This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

Fast Online Set Intersection for Network Processing on FPGA

Yun R. Qu, *Member, IEEE,* and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—Online set intersection operations have been widely used in network processing tasks, such as Quality of Service differentiation, firewall processing, and packet/traffic classification. The major challenge for online set intersection is to sustain line-rate processing speed; accelerating set intersection using state-of-the-art hardware devices is of great interest to the research community. In this paper, we present a novel high-performance set intersection approach on FPGA. In our approach, each element in any set is represented by a combination of Group ID (GID) and Bit Stride (BS); all the sets are intersected using linear merge techniques and bitwise AND operations. We map our online set intersection algorithm onto hardware; this is done by constructing modular Processing Element (PE) and concatenating multiple PEs into a tree-based parallel architecture. In order to improve the throughput on a state-of-the-art FPGA, we feed all the inputs to FPGA in a streaming fashion with the help of the synchronization GIDs. Post place-and-route results show that, for a typical set intersection problem in network processing, our design can intersect 8 sets, each of up to 32 K elements, at a throughput of 47.4 Thousand Intersections Per Second (KIPS) and a latency of $94.8\,\mu$s per batch of inputs. Compared to the classic linear merge or bitwise AND techniques on state-of-the-art multi-core processors, our designs on FPGA achieves up to $66\times$ throughput improvement and $80\times$ latency reduction.

Index Terms—Set intersection, Network Processing, Field-Programmable Gate Array (FPGA).

1 Introduction

Set intersection is a key operation in many query processing tasks of databases. Meanwhile, due to the rapid growth of Internet, set intersections are also widely performed in a plethora of network processing tasks, including network security, packet classification, and traffic clustering. For example, packet classification [1]–[3] requires multiple fields of the packet headers to be examined. After searching all the fields of an incoming packet header, the candidate rule ID sets have to be intersected to produce the final classification result [1].

Performing set intersection in network processing faces two major challenges: the increasing size of the datasets, and the demand on line-rate processing. For example, the OpenFlow table lookup [4] in Software Defined Networking (SDN) may require up to 40 sets to be intersected. At the same time, the increasing bandwidth of the current Internet has evolved to a rate of over hundreds of gigabits per second [5]. These two factors pose great challenges on intersecting many sets during run-time.

State-of-the-art VLSI chips can be built with massive amount of on-chip computation and memory resources, as well as large number of I/O pins for off-chip memory accesses; Field Programmable Gate Arrays (FPGAs) [6], with their flexibility and reconfigurability, are especially suitable for accelerating network applications [7]. Efficient algorithms and parallel architectures are still to be explored on FPGA in order to achieve extremely high performance.

In this paper, we present a novel approach based on both *Linear Merge* (LM) and *Bitwise AND* (BA) techniques. Compared to prior works where only the LM technique or the BA technique is used, our hybrid approach is both memory-efficient and hardware-friendly. Specifically, our contributions in this paper include:

- We split all the elements in the same set into multiple groups; each group is assigned a Group ID (GID). We linearly merge all the GIDs from different sets in multiple clock cycles.
- We construct a Bit Stride (BS) for each group of elements. The BSs corresponding to the same GID are bitwise ANDed in a pipelined fashion to produce the final set intersection result.
- We prototype our design on a state-of-the-art FPGA device. We present various tradeoffs on design parameters; we compare the performance of our designs in this paper with software-based set intersection engines.
- We sustain 47.4 KIPS throughput when intersecting 8 sets, each of up to 32 K elements. Compared to the classic LM technique or BA technique on state-of-the-art multi-core processors, our approach achieves up to $66\times$ throughput improvement and $80\times$ latency reduction.

The rest of the paper is organized as follows: We introduce the background and related works in Section 2. We present our novel data structures and algorithms

- *The authors are with the Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: {yunqu, prasanna}@usc.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

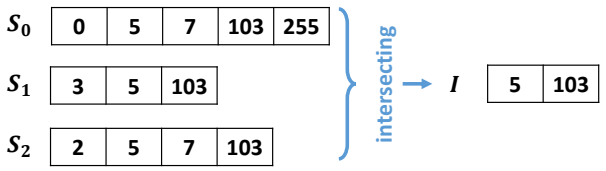

Fig. 1: An example of intersecting $M = 3$ sets, where all the elements are represented by IDs.

in Section 3. We implement our set intersection engine on FPGA in Section 4. We evaluate the performance in Section 5 and conclude the paper in Section 6.

2 Background

2.1 Notations

Set intersection is a well-known operation to select common elements in all the given sets. In this paper, we assume, without loss of generality, that the elements in each set are presorted in ascending order. We denote the number of sets to be intersected as M. We denote the number of elements in each set as N_m, where $m = 0, 1, \ldots, M-1$. We show an example of $M = 3$ in Figure 1, where $N_0 = 5$, $N_1 = 3$, and $N_2 = 4$. We denote the final intersected set as I; in the example shown in Figure 1, $I = \{5, 103\}$.

We use $\text{argmin}_m N_m$ as the index of the smallest set, where $\text{argmin}_m N_m \in \{0, 1, \ldots, M-1\}$. Similarly, $\text{argmax}_m N_m$ can be defined. Assuming the element IDs use natural numbers, we denote the largest element in any set as $(\Omega - 1)$. In Figure 1, $\text{argmin}_m N_m = 1$, $\text{argmax}_m N_m = 0$, and $\Omega = 256$.

2.2 Approaches

Set intersection has been widely studied in both database systems [8], [9] and network processing [10]–[12]. In general, there are two major categories for set intersection approaches: (1) the LM techniques, and (2) the BA techniques.

The classic LM technique requires the elements in each set to be represented by IDs, each of $\log \Omega$ bits; the common elements appearing in all the sets can be identified by iteratively checking the smallest/largest elements in all the sets [12]. This technique requires $O\left(\sum N_m \log \Omega\right)$ memory and $O\left(\sum N_m \log \Omega\right)$ time complexity.

An enhanced version [9] of the LM technique can be much more complex, where the elements in the smallest set are used to eliminate the candidates in all the other sets. The memory consumption for this enhanced version is $O\left(\sum N_m \log \Omega\right)$, while the time complexity for intersecting all the M sets is

$$O\left(\sum_{m \neq \text{argmin}_m N_m} \log\left(N_m \log \Omega\right)\right) \quad (1)$$

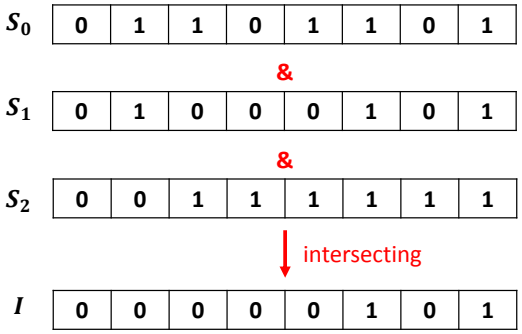

Fig. 2: An example of intersecting $M = 3$ sets, where all the sets are represented by BVs.

The major drawback of the LM techniques is that it is not easy to implement such algorithms in a streaming fashion on hardware. A single intersection on M sets introduces processing latency of multiple clock cycles.

The classic BA technique requires all the elements in the same set to be represented by a Bit Vector (BV); for a particular set, a bit is set to "1" only if the element appears at the corresponding position. For example, in Figure 2, $S_1 = \{0, 2, 6\}$; so starting from LSB, the 0th, 2nd, and 6th bits are all set to "1". The common elements in all the sets can be reported by ANDing the BVs of all the sets [10]. The memory consumption and the time complexity of this classic BA technique are both $O\left(M \cdot \Omega\right)$. This is much more expensive[1] compared to the classic LM technique, especially when the elements are "sparse" [11]. For example, the BV for any set in Figure 1 needs to be at least 256 bits.

To enhance the classic BA technique, several optimization techniques are proposed in [8], [11]. For example, multiple BVs can be folded by OR operations, where only the positions corresponding to non-zero bits are examined in the folded BV. However, neither the memory consumption nor the time complexity is reduced in the worst case. In general, the BA techniques are easy to implement on hardware, but they also consume a huge amount of memory.

2.3 Network Applications

Set intersection has a variety of applications in network processing. For example, in packet classification [1]–[3], a packet header may match different sets of rules in various fields, but only the rules, whose IDs appear in all the fields, are considered as matching the packet header. This is equivalent to intersecting all the matching rule IDs from all the fields.

