $DPICO$, a hardware based DPI engine that utilizes novel techniques to minimize the storage requirements for finite automata. The techniques proposed are modified content addressable memory (mCAM), interleaved memory banks, and data packing. The experiment results show the scalable performance of DPICO can achieve up to 17.7 Gbps throughput using a contemporary FPGA chip. Experiment data also show that a DPICO based accelerator can improve the pattern matching performance of a DPI server by up to 10 times.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General—*Security and Protection (e.g., Firewalls)*; C.2.3 [**Computer Communication Networks**]: Network Operations—*Network monitoring*

General Terms

Algorithms, Design, Performance, Security

Keywords

Finite Automata, Content Addressable Memory, FPGA, Intrusion Detection

*Christopher Hayes has since moved to Rensselaer Polytechnic Institute.

ANCS'07, December 3–4, 2007, Orlando, Florida, USA.

1. INTRODUCTION

Computer networks are becoming increasingly vulnerable to many threats, such as intrusions, worms, viruses and spam. When these threats, which continuously increase in number and complexity, are coupled with constantly increasing network line rates, the situation becomes more difficult to contain. Deep Packet Inspection (DPI) plays an important role in detecting threats by searching the payload of network packets for known patterns or signatures [5, 14]. It is also essential in traffic characterization and fine-grained network monitoring.

A DPI system performs a set of time-critical operations to sense certain network patterns or behavior while trying to minimize packet processing latency. The first step is to capture packets from network interface cards, reassemble and buffer them for processing. The second step is to search known signature patterns in the payload of packets. Following that, a DPI system will analyze the matched packets against semantic-rich policies to determine if an attack is present or an alert should be triggered. We argue that the first step can be sped up with Application Specific Integrated Circuits (ASICs) due to the invariance of the task, and the policy processing is better carried out with general-purpose processors because of the complex semantic processing. The matching of signature patterns falls in the middle of the spectrum, and can benefit from programmable hardware acceleration based on FPGAs or network processors.

The expanding signature sets and increasing line speed have made the signature pattern matching challenging. For example, the rule set of the well-known intrusion detection system, Snort [19], contains 4219 rules as of Dec 2005, and new rules are added constantly. Variable pattern length and location, and increasingly large rule sets make pattern matching a difficult task. Many string matching algorithms exist, such as those due to Boyer-Moore [9], Wu-Manber [22], and Aho-Corasick [4]. As more general cases of fixed strings, regular expressions used to depict pattern signatures further complicate the task due to possible exponential number of states. There have been numerous studies on regular expression matching [10, 23, 12, 2]. Different platforms have been used to perform DPI, including ASICs [20], FPGAs [11] and network processors [17]. Most of these methods rely on state machines or finite automata to match patterns. However, one of the key issues is that the size of the finite automata is so large (often at least tens of megabytes) that they have to be stored in off-chip memory modules. As a result, the searching on the automata incurs a large number of off-chip

memory accesses that lead to unsatisfactory performance. Thus it is important to minimize the storage requirements of state machines such that they can be put into on-chip or other fast memory modules to achieve high-speed pattern matching.

We are motivated to study efficient memory utilization of programmable hardware architecture to improve the performance of signature pattern matching. In this paper, we present *DPICO*, a hardware based DPI engine that utilizes novel techniques to reduce the storage for finite automata. We first analyze the storage of traditional Deterministic Finite Automata (DFA) and present the baseline design. Taking advantage of the multi-ported on-chip memory banks on modern ASICs and FPGAs, we then use three techniques to reduce the redundant information: (1) modified content addressable memory (mCAM), (2) interleaved memory banks, and (3) data packing. The combination of these concepts allows data to be organized in a more efficient way. Our analysis show that the storage can be reduced by over 90%. We have implemented an FPGA based prototype incorporating the proposed techniques. The DPI performance of the system is shown to be scalable and reach up to 17.7 Gbps using a contemporary FPGA chip. We evaluate the performance impact of incorporating a DPICO based accelerator in an x86/PCIe server architecture. The data show that the pattern matching performance of the DPI server can be improved by up to 10 times.

The contributions of our work are as follows. We present effective techniques to store compact finite automata in FPGAs, taking advantage of their architectural features. Our scalable design provides an empirical basis for architecting high performance DPI systems, where the pattern matching is an important processing phase. In addition, finite automata is necessary in upper level semantic policy processing, thus our approach can be extended to speed up other stateful packet inspection operations.

The paper is organized as follows: Section 2 reviews related work, Section 3 describes a motivating example and the baseline design, Section 4 elaborates the proposed techniques to reduce the storage of DFA, and the performance results are presented in Section 5. Finally, the paper is concluded in Section 6.

2. RELATED WORK

String matching techniques such as Bloom Filters [8] and Boyer-Moore [9], Wu-Manber [22], and Aho-Corasick [4] algorithms have been the foundation for many signature-based detection engines over many years. Some of these concepts have even been expanded to search against regular expressions instead of fixed strings [10, 23]. Regular expressions increase the capability and maintainability of threat detection systems by increasing the flexibility of threat definitions.

