FPGA Acceleration for the Frequent Item Problem

Jens Teubner Rene Mueller Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

***Abstract*— Field-programmable gate arrays (FPGAs) can provide performance advantages with a lower resource consumption (*e.g.*, energy) than conventional CPUs. In this paper, we show how to employ FPGAs to provide an efficient and high-performance solution for the *frequent item* problem.**

We discuss three design alternatives, each one of them exploiting different FPGA features, and we provide an exhaustive evaluation of their performance characteristics. The first design is a one-to-one mapping of the *Space-Saving* algorithm (shown to be the best approach in software [1]), built on special features of FPGAs: *content-addressable memory* and *dual-ported BRAM*. The two other implementations exploit the flexibility of digital circuits to implement *parallel lookups* and *pipelining strategies*, resulting in significant improvements in performance.

On low-cost FPGA hardware, the fastest of our designs can process 80 million items per second—three times as much as the best known result. Moreover, and unlike in software approaches where performance is directly related to the skew factor of the Zipf distribution, the high throughput is independent of the skew of the distribution of the input. In the paper we discuss as well several design trade-offs that are relevant when implementing database functionality on FPGAs. In particular, we look at *resource consumption* and the levels of *data* and *task parallelism* of three different designs.

I. Introduction

The limitations and problems associated with modern CPU architectures are well known: high *power consumption*, *heat dissipation*, *network bottlenecks*, and the *memory wall*. These problems are compounded when the CPU is embedded in a complete computer. For instance, if applications are not carefully designed, CPUs can spend much of their time waiting for memory or disk. Similarly, a modern server CPU consumes over 100 Watts of *electrical power*, not counting necessary peripherals such as memory, disks, or cooling equipment. As another example, getting data in and out of the system often results in high *latency*, to the point that any algorithmic advantages may become irrelevant.

In the search for possible solutions, *field-programmable gate arrays (FPGAs)* have been proposed as a way to extend existing computer architectures with processing elements that help alleviate or eliminate some of these problems. FPGAs are particularly interesting today because they can be either added as additional processing cores in heterogeneous multi-core architectures [2], [3] and/or embedded in critical data paths (network-CPU, disk-CPU) to reduce the load and amount of data that hits the CPU [4].

What makes FPGAs interesting for designing data processing systems is that they are not bound to the classical von Neumann architecture. Thus, they can be used to avoid the memory wall, to implement highly parallel data processing, and to provide support that would be very expensive otherwise (*e.g.*, content addressable memory). They can also guarantee extremely low latencies and high throughput rates (for instance, allowing to process data from the network at *wire-speed*, without having to bring it to memory and the CPU first). In addition, and not least important these days, FPGAs feature a far lower power consumption than CPUs, making them ideal complements to general-purpose CPUs in many-core architectures.

FPGAs for the Frequent Item Problem. In this paper we tackle a basic data mining operation, calculating the *frequent items* in a data collection, and show how it can be implemented using FPGAs. We achieve throughput rates of up to 80 million items per second, a rate two to three times higher than the best published results for software-based implementations. The solution we propose can be used advantageously in business intelligence queries, high-volume data mining, and even real-time data processing (*e.g.*, to analyze traffic directly as it comes from the network).

Contributions. Our main contribution is a highly efficient frequent item operator based on FPGAs. The operator offers a performance that far exceeds the best published results. Moreover, the throughput of the operator is independent of the distribution of the input data, whereas software-based solutions only work well if the distribution of the input data is highly skewed (for Zipf-distributed data, a higher z parameter typically implies better performance). This makes our results even more relevant in practical settings, where the actual distribution of the input data might not be known in advance.

Our paper discusses three alternatives to solve the frequent item problem in hardware: Software-Like, Parallel-Lookups, and Pipeline. They illustrate some of the design considerations that many hardware implementations for data mining tasks will face. Through the three designs, we give guidance on how to find the right balance between resource availability, circuit complexity, and performance when designing FPGA-based solutions.

As part of illustrating the design trade-offs, we complement each of the three alternatives with an in-depth experimental evaluation, where we discuss resource requirements, scalability, and performance. As a main reference for the performance of existing software solutions, we use the in-depth study of the frequent item problem by Cormode and Hadjieleftheriou [5].