Another application is the defense against Denial of Service (DoS) attacks [13] for network security. A router may see a lot of sources sending similar

1. Exponential with respect to the number of bits of each element.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

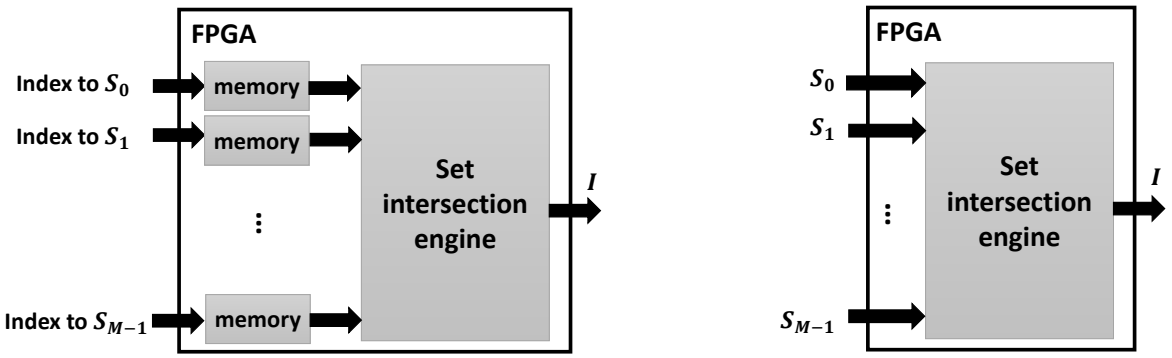

Fig. 3: Two types of set intersection operations: (Left side) The presorted sets are stored in memory; during run-time, only the indexes to the sets are provided to the hardware. (Right side) The elements of the sorted sets are fed from either external devices or other kernels on the same hardware, in a streaming fashion.

packets; each ingress port of the router may need to forward a set of packets to the same destination. In this case, performing set intersections on all the destinations from all the ingress ports is beneficial; it can report what destinations are under attack.

2.4 Our Focus

For real-time network processing, the major challenge of set intersection is the strict performance requirement. Sets have to be intersected at very high throughput to sustain line-rate processing. Design of high-performance online set intersection engine is the focus of this paper.

We assume the elements in each given set are presorted in ascending order[2]. Depending on the network application type, the sorted sets can be prepared during design-time [1], or provided as inputs during run-time [13]. This leads to two slightly different hardware implementations, as shown on the left side and right side of Figure 3, respectively. In this paper, since our focus is the high-performance set intersection engine, we assume all the elements of the sets are only known during run-time; hence we choose the implementation type as shown on the right hand side of Figure 3.

3 Data Structures and Algorithms

3.1 Motivations

As can be seen, the LM techniques are memory-efficient for sparse sets, while the BA techniques are easy for hardware implementation. This observation inspires us to exploit a hybrid data structure. The basic ideas are:

1) We split all the elements into multiple groups; each group in a set is assigned a unique Group ID (GID). All the sets are intersected on GIDs first, using the LM techniques.
2) We associate each GID with a shorter Bit Stride (BS); each bit in the BS corresponds to an element in a set. For different sets, only the BSs corresponding to the same GIDs are ANDed.

Since all the sets are represented in GIDs and BSs, we define this representation of data structures as *GID/BS representation*.

3.2 GID/BS representation

We denote the number of bits for a GID as g. We denote the number of bits in a BS as s. Based on the notations introduced in Section 2.1, each long BV in the classic BA technique can use up to Ω bits. To reduce the memory consumption, we split each BV into a total number of $\frac{\Omega}{s}$ groups, each of s bits. Hence each group corresponds to an s-bit BS. We assign a GID to a group. The GIDs are unique in each set, but the GIDs in different sets can be identical.

For instance, suppose there are three sets $S_0 = \{0, 2, 3, 5, 6, 7, 64, 65, 66\}$, $S_1 = \{4, 5, 7, 18, 124\}$, and $S_2 = \{7, 18\}$. Further suppose we have $\Omega = 128$, and we choose $s = 4$. The classic BA technique generates one 128-bit BV for each set, where most of the bits are zero. However, in our technique, we split them into groups of 4 bits each; this leads to 32 groups per set. Thus, the GID requires 5 bits each. We show the GIDs and BSs constructed for this example in Figure 4. As can be seen:

- We only keep the groups where the corresponding BSs are non-zero (at least one bit out of s bits is "1"). This leads to a significant amount of memory reduction on BVs, especially for very sparse sets.
- We use the same GIDs for different sets. For different sets, only the BSs corresponding to the same GID have to be ANDed.

Note $g = \log\lceil\frac{\Omega}{s}\rceil$. Denoting the number of GID/BS pairs stored for set m as G_m ($0 < G_m \leq N_m$), we have

2. This assumption is very common in many applications [9], [11]. Hence, the discussion of sorting all the elements in each set is beyond the scope of this paper.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

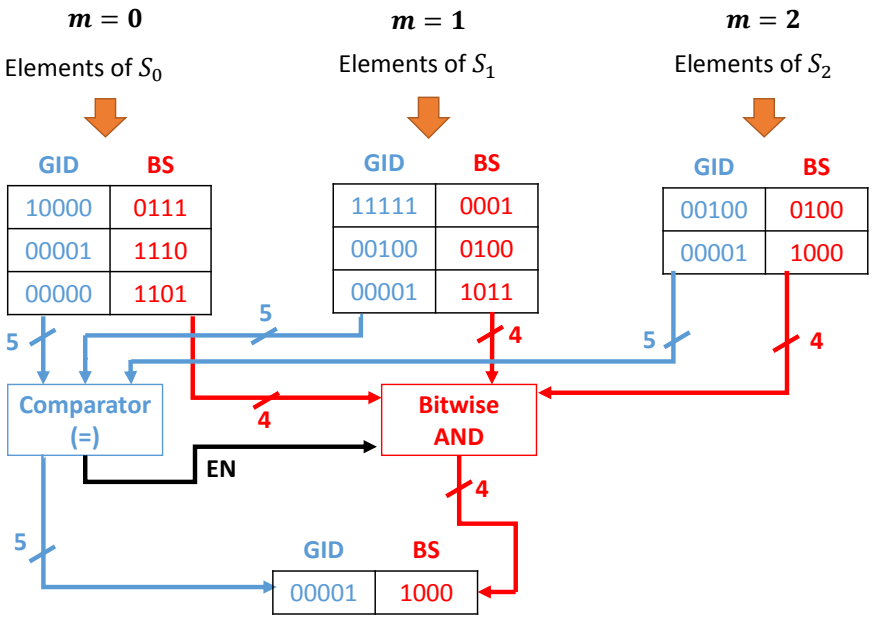

Fig. 4: Using GID and BS for set intersection. In this example, there are $M = 3$ sets, $g = 5$ bits per GID, and $s = 4$ bits per BS. After groups are intersected, an enable signal *EN* is generated to control the bitwise AND operations for the corresponding BSs.

the memory consumption for all the GIDs and BSs as

$$O\left(\sum_{m=0}^{M-1} G_m \left(\log\left\lceil\frac{\Omega}{s}\right\rceil + s\right)\right) \quad (2)$$

For $s = 1$, this memory requirement is the same as the classic LM technique. For $s = \Omega$, this memory requirement is the same as the classic BA technique. We will discuss the time complexity of our set intersection algorithm in Section 4. We will also determine the value of s later in Section 5.

3.3 Online Set Intersection

Our set intersection approach consists of two phases as follows:

1) *Preprocessing*: all the sets are preprocessed using the GID/BS representation. This phase can be done offline.
2) *Online Set Intersection*: all the GIDs are intersected for different sets; their corresponding BVs are bitwise ANDed.

As discussed in Section 2.4, we assume the elements in each given set are already sorted in ascending order, thus, we ignore the discussion of the preprocessing phase in this paper. We focus on the online set intersection phase in this section.

For set m, since there are G_m GID/BS pairs stored, we index them by $i_m = 0, 1, \ldots, G_m - 1$; the GID and BS corresponding to index i_m are denoted as GID$[m, i_m]$ and BS$[m, i_m]$, respectively. For example, in Figure 4, for set S_2, $G_2 = 2$, and $i_2 = 0, 1$. GID$[2, 0] = 00100$, BS$[2, 0] = 0100$, GID$[2, 1] = 00001$, and BS$[2, 1] = 1000$. Similarly, we assume the final intersected set I has P GID/BS pairs, indexed by $i = 0, 1, \ldots, P-1$; the GID and BS in set I are denoted as GID$[i]$ and BS$[i]$.