Schaelicke et al. characterized the performance of the Snort software on general-purpose processors. They showed that their highest performance test system could only simultaneously handle 217 payload rules on a network running at a 100 Mbps rate [18]. More recently, Dreger et al provided insights to the performance limitations of a Bro NIDS [15] on a commodity PC and explored ways to mitigate its resource demands. Through extensive experiments, they showed that Gbps network intrusion detection rate cannot be achieved without carefully tuning the system. Clearly, the performance of network intrusion detection implemented on general-purpose processors is deficient when considering increasing speed demands.

Paxson et al proposed to rethink the hardware support for efficient network analysis and intrusion prevention [16]. They described a high-level network security analysis pipeline, where the protocol analysis is one of the stages. Pattern matching against packet payload is indispensable to classify flows as increasingly more applications cannot be reliably identified by merely examining transport port numbers.

Many signature-based systems have been architected for the FPGA [21] and ASIC [10], taking advantage of the parallel structures available in these devices. These designs are predominantly based on the Aho-Corasick algorithm or other finite automaton-like structures. These architectures have incrementally improved the speed or storage utilization of signature matching through modification of the implementation.

These implementation-based improvements are complemented by algorithmic improvements directed at modifying the finite automata themselves. Kumar et al. [12] modified a DFA to combine common output transitions of individual states by creating a default transition between those states. This modified DFA, known as the delayed-input deterministic finite automata or D^2FA is found to reduce the DFA storage of a partitioned Cisco rule set from 92 MB to 2 MB [12]. This significant savings begins to bring finite automaton processing of large rule sets into a range that is conceivable on an FPGA or network processor. This storage improvement, however, is provided at the cost of the delayed input behavior, reducing the average processing throughput. Along with transition reduction, state merging techniques have also been proposed [6].

Lin et al proposed to reduce FPGA logics used to match regular expressions through sharing common sub-regular expressions [13]. Their method is similar to a number of Nondeterministic Finite Automata (NFA) based approaches [7], whose drawback is that those designs cannot easily accommodate new signature patterns. A new compilation-synthesis-placement-download procedure is needed because regular expressions are "hardcoded" into FPGA logic elements. In this paper, we focus on "memory based" finite automata because of its significant advantage over "logic based" finite automata: the memory based FA can be easily updated to incorporate new signature patterns without reprogramming FPGAs.

There are two popular categories of finite automata: Nondeterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA) [23]. They differ significantly in the complexity of storage and searching. The space complexity of NFA is $O(n)$ and its searching complexity is $O(n^2)$, while DFA's searching complexity is $O(1)$ at the cost of space complexity $O(2^n)$. For DPI systems, DFA is the preferred state machine especially for delay sensitive network applications, thus it is the focus of this paper.

3. MOTIVATION AND BASELINE DESIGN

3.1 Motivating Example

We assume that a specific network attack is present when the string "root" is found in the payload of a packet. Secondly, we assume a different type of attack is being prosecuted when the string "rmdir" is found in a packet. Each of these strings form the foundation of a rule and together

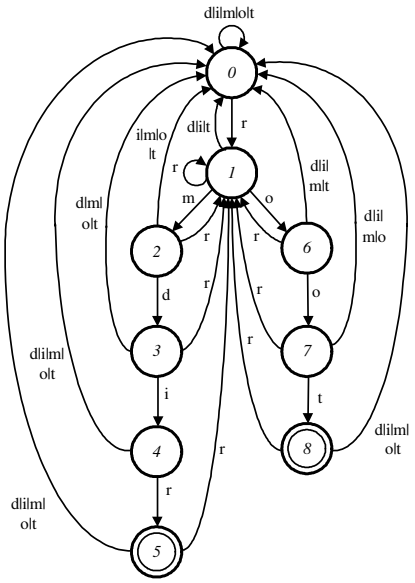

Figure 1: DFA for the Regular Expression $\{root|rmdir\}$.

they make a rule set that can be used on every packet to filter both types of attacks.

Here, we compile the rule set into a DFA that allows the rule strings to be found on at any point in the data stream, even when multiple strings overlap. Figure 1 shows a DFA for the regular expression $\{root|rmdir\}$ with the input alphabet $\Sigma = \{d, i, m, o, r, t\}$. The DFA contains nine states, two of which are accepting states, and 24 distinct transitions. When accepting state five is reached, the string "rmdir" has been matched, and likewise, when accepting state eight is reached, the string "root" has been matched.

3.2 Baseline

We use a traditional DFA implementation as a baseline for performance comparison. In this method, the state machine is implemented in two memory modules. One memory module contains the state transition table and the second module contains the match identifiers. The current state pointer and the input character are combined and used as the address to lookup the next state pointer. The current state pointer is also used as an address to lookup the match identifier for the current state.