Outline. The paper is organized as follows. The upcoming Section II formalizes the problem and briefly recapitulates the

known solutions in software, before Section III gives a general background in FPGA technology. Sections IV to VI describe our three FPGA circuits, gradually moving from a more classical, software-inspired approach to the highly parallelizable pipeline-based solution. At the end of each section, we assess resource and performance trade-offs. In Section VII, we relate our work to others', before we summarize in Section VIII.

II. Frequent Items in Software

The search for frequently occurring items is a classical data mining problem. Example applications include identifying the network hosts that generate most traffic; detecting large trade volumes originating from the same company; or finding the most visited URLs. Frequent items are also helpful in the search for frequent item *sets*, *e.g.*, as an input to the a-priori algorithm [6].

A. The Frequent Item Problem

The frequent item problem can be defined as follows. Assume a stream S of items $x_1, \ldots, x_N$ drawn from an alphabet A. The *ϕ-frequent items* are those items in S that occur more than ϕN times. ϕN is called the *support* that result items must exceed to be considered frequent items. The number of occurrences of an item x in S is termed the *frequency* f_x of x.

It is easy to see that, even for large ϕ, the exact solution to this problem requires at least $\mathcal{O}(\min\{|N|, |A|\})$ space. An algorithm would have to remember all occurrences of an item $x \in A$ in the stream to determine the exact value of f_x, which, in turn, is a prerequisite for an exact solution.

Since exact solutions are expensive, research has focused on *approximate* algorithms that provide sufficient accuracy at low space and CPU overhead. These algorithms solve a (weaker) version of the problem: *ε-approximate frequent items*. The result set for the approximate problem must include all items x with $f_x > \phi N$, but may also include some items for which $(\phi - \varepsilon)N < f_x \leq \phi N$.

The work of Cormode and Hadjieleftheriou has provided an in-depth comparison of such algorithms [5]. The comparison indicates that the *Space-Saving* algorithm by Metwally *et al.* [1] is the best one among existing software solutions. In the rest of the paper we use *Space-Saving* as our performance baseline and as a starting point for the FPGA based designs. We refer the reader to [5] for details and characteristics of the other frequent item algorithms, which we will not cover here any further.

B. The Space-Saving Algorithm

Space-Saving tries to monitor frequencies only for those items that are frequent in the input stream. To this end, the algorithm keeps a number k of $\langle \mathsf{item}, \mathsf{count} \rangle$ pairs $b_1, \ldots, b_k$, which we refer to as *bins* in the following.

For every arriving item x, the algorithm checks whether x is already monitored in some bin b_x. If yes, the associated frequency estimate, $b_x.\mathsf{count}$, is incremented by one. Otherwise, the monitored bin with the lowest count value, b_{min}, is evicted and replaced by the pair $\langle x, b_{\mathsf{min}}.\mathsf{count} + 1 \rangle$ (see Figure 1). Observe how, in the latter case, item x receives the benefit of doubt: it could have occurred as often as $b_{\mathsf{min}}.\mathsf{count}$ times before. *Space-Saving* never under-estimates frequencies and, hence, records $b_{\mathsf{min}}.\mathsf{count} + 1$ as the frequency estimate for x.

```
1 foreach stream item x ∈ S do
2     find bin b_x with b_x.item = x ;
3     if such a bin was found then
4         b_x.count ← b_x.count + 1 ;
5     else
6         b_min ← bin with minimum count value ;
7         b_min.count ← b_min.count + 1 ;
8         b_min.item ← x ;
```

Fig. 1: Algorithm *Space-Saving*. A fixed number of bins monitors the most frequent items in the stream S [1].

The number of bins k reserved for monitoring is a configuration parameter that can be used to trade accuracy for space. As detailed in [1], $\lceil 1/\varepsilon \rceil$ counters are required to find frequent items with an accuracy of ε. As an example, 100 bins are needed to obtain a result with 1 % accuracy. In practice, frequent item algorithms are used to identify clear "heavy hitters," for which task the accuracy (and hence the number of needed bins) can often be kept even lower.

As shown by Cormode and Hadjieleftheriou, *Space-Saving* exhibits a very good accuracy-to-space ratio. With explicit knowledge about the expected data distribution the space requirement can be reduced even further [1].