We show our online set intersection algorithm in Algorithm 1. Figure 4 shows an example of the corresponding architecture. The smallest GIDs in all the M sets are compared against each other, and the GIDs smaller than the maximum value (X, as denoted in Algorithm 1) are excluded from I. Only if all the smallest GIDs in all the sets are equal, have we identified a common GID in all the M sets; in this case, the corresponding BSs are ANDed. In other words, the GIDs are intersected using an LM-like technique, while the BSs corresponding to the same GIDs are intersected using bitwise AND operations.

4 Hardware Architecture

The architecture shown in Figure 4 is naive, because there are several drawbacks to be noticed:

1) The performance of comparing GIDs and bitwise ANDing BSs deteriorates as M increases; intersecting a large number of sets can lead to very slow clock rate.
2) Intersecting GIDs may require multiple clock cycles to complete; this degrades the overall throughput performance of the architecture.

In this section, we will improve the performance of our hardware architecture on FPGA using (1) *modular Processing Element* (PE) (Section 4.1), and (2) *tree-based parallel architecture* (Section 4.2). We present a streaming technique to feed different *batches* (see Section 4.3)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

Algorithm 1 Online Set Intersection

Input A total number of M sets S_m, $m = 0, 1, \ldots, M-1$, represented using the GID/BS representation.

Output An intersected set I, whose entries are indexed by i. $\forall i$, $\forall m$, $\text{GID}[i] = \text{GID}[m, i_m]$ for some i_m; $\text{BS}[i] = \text{BS}[0, i_0] \,\&\, \text{BS}[1, i_1] \,\&\, \ldots \,\&\, \text{BS}[M-1, i_{M-1}]$ where $\text{GID}[i] = \text{GID}[0, i_0] = \text{GID}[1, i_1] = \cdots = \text{GID}[M-1, i_{M-1}]$.

1: **for** $m = 0$ to $M-1$ **do**
2: $i_m \leftarrow 0$ {pointers initialization}
3: **end for**
4: $i \leftarrow 0$ {pointer for I}
5: **while** $(i_0 < G_0) \,||\, (i_1 < G_1) \,||\, \ldots \,||\, (i_{M-1} < G_{M-1})$ **do**
6: $X \leftarrow \text{GID}[0, i_0]$ {set current maximum}
7: **for** $m = 0$ to $M-1$ **do**
8: **if** $\text{GID}[m, i_m] > X$ **then**
9: $flag \leftarrow$ **false** {reset flag}
10: $X \leftarrow \text{GID}[m, i_m]$ {elements not equal}
11: **else if** $\text{GID}[m, i_m] < X$ **then**
12: $flag \leftarrow$ **false** {reset flag}
13: **if** $i_m < G_m$ **then**
14: $i_m \leftarrow i_m + 1$ {advance index}
15: **else**
16: go to Step 37
17: **end if**
18: **else**
19: **if** $m = 0$ **then**
20: $flag \leftarrow$ **true** {set flag for S_0}
21: **else if** $m = M-1$ **then**
22: **if** $flag =$ **true then**
23: $flag \leftarrow$ **false** {elements equal}
24: $\text{GID}[i] \leftarrow \text{GID}[0, i_0]$
25: $\text{BS}[i] \leftarrow \text{BS}[0, i_0]$
26: $i_0 \leftarrow i_0 + 1$ {advance index}
27: **for** $m' = 1$ to $M-1$ **do**
28: $\text{BS}[i] \leftarrow \text{BS}[i] \,\&\, \text{BS}[m', i_{m'}]$
29: $i_{m'} \leftarrow i_{m'} + 1$ {advance index}
30: **end for**
31: $i \leftarrow i + 1$
32: **end if**
33: **end if**
34: **end if**
35: **end for**
36: **end while**
37: report I consisting of $\text{GID}[i]$, $\text{BS}[i]$, where $i = 0, 1, \ldots, P-1$

of data back-to-back; our intention is to achieve very high throughput by minimizing the communication overheads between different batches of data.

4.1 Modular PE

We show the internal organization of a modular PE in Figure 5. A modular PE takes GIDs and BSs from two ordered sets; the GIDs and BSs are fed in ascending order into the PE. The basic operations of a modular PE consist of reporting common GIDs, and performing bitwise AND operations on the corresponding BSs. Specifically, a modular PE contains the following components: 2 FIFOs, one g-bit comparator, and one s-bit bitwise AND gate.

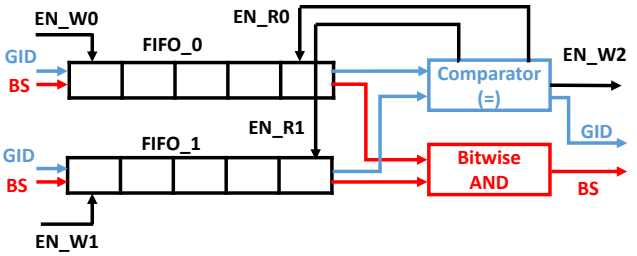

Fig. 5: Internal organization of a modular PE. The data width of any GID is g bits. The data width of any BS is s bits. The data width of any control signal (*e.g.*, *EN_W0*, *etc.*) is 1 bit. Other FIFO control signals are omitted for simplicity, *e.g.*, *not_full*, *not_empty*, *etc.*

TABLE 1: Truth table for the control signals (assuming neither of the FIFO is empty)

Case	Equal GIDs	GID in FIFO_0 is smaller	GID in FIFO_1 is smaller
EN_R0	1	1	0
EN_R1	1	0	1
EN_W2	1	0	0

4.1.1 2 FIFOs

The FIFOs are used to buffer the GID/BS pairs. The write enable signals *EN_W0* and *EN_W1* are fed from the inputs. The read enable signals *EN_R0* and *EN_R1* are generated internally by the comparator.

4.1.2 g-bit Comparator

The comparator compares two GIDs, each of g bits. Based on the comparison result, the comparator generates 3 control signals:

1) On the one hand, if two GIDs are identical, the comparator sets *EN_W2* = 1 for the next PE to accept this GID and the corresponding ANDed BS. To compare the next two GIDs, both *EN_R0* and *EN_R1* are set to 1 for the two FIFOs.
2) On the other hand, if two GIDs are not equal, the comparator sets *EN_W2* = 0. Since we only keep track of the maximum value of the smallest elements in two sets, out of the two GIDs, the smaller one is discarded; hence, only one of *EN_R0* and *EN_R1* is set to 1 for the corresponding FIFO to provide the next GID.

We summarize the values of the control signals generated by the g-bit comparator in Table 1.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

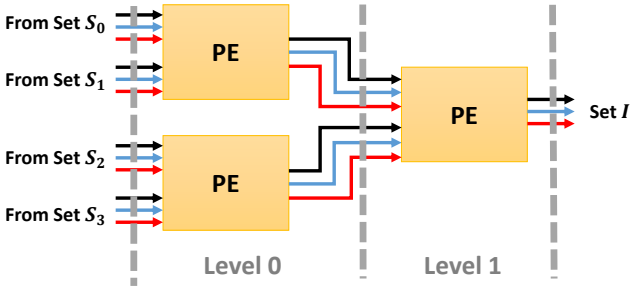

Fig. 6: An example of tree-based architecture: $M = 4$ sets are intersected using 2 levels of PEs.

4.1.3 s-bit Bitwise AND Gate

The bitwise AND gate perform bitwise AND operations on two BSs, each of s bits. The result is an ANDed BS, consisting of s bits. Although the bitwise AND gate produces results for any two compared GIDs, the control signal *EN_W2* decides whether the result produced by this gate should be accepted by the next PE. The ANDed BS is only accepted when the two GIDs compared are identical, as shown in Table 1.

4.2 Tree-based Parallel Architecture

The modular PE discussed in Section 4.1 only intersects 2 sets. To intersect a large number of sets, multiple modular PEs have to be used. Also, to reduce the processing latency, parallel architectures have to be exploited. Our intuition in this paper is to intersect M sets iteratively, two sets at a time using the modular PE in Section 4.1.