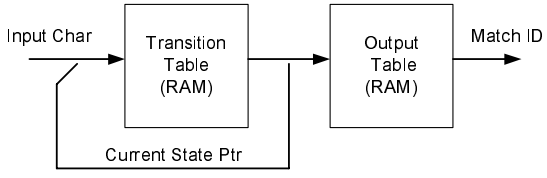

Figure 2: Baseline State Machine Block Diagram

When this method is used, each state is given a fixed and constant amount of memory. This section of memory contains a location for each possible input character. For instance, if the input character were eight bits wide, each state would require enough memory to contain 256 (2^8) next-state transitions. When using this technique, the minimum amount of memory (S) needed for a finite automaton is derived from the number of states in the automaton and the number of strings or regular expressions that can be matched.

$$S = 2^b N_S \lceil \lg(N_S) \rceil + N_S \lceil \lg(N_M + 1) \rceil$$

where b is the number of bits in the input character, N_S is the number of states in the automaton, and N_M is the number of MatchIDs in the automaton.

If the input to the state machine were a 3-bit character (to encode the alphabet in the motivating example), the baseline design would require 72 locations in memory; the product of the number of possible input characters by the number of states. Some of the information stored, however, is redundant as there are only 24 unique transitions stored in those 72 locations. It is through the elimination of this redundancy that we achieve increased storage efficiency.

4. PROPOSED TECHNIQUES

In general, multiple regular expressions are encoded into a single finite automaton. When an accepting state of the automaton is reached, a regular expression is matched. Each accepting state is given a Match ID to signify which regular expression has been matched. If the Match ID is all zeros, the current state is not an accepting state and hence there is no regular expression match.

In our design, we assume that the state transition table for the finite automata will be stored in a uniform cost RAM as shown in Fig. 2. Each state of a finite automaton consists of one or more next-state transitions. A transition will require exactly one location in memory. States in the DFA can contain a variable number of next-state transitions. Next-state transitions for any given state will be stored contiguously in the memory, and likewise, states themselves will be stored contiguously.

In such a design, we take advantage of three techniques to allow for the reduction of redundant information: (1) modified Content Addressable Memory (mCAM), (2) interleaved memory banks, and (3) data packing. The combination of these concepts allows data to be organized in a more efficient way.

4.1 Modified Content Addressable Memory

For a given DFA, the memory in the baseline design will contain repeated data. For instance, a given state will have many transitions that have the same next-state pointer. Since the baseline design explicitly stores every transition, the repetition of data is inherent to the architecture. To remove the pointer repetition, it is necessary to modify the next-state lookup technique. This is accomplished by creating two types of transitions: **labeled** transitions and **default** transitions. To translate an explicit state transition table for a DFA, where every possible transition is stored, to an implicit one, we take the most frequent next-state pointer for a given state and make it the default transition. All other transitions are converted into labeled transitions. Each state can contain up to one labeled transition for each label and each state will contain exactly one default transition.

This default/labeled transition technique can be applied to all the deterministic finite automata (DFA). Some DFA variants such as the delayed-input finite automaton (D^2FA [12]), have inherent default transitions. These transitions are different, however, because they cause the input to be

delayed when they are taken. States that do not have delayed default transitions can be given a non-delayed default transition that can be selected in the same manner as the DFA. In this case, the delayed and non-delayed default transitions need to be distinguished.

Using these transitions, we construct a modified CAM structure (mCAM) where each state has its own associative memory. Fig. 3 depicts a set of states stored in mCAM. Labels from the labeled transitions are the keys to the CAM and the next state pointer information is the data in the CAM. The default transition and state information are also stored in the memory and are not accessed associatively. The size of states is non-uniform thus it has to be stored within each state for identifying the boundaries. The variable size of states makes the state access and transition non-trivial.

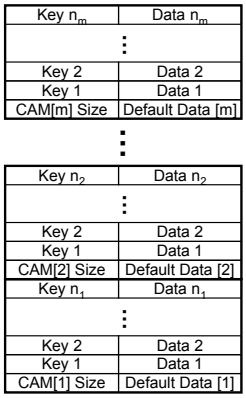

Figure 3: mCAM Diagram

All transitions for a state (default and labeled) are read from the memory. The labels are compared against the input character. If the input character matches one of the labels on a labeled transition, its next-state pointer is taken, otherwise, the next-state pointer for the default transition is taken. It is clear to see that this technique requires a lot of memory accesses in order to retrieve the pointer for the next state if a state contains a high number of unique next-state transitions. Later, we will see that this issue can be addressed in the FPGA implementation such that the next-state pointer can be resolved in a single clock cycle.

This mCAM technique has both advantages and disadvantages. The number of locations required for the state is reduced, however, the labeled transitions require storage of the next state pointer along with the label. Secondly, the state no longer requires constant space in the memory and can be packed more tightly in memory. This implies that the next state pointers would need to increase in size to account for the fact that states do not occur on regular boundaries. Finally, the match identifier for the state must be stored. Since every state contains exactly one default transition, the match identifier is stored with the default transition.

The minimum amount of memory required for this method is based on the number of states, match identifiers and labeled transitions. In this case, the effective storage savings from straightforward technique will be dictated by the ratio, r, of the average number of transitions per state for the automaton to the total number of possible transitions, 2^b. As that ratio decreases, the potential storage savings increases. The minimum memory required can be calculated as

$$S_{eff} = N_S(\lceil \lg(N_M + 1) \rceil) + N_T(\lceil \lg(N_T) \rceil + b)$$

$$r = \frac{N_T}{2^b N_S}, N_T \geq N_S$$

where N_T is the number of transitions in the automaton.