C. Implementation Considerations

Though succinct and elegant, the challenge in realizing *Space-Saving* is that the same data—the currently monitored bins—have to be accessed under *two* independent criteria:

- *(i)* Line 2 in Algorithm *Space-Saving* needs to access the set of bins based on their item values (and the current input item x).
- *(ii)* If no match was found, bins have to be accessed via their count values to determine b_{min} (line 6).

To be able to answer *(ii)* efficiently, existing implementations keep their bins physically organized according to their count values. Metwally *et al.* [1] propose the use of a linked list for this purpose. Their data structure, dubbed *Stream-Summary*, implements a sorted list that can be re-organized with only $\mathcal{O}(1)$ effort after each bin update.

On the down side, as many as 10 pointer updates are necessary in *Stream-Summary* for each re-organization. In an otherwise very compact algorithm like *Space-Saving*, this could have noticeable impact on performance. As an alternative, Cormode and Hadjieleftheriou discussed an implementation that uses a min-heap (worst case complexity $\mathcal{O}(\log k)$) to have the minimum count value accessible at all times. In their

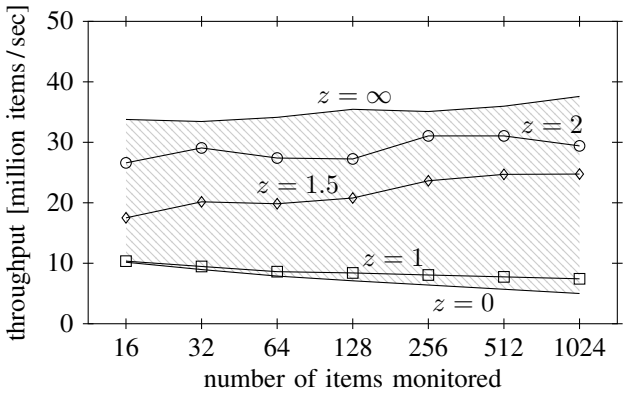


Fig. 2: Performance of Space-Saving [5] for different Zipf distributions $z \in \{0, 1, 1.5, 2, \infty\}$.

experimental assessment, the heap-based implementation came out only slightly behind *Stream-Summary*.

Either implementation has to invest re-organization effort whenever a counter increment leads to a violation of the sort or heap property. In effect, both implementations are sensitive to the distribution of the input data. [1] and [5] both report that high skewness in the input data can improve performance by a factor of around two.

Access operation *(i)* (line 2) suggests the use of a hash table for item lookups. Its complexity can typically be approximated as $\mathcal{O}(1)$. Since bins are primarily organized by count values, the hash table provides a secondary access mechanism that points into the main data structure.

D. Evaluation

We used the publicly available implementation of the *Space-Saving* algorithm by Cormode and Hadjieleftheriou to obtain a baseline for our work. Here we focus on the implementation that uses a min-heap as the primary bin organization (referred to as "SSH" in [5]). The implementation is going to be the basis for our first implementation on top of an FPGA, and it showed very good performance in the study of Cormode and Hadjieleftheriou.

We repeated the measurements of [5] on comparable hardware (a Core2 Duo T9550 2.66 GHz system with 6 MB L2 cache and 4 GB main memory) and obtained similar results (see Figure 2).

The most remarkable characteristic seen in Figure 2 is the dependence of the throughput on the input *data distribution*. While we see a throughput of around 5–10 million items per second for uniformly distributed data ($z = 0$), performance increases by a factor of more than three with increased skewness (*i.e.*, for $z \gtrsim 1.5$). The actual throughput in practice is going to be in the band indicated using diagonal shading ▨.

The cause of this behavior was not analyzed in detail by Cormode and Hadjieleftheriou. Table I fills in these details. During processing, updates to count values may necessitate re-organizations of the min-heap used to monitor items. The likelihood of such re-organizations depends on the skewness in the input data. For instance, while actual swapping of items in the min-heap is rare for $z = 2$, around four swaps are necessary on average to process each input item when $z = 1$.

Zipf parameter	heap swaps per item 64 bins	256 bins	1,024 bins
$z = \infty$	0	0	0
$z = 2$	0.04	0.02	0.02
$z = 1.5$	0.44	0.34	0.24
$z = 1$	3.12	3.99	4.49
$z = 0$	4.10	6.03	8.00

TABLE I: Average number of heap modifications per input item, as a function of the Zipf distribution parameter z (numbers obtained using an algorithm simulation in software).