For M sets, we choose to deploy $\log M$ levels of PEs; for instance, $M = 4$ in Figure 6, so 2 levels of PEs are deployed. In our notations, level 0 always consists of all the leaves of the tree, where level $(\log M - 1)$ consists of only the root of the tree. We denote this architecture as *tree-based parallel architecture*, because (1) all the PEs are connected in a tree-like fashion, and (2) all the PEs at the same level perform set intersections in parallel.

In our architecture, we notice that the size of the intersection of any two sets is no greater than the size of the smaller set; as we go towards the root, smaller and smaller FIFOs can be used. The clock rate supported by the PE at the root is no slower than the clock rates supported by the PEs at the leaves.

The naive architecture shown in Figure 4 leads to a time complexity (or processing latency) of

$$O\left(\sum_{m=0}^{M-1} G_m\left(\log\left\lceil\frac{\Omega}{s}\right\rceil + s\right)\right) \sim O\left(M \cdot \max_m[G_m]\left(\log\left\lceil\frac{\Omega}{s}\right\rceil + s\right)\right) \quad (3)$$

where G_m $(0 < G_m \leq N_m)$ denotes the number of GID/BS pairs stored for set m.

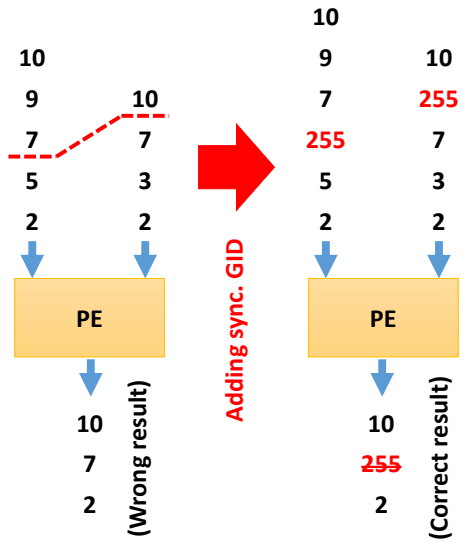

Fig. 7: Adding synchronization GIDs, where $g = 8$ and $M = 2$. Red numbers denote synchronization GIDs, while black numbers denote regular GIDs.

However, using our tree-based parallel architecture introduced in this section, we can intersect M sets with a (parallel) time complexity of

$$O\left(\left(\log M\right) \cdot \max_m[G_m] \cdot \left(\log\left\lceil\frac{\Omega}{s}\right\rceil + s\right)\right) \quad (4)$$

Note that this upperbound is quite a loose upperbound. As discussed, this is because the number of common elements in two sets is no more than the number of elements in the smaller set; we have (possibly) smaller and smaller sets to be merged linearly as we go down towards the tree root.

4.3 Streaming Inputs

Our architecture can intersect M sets at a time. We denote such M sets to be intersected as a *batch*. For example, in Figure 7, suppose a batch of two sets, consisting of GIDs $\{2, 5\}$ and GIDs $\{2, 3, 7\}$, are to be intersected; another batch of two sets, consisting of GIDs $\{7, 9, 10\}$ and GID $\{10\}$, are to be intersected. Using a single modular PE, we need to generate the correct results consisting of GID $\{2\}$ and GID $\{10\}$ sequentially.

In our architecture, all the GID/BS pairs can be streamed in; this means the time for getting GIDs from different batches can be overlapped. This benefits the throughput performance. However, the inputs from different batches need to be distinguished to avoid any confusion; otherwise the intersected result can be wrong. Continuing the example discussed above, we show the wrong results generated from two batches of inputs on the left side of Figure 7.

We employ *synchronization GID* to separate GIDs from different batches. As opposed to regular GIDs, a synchronization GID is a GID with all of its g

bits set to 1. A synchronization GID is forbidden in the input; meanwhile, all of the synchronization GIDs in the final output[3] are discarded. Continuing the example discussed in this subsection, we show how we generate the correct intersected sets using synchronization GIDs in Figure 7. As can be seen in this figure, since $g = 8$, we add the synchronization GID 255 immediately after the end of the first batch; note that the same synchronization GID must be added to all the M sets (in this example, $M = 2$). The synchronization GID is an overhead for streaming inputs:

- *Time overhead*: it takes 1 extra clock cycle to synchronize all the sets of the same batch.
- *Resource overhead*: the synchronization GID uses g bits itself; also, the corresponding s-bit BS cannot be utilized for this GID.

We add synchronization GIDs during the preprocessing phase; thus, the synchronization GIDs are streamed in just like all the other regular GIDs.

5 Evaluation

We organized this section as follows:

- In Section 5.1, we introduce the setups of our experiments.
- In Section 5.2, we determine the values of g and b by investigating their effect on the hardware performance.
- In Section 5.3, we examine the impact of various values of Ω on the performance with respect to throughput, latency, resource utilization, and power.
- In Section 5.4, we examine the impact of various values of M on the performance with respect to throughput, latency, resource utilization, and power.
- In Section 5.5, we evaluate the performance of our set intersection engine using real-life datasets.
- In Section 5.6, we compare our work with prior works on various platforms.

5.1 Experimental Setup

5.1.1 Hardware and Software

We conducted experiments on the state-of-the-art Xilinx Virtex 7 FPGA (XC7VX1140T-FLG1930 -2L) [6]. This FPGA has 218800 logic slices, 1100 I/O pins, and 68 Mb (1880 blocks) BRAM; it can be configured to realize a large amount of distributed RAM (distRAM, up to 18 Mb). Our design does not depend on the type of the FPGA, as long as there are enough hardware resources available.

To simplify our designs, we instantiated all the memory modules (*e.g.*, FIFO) using single-port distRAM or BRAM. We evaluated the performance using Xilinx Vivado 2014.2 design tool [14]. A conservative clock constraint of 250 MHz was used for all of our designs; Vivado TCL scripts were used to automate the design space exploration.

5.1.2 Performance Metrics

The following performance metrics were considered in our experiments:

- **Throughput**: the number of intersected sets (I) produced per unit time (in KIPS). We recorded the throughput values based on the clock rates from the post-place-and-route timing reports.
- **Latency**: the processing time required for intersecting M sets of the same batch. We reported the latency values based on simulation results.
- **Resource Utilization**: the percentages of basic FPGA resources utilized. We investigated (1) logic slice utilization, (2) BRAM utilization, and (3) I/O pin utilization, based on the post-place-and-route resource utilization reports.
- **Power**: the power consumption of an entire design on FPGA, including both static power and dynamic power. We fixed the temperature at $25\,^{\circ}$C. We used Switching Activity Interchange Format (SAIF) files as inputs to Vivado power analysis tool.

Throughput and resource utilization are very commonly used for most FPGA-based implementations [7], [15]. Latency has regained much attention recently in SDN [4]. Power is a very important metric, especially for large data centers and wireless networks [16], [17].

In addition, for throughput, we further defined:

- **Peak throughput** (T_{peak}): the throughput determined by the hardware architecture for a given set of design parameters (*e.g.*, M, g, *etc.*). When calculating the peak throughput, we assume the FIFOs in the PEs are full for worst-case analysis.
- **Sustained throughput** ($T_{sustained}$): the throughput measured for a specific data trace. The sustained throughput varies during run-time, because the number of GID/BS pairs buffered in the FIFOs depends on the data trace.