From this, we evaluate the differences in minimum storage requirements to see where this method is most effective. The results are shown in Figure 4 where x axis is the number of states in the DFA and y axis is the percentage of memory reduction $((S - S_{eff})/S)$. The proposed method is most effective when the finite automata with less average transitions per state. Also, any finite automaton with a ratio above 0.5 does not benefit from this method since the overhead of storing labels in the memory outweighs the savings from storing less transitions. Rule sets used in recent research by Kumar et al. [12] result in transition ratios between 0.01 and 0.09 when these rule sets implemented as a DFA. Most of these rule sets also have transition ratios less than 0.5 when implemented as D^2FA. Thus the proposed modified CAM structure has potential to reduce both the DFA and D^2FA storage significantly.

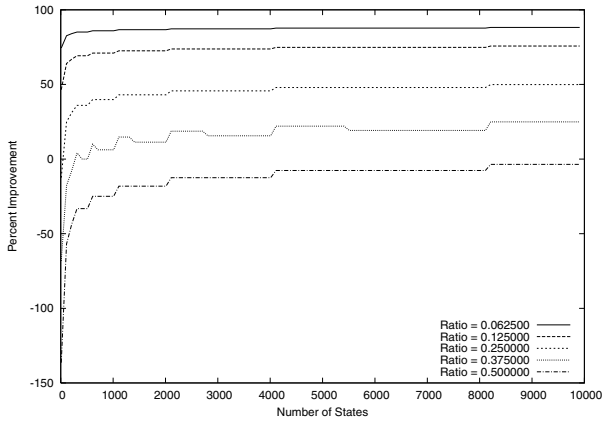

Figure 4: Storage Improvement with Varied Transition Ratios (b=8)

4.2 Interleaved Memory Banks

The CAM implementation will need to analyze all of the labeled transitions and the default transitions simultaneously in order to evaluate a state in a single cycle. Current FPGA products generally contain hundreds of individual memory elements on-chip. Using these elements, we propose that the state transition information be organized into sequentially interleaved memory banks (shown in Fig.5) to take advantage of the potential memory bandwidth available in FPGA devices. If the number of banks, n, is greater than or equal to the number of next-state transitions for the largest state (N_{max}), all the transitions can be read simultaneously and processed in parallel. This architecture will allow each state to be processed within a single evaluation cycle. If the number of next-state transitions exceeds n, a transition will take a constant time of $\lceil \frac{N_{max}}{n} \rceil$ cycles, which can be addressed through pipelined design. In fact, on-chip

256 and more memory banks have become common in modern FPGAs [3], which guarantee one-cycle state evaluation and transition.

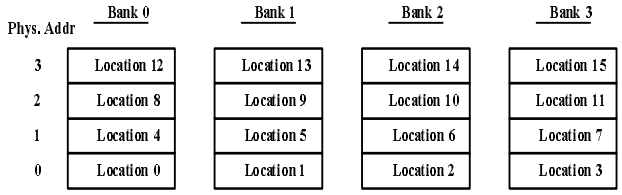

Figure 5: Sequentially Interleaved Memory Banks (n=4).

On each read cycle, n words are read in parallel from the interleaved memory. Any location in any bank can be the start address for the n-word read. For example, a read access can retrieve words 5, 6, 7 and 8, which span in two continuous rows. Since a state can have less than n transitions, the size of the state must be encoded so that the read logic can ignore unnecessary information. Here, we propose to insert the end offset of the state to the first location of the state so that the framing can be determined appropriately.

Each location in the memory either contains a default transition or a labeled transition. The location with a labeled transition will contain the label and the next state pointer associated with the transition. The location with a default transition will contain the next state pointer for the default transition and necessary state information since there is one default transition per state. The state information includes the offset of the last transition for the state and the match identifier for the state. Since every state has a default transition, it is placed as the first location for the state. Figure 6 shows the storage of labeled transitions and default transitions. The bit range of each field is indicated in the figure.

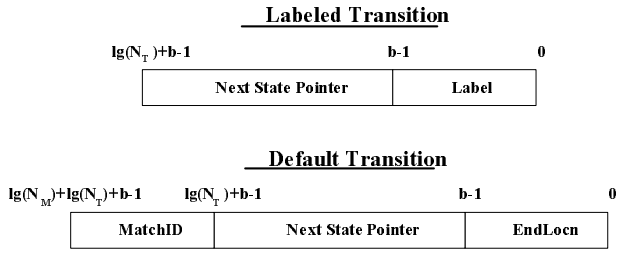

Figure 6: Storage of default transition and labeled transition.

Using these concepts, we propose a general architecture for performing regular expression matching. Fig.7 depicts the block diagram of the DPI engine called DPICO. The architecture consists of components for address calculation, transition storage, label comparison and determining next state address. The transitions are stored in a sequentially interleaved memory as discussed previously. Each memory bank has an address calculation unit to generate proper bank address based on the current state address. The input character is compared against the labels of the current state state through a set of selection logic (bottom portion of Fig.7.)

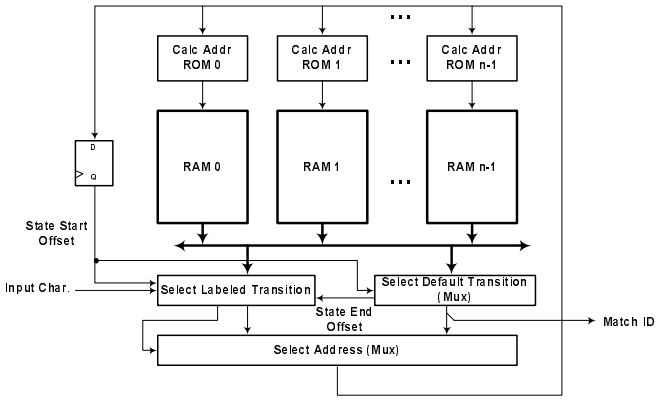

Figure 7: Block Diagram of DPICO.

4.3 Operation of DPICO

Since n is equal to or larger than the number of transitions for the state, all the transitions are read simultaneously from memory banks 0 through n-1. The transitions are then processed to identify the proper next state using selection logic. The logic of selecting labeled transition is shown in Fig. 8. All the transitions read from the RAM are qualified based on the beginning offset of the state and the end offset read from the default transition. The memory outputs outside of the range are simply ignored, and only the qualified labeled transitions are compared against the input character. If a match exists, a multiplexer selects the output of the RAM that contains that label. If there is no match the next state pointer from the default transition is used. On each read cycle, the match identifier, which is also contained in the default transition location, is output from the state machine. The delay of address calculation, memory access and next state selection logic determines the clock cycle time. Since the engine consumes one input character per clock cycle, the maximum processing throughput of this approach for a DFA is simply the product of the input character size and the processing clock frequency: $T = b \cdot f_{max}$.

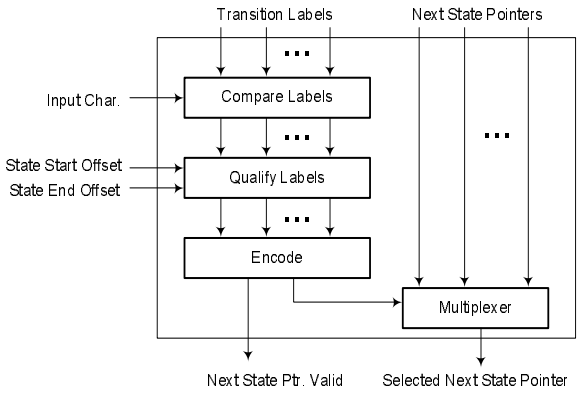

Figure 8: Diagram of Selecting Labeled Transition

4.4 Packing Across Memory Boundaries

In practical application, the previous solution does not equal the minimum memory utilization described as S_{eff}. Instead, the actual memory size (S_{act}) is driven by the size

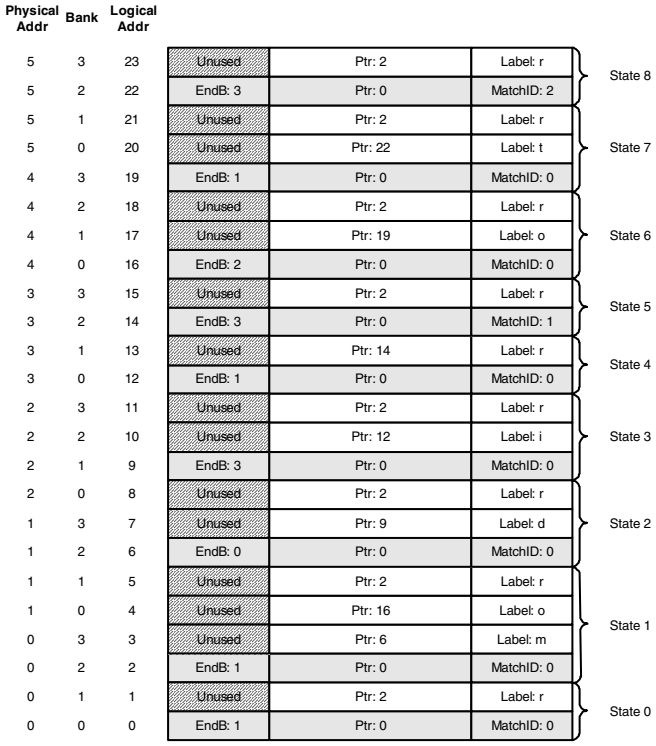

Figure 9: Memory organization for {root | rmdir}.

of a memory location, L, which is determined by the size of the default transition, S_D, and the labeled transition, S_L.

$$S_{act} = L \cdot N_T, L = max(S_D, S_L)$$

Fig.9 shows the memory organization of the motivating example. It can be seen that some unused bits are present in every label transition. The wasted memory space (denoted as "unused" in Fig. 9) comes mainly from the disparity in the size of the default and labeled transitions and the assumption that each transition would occupy one location in memory. The labeled transitions tend to have less information and therefore generally smaller than default transitions.