The sustained throughput is hard to measure; we defer the discussion of the sustained throughput until later sections. For a given design, the peak throughput mainly depends on (1) the clock rate achievable on FPGA and (2) the size of the largest set to be intersected. Let f denote the maximum frequency achievable for a design. Considering the time overhead on synchronization GIDs, we have:

$$T_{peak} = \frac{f}{\max_m[G_m] + 1} \tag{5}$$

5.1.3 Datasets

We conducted extensive experiments on the real datasets from the classic 5-field packet classification

3. Only at the root level of the tree-based architecture.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

TABLE 2: Performance with respect to g and b ($M = 4$)

g \ b		2	4	8	16	32	64
2	Clock rate (MHz)	486.85	441.31	375.66	379.51	383.58	312.50
	Logic slices (%)	0.01	0.01	0.02	0.02	0.04	0.07
	BRAM (%)	0.00	0.00	0.00	0.00	0.00	0.00
	I/O pins (%)	2.90	3.81	5.63	9.27	16.54	31.09
4	Clock rate (MHz)	321.13	330.58	311.72	306.00	328.19	269.47
	Logic slices (%)	0.02	0.03	0.03	0.04	0.05	0.07
	BRAM (%)	0.00	0.00	0.00	0.00	0.00	0.31
	I/O pins (%)	3.81	4.72	6.54	10.18	17.45	32.00
6	Clock rate (MHz)	305.53	304.51	329.60	299.04	290.87	255.43
	Logic slices (%)	0.04	0.05	0.05	0.05	0.06	0.08
	BRAM (%)	0.00	0.00	0.00	0.15	0.15	0.31
	I/O pins (%)	4.72	5.63	7.45	11.09	18.36	32.90
8	Clock rate (MHz)	299.31	297.53	299.31	292.57	286.20	306.37
	Logic slices (%)	0.11	0.09	0.09	0.10	0.10	0.12
	BRAM (%)	0.00	0.15	0.15	0.15	0.15	0.31
	I/O pins (%)	5.63	6.54	8.36	12.00	19.27	33.81
10	Clock rate (MHz)	265.96	287.27	282.17	276.24	264.55	268.89
	Logic slices (%)	0.27	0.28	0.28	0.28	0.29	0.30
	BRAM (%)	0.15	0.15	0.15	0.15	0.31	0.63
	I/O pins (%)	6.54	7.45	9.27	12.90	20.18	34.72
12	Clock rate (MHz)	257.86	259.13	259.07	258.13	258.26	257.27
	Logic slices (%)	1.04	1.05	1.05	1.06	1.08	1.10
	BRAM (%)	0.15	0.15	0.31	0.63	1.27	2.39
	I/O pins (%)	7.45	8.36	10.18	13.81	21.09	35.63

problem [1], due to the availability of the rule sets and the packet traces [18]. Assuming all the 5 sets from the same packet header had already been obtained, we only focused on intersecting $M = 5$ sets in this paper[4]. In order to make all the implementations "modular", for $M = 5$, we designed our intersection engines to take input data from at most 8 sets concurrently; *i.e.*, all the values of M were rounded up to the nearest power of 2.

For real-life datasets in the 5-field packet classification, the largest real-life rule set, to the best of our knowledge, had $32\,\mathrm{K}$ rules [18]; *i.e.*, $\Omega = 32\,\mathrm{K}$ in this case. To measure the sustained throughput, we categorized different packet traces [18] based on the values of $\max_m \left[G_m\right]$. We conduct 10 runs (as examples) for each category; each run performs $10\,\mathrm{K}$ set intersections.

To investigate the sustained throughput, we defined *selectivity* (denoted as η) to be the ratio of the size of the intersection to the size of the largest set to be intersected:

$$\eta = \frac{|I|}{\max_m[G_m]} \tag{6}$$

As can be seen later, η has significant impact on the throughput and latency performance[5].

4. The packet classification problem also involves searching all the fields to get all the sets before intersecting all the sets.

5. In [9], selectivity was defined to be the ratio of the size of the intersection to the size of the smallest set. However, they are very similar definitions and have similar impact on the performance.

5.2 Determining Parameters

There are many ways to determine the values of g and b. For instance, our approach exploits the LM techniques, which is a data-dependent algorithm. For a specific data trace, there may exist an "optimal" combination of g and b, which gives the highest throughput or lowest processing latency. However, at the design time, we usually don't know the statistics of the input data; the input data can also be purely random. In such cases, it is impossible to always use the "optimal" values of g and b. In this paper, we assume very little information on the input data is known at the design time; thus, we determine the values of g and b based on the hardware performance.

5.2.1 FIFO depth

The maximum value of G_m is no greater than $(2^g - 1)$, because for any set m, the maximum number of possible g-bit GIDs is $(2^g - 1)$, excluding the synchronization GID as discussed in Section 4.3.

There is no direct relation between the FIFO depth and the values of G_m. For simplicity, we use a FIFO depth greater than $(2^g - 1)$; this ensures that there is no data drop for the same batch of streaming inputs. In the tree-based parallel architecture as introduced in Section 4.2, the FIFOs in the PEs at various levels may require different FIFO depths; however, in this paper, we simply use the same FIFO depth for all the levels.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

To summarize, the relationship between the FIFO depth, the value of g, and the value of G_m, $m = 0, 1, \ldots, M-1$ in this paper can be described as:

$$\text{FIFO depth} > 2^g - 1 \geq \max_m \left[G_m\right] \quad (7)$$

When conducting experiments, we always follow the relation indicated in Equation 7 in this paper.

5.2.2 g and b

To determine the values of g and b, we first fix $M = 4$ as an example; similar trends can be seen for other values of M. We fix the depth of all the FIFOs in the modular PEs to be 2^g; thus, large values of g result in deep FIFOs. We show the clock rate achieved by our design and the corresponding resource consumption in Table 2. As can be seen:

- Since both M and the FIFO depth are small, our designs utilize very small amounts of logic and memory resources.
- As the values of g increases, the clock rate usually degrades. Since the memory resources (distRAM and BRAM) on FPGA are organized in modules, deep FIFOs require a large number of modules to be connected by long wires.
- As the values of b increases, the clock rate also degrades. Since each PE in our designs performs AND operations in every clock cycle, it requires longer clock period to AND wide BSs.
- As the values of g or b increases, there are very few cases where the clock rate varies. The small variations are caused by the design suite.

As can be seen in Table 2, the best clock rate is achieved at $b = 4$ or $b = 8$ in most cases; this is because for $b = 4$ or $b = 8$, very short BSs are ANDed in each PE, resulting in compact circuits and short wire lengths. Hence we tend to use small values of b in all of our experiments. Recalling Section 3.2, we have $g = \log\lceil \frac{\Omega}{s} \rceil$; therefore we choose the values of g based on both values of b and Ω.

5.2.3 Case Study

Let us study the case where $\Omega = 32\,\text{K}$ and $M = 8$ as an example; we follow the same design methodology for other values of g, b, Ω, and M in this paper. Since we tend to use small values of b, we restrict b to be 2, 4, 8, and 16. The corresponding values of g are 11, 12, 13, and 14, respectively. Under these configurations, we show the performance with respect to the clock rate and the resource utilization in Table 3. As can be seen, the best clock rate is achieved when $g = 12$ and $b = 8$; this matches our conclusion in Section 5.2.2 that the best clock rate is achieved when $b = 4$ or $b = 8$.

Note in Table 3, as the value of g increases, there are variations with respect to the utilization of logic slices and BRAM. This is because we do not put any restrictions on the memory type (distRAM or BRAM) of the FIFOs; instead, we relay on the Vivado design suite to choose the memory type for best performance. A simple calculation reveals that as g increases, the total memory consumption still increases.

TABLE 3: Performance for various combinations of g and b, where $\Omega = 32\,\text{K}$, $M = 8$

g	11	12	13	14
b	16	8	4	2
Clock (MHz)	222.52	261.23	196.19	180.54
slices (%)	0.80	2.50	5.72	0.25
BRAM (%)	1.06	0.74	0.74	5.95
I/O (%)	23.90	18.18	15.72	14.90

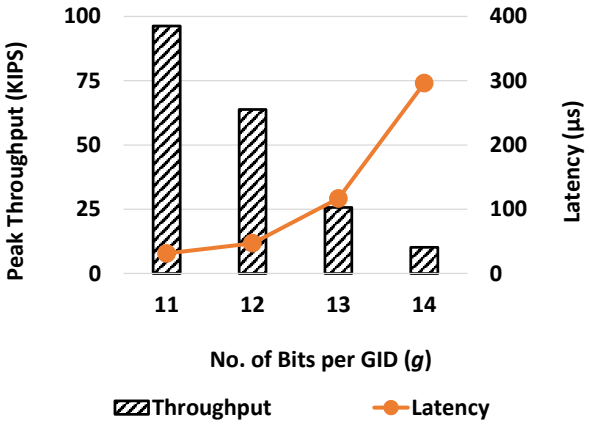

Fig. 8: Peak throughput for $b = 8$, and $M = 8$

5.3 Varying Ω

5.3.1 Throughput and Latency

Using $b = 8$ and $M = 8$ as our configuration, we show the peak throughput and the corresponding latency with respect to various values of g in Figure 8. For $b = 8$ and $g = 11, 12, 13, 14$, the corresponding values of Ω are 16 K, 32 K, 64 K, and 128 K, respectively; these values are sufficiently large for network applications [12]. As the value of g increases, the FIFO depth increases exponentially; the peak throughput tapers while the latency increases dramatically. The reason is that our set intersection approach still employs the LM techniques, whose time complexity is linear with respect to $\max_m \left[G_m\right]$ (or 2^g in this paper, because of Equation 7).