One method to better approach the minimum memory calculation is to pack the labeled transitions across memory location boundaries. Figure 10 shows an example of the packing strategy for a state machine with varied numbers of transitions per state. The larger default transition occupies a full memory location and is always positioned on regular memory location boundaries. Labeled transitions are packed between default transitions. Some space may be unused for a state. This unused space, however, tends to be smaller than the unused space for an unpacked design. This packing does not increase the complexity and latency of selecting the next state because it only changes which bits are fed to the comparison logic and all bits are read simultaneously anyway. The memory bandwidth is saved effectively.

5. PERFORMANCE EVALUATION

We evaluate our DPICO engine in two aspects. First we study the effectiveness of the design and explore the design space. Particularly we apply pipelining technique to optimize the design. Second, we investigate the performance

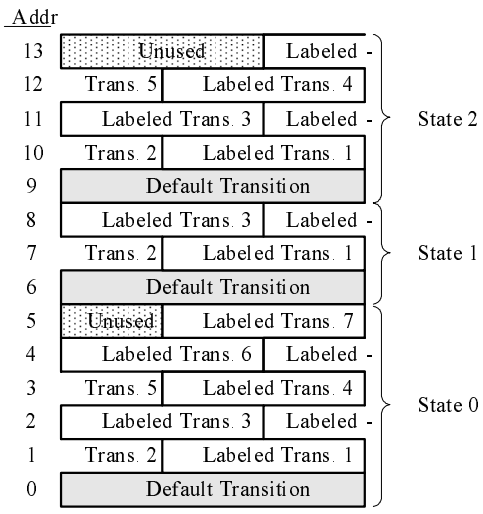

Figure 10: Example Packed Memory Map

Table 1: Compression Results

Rule Set	No. of Rules	Baseline Size (kbits)	DPICO Unpacked Size (kbits)	DPICO Min. Size (kbits)	Trans. Ratio (r)	Pct. Savings
imap	46	16,526.8	698.4	557.8	0.018	96.5
ftp	76	11,448.4	522.2	408.7	0.017	96.4
netbios	633	2,146.7	65.0	53.1	0.011	97.5
nntp	13	7,820.8	322.6	262.5	0.017	96.6
exploit	122	55,270.1	7,182.9	4884.0	0.046	91.2

impact of incorporating such a pattern matching accelerator to a commodity server architecture since our immediate goal is to speed up the pattern matching phase of the DPI workload with an acceleration card. We then discuss the applicability of DPICO as on-chip accelerators.

5.1 Performance Characteristics of DPICO

We characterized the performance of the DPICO design in two ways - (1) by measuring the compression factor against a particular DPI rule set and (2) by finding the speed and sizing characteristics of the design when implemented in an FPGA. This subsection summarizes those results.

To show compression results, we begin with a subset of the Snort rule set. We choose five representative rule files, namely imap, ftp, netbios, nntp and exploit because their size is relatively large in the whole set. The rule files are converted into DFA and D^2FA, in the memory table format of the baseline design. Next, these rulesets were converted into the table format of the DPICO design. The sizing results are found in Table 1 which shows overall DPICO can reduce memory usage by over 90% for all rule sets under study. Data packing can reduce about 20 - 30% of the storage.

The DPICO has been written in VHDL and targeted for a Xilinx Virtex 4 SX35 FPGA. The design was simulated using a VHDL simulator, synthesized with Synplicity Synplify and implemented using the Xilinx ISE tool chain.

The baseline DFA design was found to operate at a maximum of 267.7 MHz using almost no logic resources including Look Up Table(LUT) and Register(REG). Table 2 shows the speed and sizing results from the storage saving design

Table 2: Speed and Size of the DPICO

No. of Memory Banks	LUT	REG	f_{max} (MHz)	T_{max} (Mbps)
2	114	129	144.9	1159.2
4	183	204	122.3	978.4
8	320	352	106.9	855.2
16	698	642	98.7	789.6
32	1672	1252	84.8	678.4
64	3541	2346	78.9	631.2
128	7659	4810	74.0	592.0
256	16052	9563	68.1	544.8

Table 3: Speed and Size of Pipelined DPICO

No. of Memory Banks	Added Pipeline Stages	LUT	REG	f_{max} (MHz)	T_{max} (Mbps)	Non-pipelined T_{max} (Mbps)
2	1	102	149	247.8	1982.4	1159.2
4	2	189	253	307.4	2459.2	978.4
8	2	336	418	251.9	2015.2	855.2
16	3	596	889	271.9	2175.2	789.6

when the data is unpacked and the automaton is a DFA. It is assumed that the number of parallel reads is greater than or equal to the maximum number of transitions, both default and labeled, in any state. The throughput given assumes 8-bit input characters are used. The results show that the clock frequency decreases as the number of banks increase. This is because the propagation delay of the logic encompassed in the Select Labeled Transition Block and the Select Default Transition Block must increase as the number of banks increases. They contain multiplexers that must be sized appropriately to select data from the appropriate number of banks. Larger multiplexers have longer propagation delays, subsequently decreasing the clock frequency at which the design can operate.

We then add pipeline stages to the DPICO design to minimize the propagation delay between clocked elements (registers or block memory), thus increasing the maximum clock frequency. When pipeline stages are added, the state machine can process multiple time-multiplexed streams of data. The number of time slots is equal to the number of pipeline stages, and the bandwidth of each individual data stream is the total bandwidth of the state machine divided by the number of clocked stages. The pipelined design presents a tradeoff between maximum processing bandwidth for the engine, which increases when the number of pipeline stages increases, versus the processing bandwidth for each stream, which decreases as the number of pipeline stages increases. Table 3 shows the results for varied pipelined designs. Each row of the table has different number of memory banks and pipeline stages. It can be seen that pipelined design brings significant improvement over non-pipelined design. The reduced clock frequency with more memory banks affect the overall performance of pipelined designs: 4 banks of memory outperform the other configurations.

DPICO engines can provide scalable performance by incorporating multiple engines on one FPGA chip to exploit packet level parallelism. We implement such a multi-engine design in which each engine handles a packet flow independently. Detailed performance results are shown in Table 4. The first column is the number of memory banks under test including 2, 4, 8 and 16. The second column of the table shows the number of DPICO engines implemented on the chip, i.e. the number of packet flows that are processed simultaneously. We also compare the non-pipelined design with the pipelined design. It can be observed that multiple DPICO engines have little negative impact on the operation frequency of each engine. The total throughput of the pattern matching processing is scalable to the number of engines on a chip. With a contemporary FPGA chip, the performance can reach 17.7Gbps. Again, four banks give the best overall performance due to the complexity of comparison logics. The results also imply the potential of DPICO on more advanced FPGA chips.

5.2 Performance Improvement to a DPI Server Architecture

Our immediate goal is to incorporate DPICO acceleration card into a commodity server architecture with x86 cores and PCI Express buses. The DPICO engine sits in the FPGA on the PCIe accelerator and DFAs are preloaded. A DPI system such as Snort and Bro runs on x86 cores, which send packet streams to the accelerator for pattern matching. The packet transfer is over the PCIe bus using DMA operations. The DPICO engine search the payload against DFAs and sends the results of pattern matching back to the x86 cores. We'd like to study the potential of performance improvement of such a configuration.

We set up the experiment as follows. We use realistic rules from Snort rule set as the regular expressions to be matched. We capture packet traces at the network link connecting our campus to the ISP. The packet trace file contains 406K TCP and UDP packets with nonzero payload size. This realistic trace file is used as the input packet stream to the DPI system. We create a compiler to parse regular expressions and generate corresponding DFAs. We run the regular expression matching workload on a server with Dual Intel Xeon processors (4MB L2 cache) and 1GB memory. We instrument the workload with gprof [1] to record the time spent on the matching process (excluding the DFA creation, bookkeeping, etc.) Then, we load the DFAs into DPICO engine and process the same packet trace on the accelerator card. We compare processing time in the two scenarios, taking into account the overhead such as passing packet payload over the PCIe bus. In the experiment, the packets are transferred over PICe bus in sequential order, not being combined in batches to maximize the bandwidth utilization. Payload transfer and the pattern matching on FPGA are not pipelined, which can be optimized later. These conservative settings imply the worst performance of DPICO based accelerator. We leave these optimizations for future work.

Fig. 11 shows the results of the experiments. The y axis of the plot is the time spent on pattern matching. The left two bars represent the CPU processing time on the two Snort rule files, namely exploit.rules and web-cgi.rules. The processing time is the time spent on searching the DFAs. The righ-most five bars depict the time used by the DPICO engines (in configurations of 1 - 4 and 8 engines) together with the overhead. In fact the majority of the overhead is the payload transfer over PCIe bus (16x speed at 8GB/s). The delay of DPICO engines is deterministic as exactly one byte is consumed in one cycle, regardless of rule sets. This figure clearly shows that the performance improvement of a DPICO accelerator on an x86/PCIe server architecture can reach up to 10 times.

Table 4: Speed of DPICO with Multiple Engines

	Non-pipelined			Pipelined		
No. of Memory Banks	No. of DPICO Engines	f_{max} (MHz)	T_{max} (Mbps)	Pipe-line Stages	f_{max} (MHz)	T_{max} (Mbps)
2	1	150.6	1204.8	2	269.8	2158.4
2	2	137.4	2198.4	2	269.8	4316.8
2	3	150.2	3604.8	2	269.8	6475.2
2	4	133.5	4272.0	2	248.3	7945.6
2	8	137.8	8819.2	2	242.8	15539.2
4	1	129.9	1039.2	3	287.1	2296.8
4	2	129.8	2076.8	3	287.1	4593.6
4	3	124.4	2985.6	3	254.2	6100.8
4	4	120.5	3856.0	3	277.0	8864.0
4	8	121.3	7763.2	3	277.3	17747.2
8	1	106.5	852	4	212.9	1703.2
8	2	98.6	1577.6	4	215.6	3449.6
8	3	98.6	2366.4	4	192.1	4610.4
8	4	98.6	3155.2	4	190.3	6089.6
8	8	101.1	6470.4	4	212.3	13587.2
16	1	97.5	780	4	202.4	1619.2
16	2	95.1	1521.6	4	192.4	3078.4
16	3	97.5	2340.0	4	202.4	4857.6
16	4	94.9	3036.8	4	187.3	5993.6
16	8	92.6	5926.4	4	187.3	11987.2