5.3.2 Resource Utilization

In Figure 9, we show the corresponding resource utilization with respect to (1) the logic slices, (2) BRAM, and (3) I/O pins on FPGA. As can be seen, the I/O pin utilization increases slightly as g increases; this matches our intuition because each input GID/BS pair requires $(g + b)$ input pins. There are variations with respect to the logic slice utilization and BRAM utilization; this also matches our observation discussed in Section 5.2.3.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

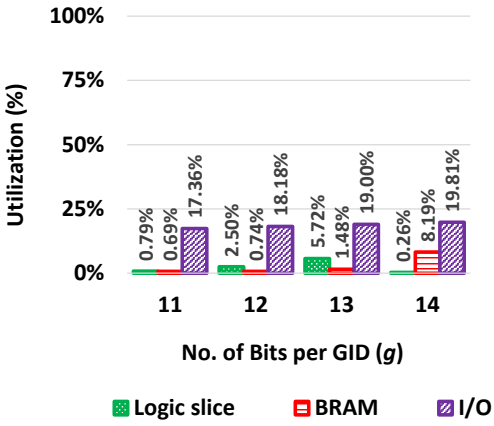

Fig. 9: Resource utilization for $b = 8$, and $M = 8$

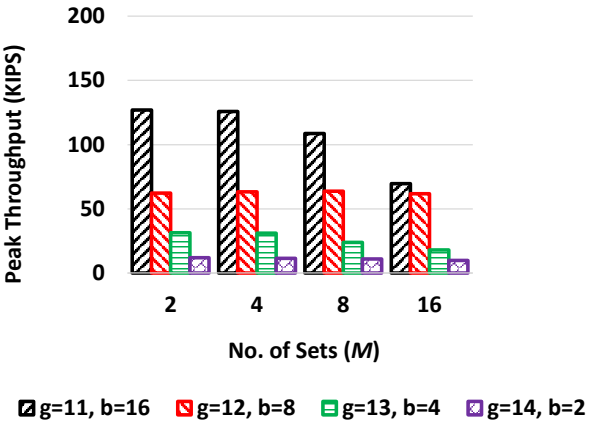

Fig. 11: Peak throughput for $\Omega = 32\,\mathrm{K}$

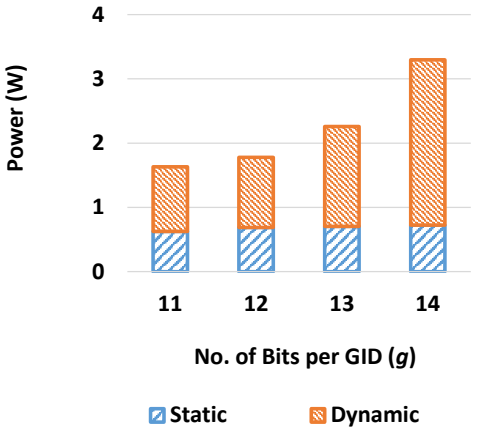

Fig. 10: Power consumption for $b = 8$, and $M = 8$

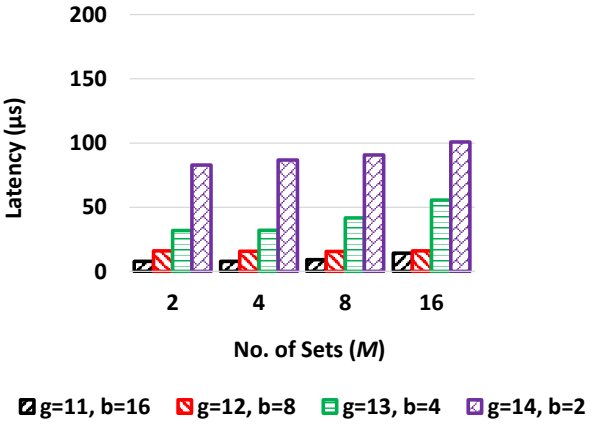

Fig. 12: Worst-case latency for $\Omega = 32\,\mathrm{K}$

5.3.3 Power Consumption

In Figure 10, we show the corresponding power consumption. As can be seen, the static power consumption varies little while the dynamic power consumption increases as g increases.

The trends shown in Figure 8, Figure 9, and Figure 10 can be observed for other combinations of b and M as well. Again, most of our designs on FPGA only consume very few logic slices, which is consistent with the results shown in Table 2. This is an advantage because the remaining logic slices can be used to implement other database kernels besides set intersection.

5.4 Varying M

5.4.1 Throughput and Latency

To examine the effect of M on the performance, we still use $\Omega = 32\,\mathrm{K}$ as an example, although similar trends can be seen for other values of Ω as well. Varying M, we show the peak throughput and the "worst-case" latency in Figure 11, and Figure 12, respectively.

As can be seen in Figure 11 and Figure 12, as M increases, the peak throughput and the worst-case latency deteriorate; this is because the clock rate degrades as M increases. For larger values of M, more resources are utilized, leading to less routing choices, longer wire lengths, and slower clock rates (see Section 5.4.2).

We have two important observations in Figure 11 and Figure 12:

- The peak throughput and the worst-case latency are dominated by the largest size of the sets to be intersected (2^g).
- M only has limited impact on the performance, especially when 2^g is large.

As can be seen in Equation 4, in each FIFO, all the 2^g GID/BS pairs buffered have to be checked in the worst case, leading to a time complexity of $O(2^g)$. This explains why the peak throughput is halved and the worst-case latency is doubled each time as g increases. This matches our intuition in Equation 4: our tree-based parallel architecture is in favor of intersecting a large number of small sets rather than intersecting very few large sets.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

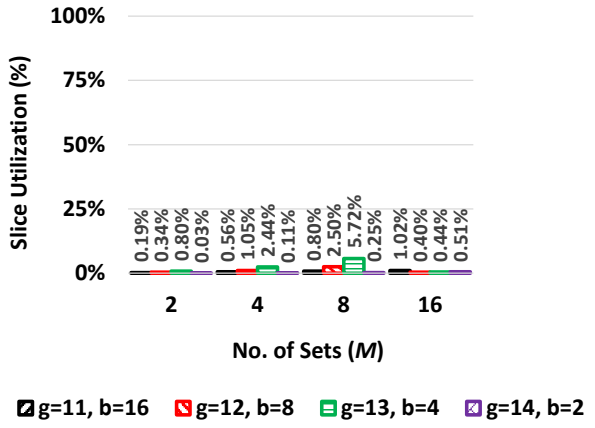

Fig. 13: Utilization of logic slices for $\Omega = 32\,\mathrm{K}$

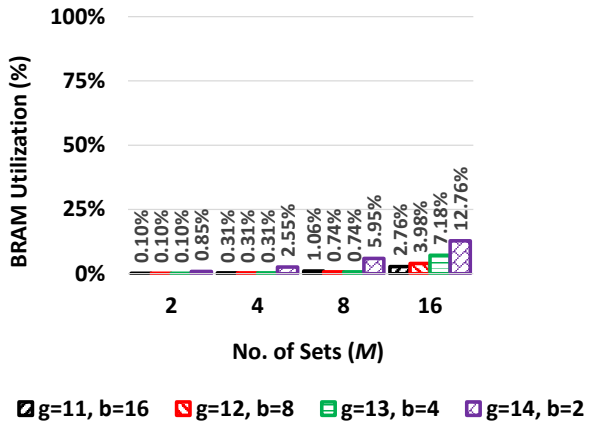

Fig. 14: Utilization of BRAM for $\Omega = 32\,\mathrm{K}$

5.4.2 Resource Utilization

In Figure 13 and Figure 14, we can see that the total memory utilization increases with respect to M, in spite of the variations with respect to the logic slice utilization or the BRAM utilization only. The reasons are the same as discussed in Section 5.2.3.

Figure 15 show that, the total number of I/O pins available on FPGA bottlenecks the scalability of our design, since intersecting a large number of M sets requires a large number of $O(M)$ parallel input pins to be used.