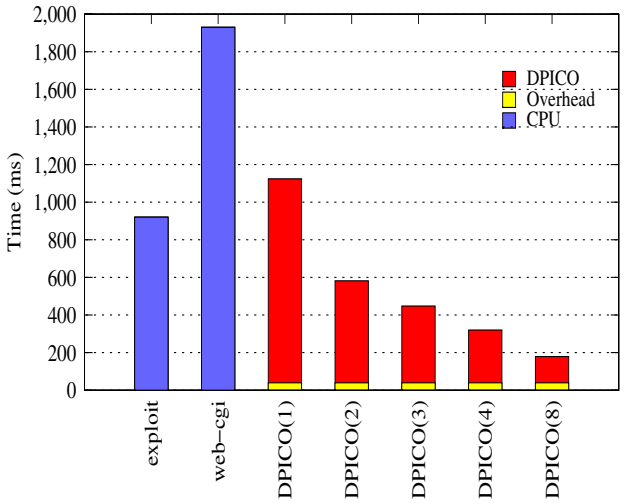

Figure 11: Performance Improvement to x86/PCIe Server Architecture

5.3 Discussions

The DPICO design takes advantages of some features of modern FPGA chips, but its applicability is not limited to FPGAs. We argue that DPICO can be integrated as an on-chip accelerator of a general purpose CPU, although it is not the focus of this paper. The control logic of DPICO is simple thus does not demand significant silicon area. On-chip DFA storage can eliminate the overhead of transfering payloads over peripheral buses, however, on-chip memory banks in general purpose CPUs are expensive in area and power consumption. Correlations among memory size, performance and power is worthy studying. As pattern matching rule sets expand, the limited on-chip memory may get fully utilized thus requiring swapping unused DFAs to main memory. So, it is also interesting to investigate the trade-offs between a dedicated on-chip DFA memory and a shared DFA/cache memory, and related DFA replacement policies.

6. CONCLUSION

In this paper we present a high speed DPI engine, DPICO, which is shown to be most effective when the average transition ratio of a DFA is significantly less than 0.5. The proposed techniques can reduce the memory usage of DFAs by 90%. Performance evaluation results show that a pipelined implementations with multiple DPICO engines can reach total a throughput up to 17.7Gbps in contemporary FPGA devices. Experiment data also show up to 10 fold improvement on pattern matching when incorporating an DPICO based accelerator to an x86/PCIe server architecture running DPI workload. The high speed and scalability of DPICO make it a promising candidate for a wide range of DPI applications such as network intrusion detection and spam filtering.

Acknowledgment

This work is supported in part by the National Science Foundation under grant No. CNS 0709001 and a grant from Intel Research Council.

7. REFERENCES

[1] Gnu gprof. Free Software Foundation.

[2] TRE: POSIX Compliant Regular Expression Matching Library. http://laurikari.net/tre/.

[3] Virtex 4 family overview, January 2007. Xilinx, Inc. http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

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

[5] F. Anjum, D. Subhadrabandhu, and S. Sarkar. Signature-based intrusion detection for wireless ad-hoc networks: A comparative study of various routing protocols. In *IEEE Vehicular Technology Conference*, October 2003.

[6] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. *INFOCOM 2007*, pages pp. 1064–1072, May 2007.

[7] Joao Bispo, Ioannis Sourdis, Joao M.P. Cardoso, and Stamatis Vassiliadis. Synthesis of regular expressions targeting fpgas: Current status and open issues. In *Int. Workshop on Applied Reconfigurable Computing (ARC 2007)*, pages 179–190, Mangaratiba, Brazil, March 2007.

[8] B. H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[10] B.C. Brodie, R.K. Cytron, and D.E. Taylor. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In *ISCA*, Boston, MA, June 2006.

[11] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781–1792, October 2006.

[12] S. Kumar, S. Dharmapurikar, P. Crowley, J. Turner, and F. Yu. Algorithms to accelerate multiple regular expression matching for deep packet inspection. In *SIGCOMM*, Pisa, Italy, September 2006.

[13] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of regular expression pattern matching circuits on fpga. In *DATE 2006*, Munich, Germany, 2006.

[14] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A fast string-matching algorithm for network processor-based intrusion detection systems. *ACM Transitions on Embedded Computing*, xx(12):614–633, 2004.

[15] V. Paxson. A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463–1792, December 1999.

[16] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *Proc. USENIX Hot Security*, August 2006.

[17] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *ACM Symposium on Architecture for Network and Communication Systems*, San Jose, CA, December 2006.

[18] L. Schaelicke, B. Moore T. Slabach, and C. Freeland. Characterizing the performance of network intrusion detection sensors. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003), LNCS, Springer-Verlag*, September 2003.

[19] Snort. http://www.snort.org/, 2003.

[20] L. Tan and T. Sherwood. Architectures for Bit-Split String Scanning in Intrusion Detection. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, January-February 2006.

[21] N. Weaver, V. Paxson, and J. M. Gonzalez. The shunt: An fpga-based accelerator for network intrusion prevention. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, Monterey, CA, 2007.

[22] S. Wu and U. Manber. Fast text searching: Allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[23] 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 *ACM Symposium on Architecture for Network and Communication Systems*, San Jose, CA, December 2006.