5.4.3 Power Consumption

We show the static power and dynamic power for $\Omega = 32\,\mathrm{K}$ in Figure 16, with respect to various combinations of g and b. As can be seen:

- As g increases, the dynamic power consumed by our designs increases linearly with respect to the total memory consumption.
- As M increases, the dynamic power consumed by our designs also increases linearly with respect to the total memory consumption.
- As g or M increases, the static power consumption only increases slightly.

Fig. 15: Utilization of I/O pins for $\Omega = 32\,\mathrm{K}$

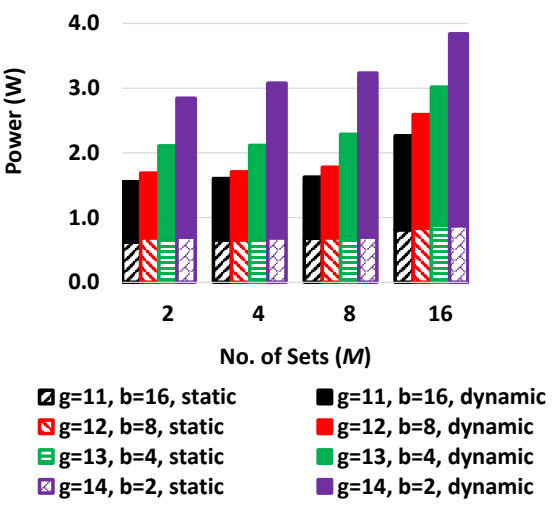

Fig. 16: Power consumption for $\Omega = 32\,\mathrm{K}$

Hence we observe that the total power consumption is almost linear with respect to the total memory utilized. This observation matches our intuition that the memory power dominates the total power consumption.

5.5 Real Datasets

In this subsection, we use the real-life datasets in the 5-field packet classification to test our online set intersection engines, as introduced in Section 5.1.3.

5.5.1 Throughput and Latency

For a batch of M sets, the set intersection is not considered as complete unless all the GIDs have been examined (G_m GIDs for set m). In our tree-based parallel architecture, the sustained throughput is lowerbounded by the throughput achieved at level 0 of the tree. We have

$$T_{sustained} \geq \frac{f}{2 \cdot \max_m [G_m]} \tag{8}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

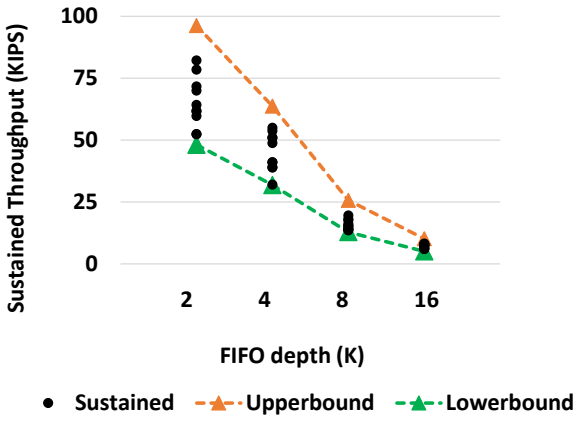

Fig. 17: Sustained throughput for the classic 5-field packet classification ($b = 8$, $M = 5$)

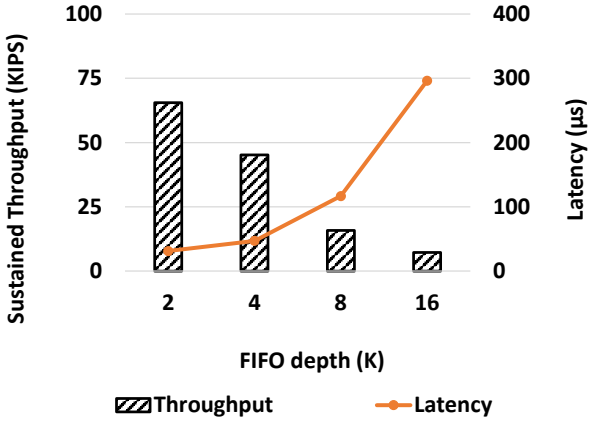

Fig. 18: Average sustained throughput and latency for the classic 5-field packet classification ($b = 8$, $M = 5$)

Besides, the sustained throughput is upperbounded by the peak throughput. Hence:

$$\frac{f}{2 \cdot \max_m[G_m]} \leq T_{sustained} \leq \frac{f}{\max_m[G_m] + 1} \qquad (9)$$

We show the sustained throughput with respect to various FIFO depths (from 2 K to 16 K) in Figure 17. We indicate in this figure both the lowerbound and upperbound of the sustained throughput based on Equation 9. For each FIFO depth, we show the sustained throughput for 10 runs, each run corresponding to 10 K set intersections, as introduced in Section 5.1.3.

For each FIFO depth (10 runs), we show the average sustained throughput and the corresponding average latency in Figure 18. As $\max_m [G_m]$ increases, both the throughput and the latency deteriorate. Since it takes linear time to merge all the GIDs in our approach, the performance with respect to throughput and latency is adversely affected by $\max_m [G_m]$.

5.5.2 Resource Utilization and Power

For real datasets, the performance with respect to resource utilization and power consumption are consistent with Figure 9 and Figure 10:

- The logic slice utilization increases linearly as the FIFO depth increases; the total resource utilization is always kept under 25%.
- Our designs only consume a small amount of power, due to the low resource utilization.

In spite of the same power performance as Figure 10, the energy performance on real datasets may vary; this is because different datasets can introduce various values of processing latency.

5.6 Comparison with Prior Works

5.6.1 Baseline

To the best of our knowledge, we are not aware of online set intersection engines on FPGA. Hence, to compare this paper with prior works, we deployed software-based set intersection engines on state-of-the-art multi-core General-Purpose Processors (GPPs) as the *baseline implementations*. We conducted experiments on a 2× AMD Opteron 6278 processor [19] and a 2× Intel Xeon E5-2470 processor [20]. The AMD processor has 16 physical cores, each running at 2.4 GHz. Each core is integrated with a 16 KB L1 data cache, 16 KB L1 instruction cache, and a 2 MB L2 cache. A 6 MB L3 cache (Last-Level Cache, LLC) is shared among all the 16 cores; all the cores have access to 64 GB DDR3-1600 main memory. The AMD processor runs openSUSE 12.2 OS (64-bit 2.6.35 Linux Kernel, gcc version 4.7.1). The Intel processor also has 16 physical cores, each running at 2.3 GHz. Each core has a 32 KB L1 data cache, 32 KB L1 instruction cache, and a 256 KB L2 cache. All the 16 cores share a 20 MB L3 cache (Last-Level Cache, LLC), and they have access to 48 GB DDR3-1600 main memory. This processor runs openSUSE 12.3 OS (64-bit 3.7.10 Linux Kernel, gcc version 4.7.2). Both of the AMD and the Intel processors have 32 logical cores.

On each GPP platform, we implemented the classic LM technique [9], [12] and the classic BA technique [8], [10], [11] using OpenMP [21]. We assumed our hybrid approach (using the GID/BS representation) could perform at least as good as the better of the two techniques; hence we ignored the implementation of our hybrid approach on the GPP platforms. Our implementations relied on the OS to allocate the hardware resources to each thread dynamically. The thread synchronization and IO overhead were also considered in the performance measurement.

5.6.2 Throughput and Latency

Setting $M = 8$ and $\Omega = 32$ K as an example, we compare the throughput and latency performance of this work with the baseline implementations in Figure 19 and Figure 20, respectively. Our GID/BS-based

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2537818, IEEE Transactions on Parallel and Distributed Systems

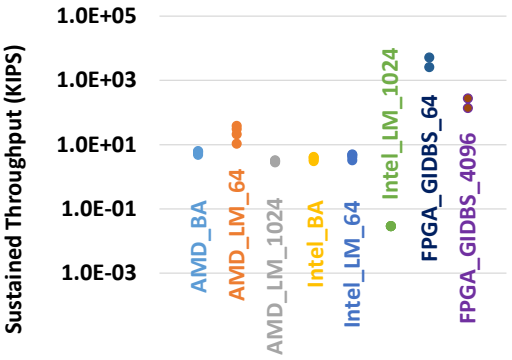

Fig. 19: Comparing sustained throughput

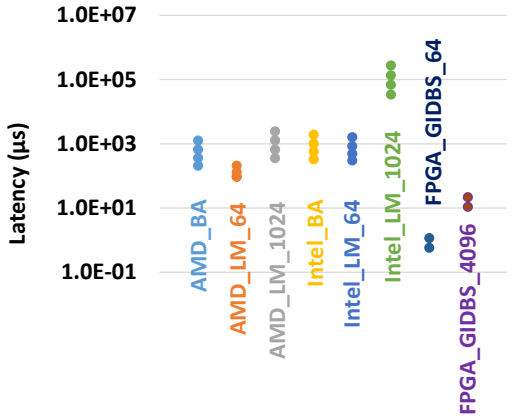

Fig. 20: Comparing latency

implementations have $b = 8$ in this example. Similar trends can be seen for other combinations of these parameters.

In these figures, each data label indicates the platform used, the approach exploited, and $\max_m \left[N_m\right]$ tested (the maximum size of all the M sets). For instance, "AMD_LM_64" denotes the implementation using the LM technique on the AMD platform, with $\max_m \left[N_m\right] = 64$. There are two exceptions:

1) The performance of the BA technique does not depend $\max_m \left[N_m\right]$, so we ignore $\max_m \left[N_m\right]$ in the corresponding implementations.
2) For our GID/BS-based designs on FPGA, the data labels indicate $\max_m \left[G_m\right]$. Note that:

$$\max_m \left[G_m\right] \leq \max_m \left[N_m\right] \leq b \cdot \max_m \left[G_m\right] \quad (10)$$

With $b = 8$ and $\max_m \left[G_m\right] = 4096$, our GID/BS-based design can intersect 8 sets with $32\,\mathrm{K}$ elements per set.

For each of our baseline implementations, we measure the performance with respect to various numbers of batches processed concurrently (1, 2, 4, or 8 concurrent batches). Increasing the number of concurrent batches improves the throughput but degrades the latency. For each of our GID/BS-based designs, we show the lowerbound and the upperbound of the throughput indicated by Equation 9 in Figure 19; we show their corresponding latency values in Figure 20.

We have the following observations:

- The BA technique performs better than the LM technique when the sets are relatively large, but worse when the sets are sparse ($\max_m \left[N_m\right] \ll \Omega$).
- For the same configuration, our AMD platform outperforms our Intel platform, due to its larger L2 (on-chip) cache size and higher clock rate.
- Compared to the baseline implementations, our GID/BS-based designs on FPGA sustain higher throughput (up to $66\times$ improvement) at lower processing latency (up to $80\times$ reduction).

As can be seen, our GID/BS-based designs presented in this paper demonstrate superior performance for online set intersection. The reasons are: (1) Our approach in this work exploits a hybrid data structure that performs at least as good as the better of the LM and BA techniques. (2) Our implementations are deployed on FPGA for better streaming performance compared with the GPP platforms.

6 Conclusion

In this paper, we presented a high-performance online set intersection engines on FPGA. The designs were based on a hybrid data structure combining the advantages of the LM and BA techniques. Compared to the classic LM and BA techniques on multi-core platforms, our prototypes on a stand-alone FPGA demonstrated superior performance with respect to throughput and latency.

A future direction towards online set intersection is to explore even more hybrid data structures (*e.g.*, hashing) for fast streaming applications. We also plan to target this problem using heterogeneous systems in the future; for example, it is also interesting to investigate the performance tradeoffs between the Processing System (PS) and the Programmable Logic (PL), when a Zynq-based system is deployed [22].

Acknowledgments

This work is supported by the U.S. National Science Foundation (NSF) under grants No. CCF-1116781 and No. CCF-1320211. Equipment grant from Xilinx is gratefully acknowledged.

References

[1] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.
[2] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient Multimatch Packet Classification and Lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, 2005.
[3] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "HEXA: Compact Data Structures for Faster Packet Processing," in *Proc. IEEE ICNP*, 2007, pp. 246–255.

[4] "OpenFlow Switch Specification V1.3.1," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.
[5] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification," in *Hot Interconnects*, 2012, pp. 1–8.
[6] "Virtex-7 FPGA Family," http://www.xilinx.com/products/virtex7.
[7] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" in *Proc. IEEE FCCM*, 2012, pp. 232–239.
[8] B. Ding and A. C. König, "Fast Set Intersection in Memory," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 255–266, 2011.
[9] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the Performance of List Intersection," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 838–849, Aug. 2009.
[10] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *Proc. SIGCOMM*, 2001, pp. 199–210.
[11] J. Li, H. Liu, and K. Sollins, "Scalable Packet Classification using Bit Vector Aggregating and Folding," MIT LCS Technical Memo: MIT-LCS-TM-637, 2003.
[12] Y. R. Qu, S. Zhou, and V. K. Prasanna, "A Decomposition-Based Approach for Scalable Many-Field Packet Classification on Multi-core Processors," *Intl. Journal of Paral. Prog.*, pp. 1–23, September 2014.
[13] S. T. Zargar, J. Joshi, and D. Tipper, "A Survey of Defense Mechanisms against Distributed Denial of Service (DDoS) Flooding Attacks," *IEEE Comm. Surv. & Tutor.*, vol. 15, no. 4, pp. 2046–2069, 2013.
[14] "Vivado Design Suite," http://www.xilinx.com/products/design-tools/vivado.html.
[15] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, "Optimizing Many-field Packet Classification on FPGA, Multi-core General Purpose Processor, and GPU," in *Proc. ACM/IEEE ANCS*, 2015, pp. 87–98.
[16] V. Adhinarayanan, T. Koehn, K. Kepa, W. chun Feng, and P. Athanas, "On the Performance and Energy Efficiency of FPGAs and GPUs for Polyphase Channelization," in *Proc. IEEE ReConFig*, 2014, pp. 1–7.
[17] S. Guo, C. Dang, and Y. Yang, "Joint Optimal Data Rate and Power Allocation in Lossy Mobile Ad Hoc Networks with Delay-Constrained Traffics," *IEEE Trans. on Computers*, vol. 64, no. 3, pp. 747–762, 2015.
[18] "Evaluation of Packet Classification Algorithms," http://www.arl.wustl.edu/~hs1/PClassEval.html.
[19] "AMD Opteron 6200 Series Processor," http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=791&f1=AMD+Opteron%E2%84%A2+6200+Series+Processor&f2=&f3=Yes&f4=&f5=&f6=G34&f7=B2&f8=32nm&f9=&f10=6400&f11=&.
[20] "Intel Xeon Processor E5-2470," http://ark.intel.com/products/64623/Intel-Xeon-Processor-E5-2470-20M-Cache-2_30-GHz-8_00-GTs-Intel-QPI?wapkw=e5+2470.
[21] "OpenMP," http://openmp.org/wp/.
[22] "Zynq-7000 All Programmable SoC," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

Yun R. Qu received the BS degree (2009) in electrical engineering from Shanghai Jiao Tong University, and the MS degree (2011) in electrical engineering at the University of Southern California. In 2015, Yun graduated from University of Southern California as a PhD in computer engineering; his research focuses were multi/many-field packet classification, and online traffic classification. He is currently working in the Xilinx Labs at San Jose, California. His research interests include algorithm design, algorithm mapping onto custom hardware, high-performance and power-efficient architectures, and productivity language for heterogeneous system prototyping. He is a member of IEEE and ACM.

Viktor K. Prasanna is Charles Lee Powell Chair in Engineering in the Ming Hsieh Department of Electrical Engineering and Professor of Computer Science at the University of Southern California. He received his BS in Electronics Engineering from the Bangalore University, MS from the School of Automation, Indian Institute of Science and PhD in Computer Science from the Pennsylvania State University. His research interests include High Performance Computing, Parallel and Distributed Systems, Reconfigurable Computing, and Smart Energy Systems. He serves as the Director of the Center for Energy Informatics at USC.

He served as the Editor-in-Chief of the IEEE Transactions on Computers during 2003-06. Currently, he is the Editor-in-Chief of the Journal of Parallel and Distributed Computing. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the Steering Co- Chair of the IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS) and is the Steering Chair of the IEEE/ACM International Conference on High Performance Computing (HiPC). Prasanna is a Fellow of the IEEE, the ACM and the American Association for Advancement of Science (AAAS). He is a recipient of the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University. He received the 2015 IEEE Computer Society W. Wallace McDowell award.