Table I also explains the slight performance decrease for data with low skew as we increase the number of monitored items (4 swaps/item for $z = 0$ and 64 bins; 8 swaps/item for 1,024 bins).

III. FPGA Background

FPGAs, informally sometimes referred to as "programmable logic", are general-purpose hardware chips. In contrast to ASICs (application-specific integrated circuits), FPGAs have no pre-determined functionality. Rather, they can be configured to implement arbitrary logic by combining gates, flip-flops, and memory banks.

A. FPGA Resources

FPGAs provide a variety of resources. *Configurable logic* is provided through *lookup tables (LUTs)*, each of which can implement an arbitrary Boolean function with n inputs and one output ($n = 6$ for Virtex-5 FPGAs [7]). Lookup tables are backed up by carry logic that can be used to implement particular functionality directly in silicon and very efficiently. *Flip-flop registers*, one-bit memory entities, are woven into the logic fabric and thus provide fully *distributed storage*. Larger quantities of memory are available in the form of *Block RAM* (or *BRAM*). Virtex-5 chips, for instance, include a number of BRAM blocks, each of which provides 36 kbit of fast storage.

To satisfy even larger memory requirements, off-chip memory can be added using, *e.g.*, standard DDR-RAM. Computations and experimental assessments in this work are based on a Xilinx VC5VFX130T FPGA model. It has a reasonably large chip space and two additional, on-chip PowerPC cores. See Table II for the resources available on this FPGA.

Other Virtex-5 chips provide significantly more resources: up to 2.5 times more logic resources (lookup tables and flip-flops) and almost twice as many BRAM blocks. The upcoming Virtex-6 series offers up to 474,240 lookup tables and 948,480 flip-flops, and up to 1,064 BRAM blocks; significantly more than the hardware used for the work we report here.

FPGAs typically operate at clock frequencies that are significantly lower than those of general-purpose CPUs (≈ 100 MHz). They are still competitive because tailor-made

lookup tables (6-to-1 lookup tables)	81,920
flip-flops (1-bit registers)	81,920
block RAM (total kbit)	10,728
block RAM (number of 36 kbit blocks)	298
18-bit multipliers	320
PowerPC Cores	2

TABLE II: Selected characteristics of the Xilinx FPGA used in this paper, model VC5VFX130T.

circuits can perform more work in less cycles than software-based systems. A side effect of the low clock frequency, on the other hand, is the *low energy consumption* of FPGAs (≈ 1 Watt vs. ≈ 100 Watts for a modern CPU).

B. System Integration

FPGA are usually programmed in a *hardware description language*, such as *VHDL (VHSIC hardware description language)* or *Verilog*. A *synthesizer* compiles the description of a hardware circuit into a *bitstream* that is then loaded into the FPGA chip. Existing circuits can also be re-configured, either off-line, at runtime upon workload changes, or even dynamically for individual user queries.

The configured FPGA can be integrated into a system in various configurations. Common configurations are to use the FPGA as a *co-processor* to a general-purpose CPU or to insert the configurable logic into the *data path* of the system [8]. The former case relies on fast interconnects such as PCI Express or HyperTransport; in the latter case the FPGA can be directly wired to the controllers of disk drives or network cards.

C. Block RAM: Dual-Ported and Configurable.

In this paper we exploit the high configurability of the available BRAM. In Virtex-5 FPGAs, each BRAM block provides 36 kbit of on-chip memory. Unlike in commodity systems, the *word size* of each block can be configured: the 36 kbit of BRAM can be partitioned into, for instance, 1,024 words of 36 bit, 4,096 words of 9 bit, or 32,768 single-bit words.[1] Multiple BRAM blocks can be wired together to obtain larger memories and/or larger word sizes.

All BRAM blocks are *dual ported*. Two independent ports provide access to the same physical data and as truly concurrent operations.[2] Moreover, the word size of both ports can be configured independently; data might be written, *e.g.*, as two 8-bit words on one port, later accessed as a single 16-bit word using the other port. We will shortly see how we can use this feature to implement key-value stores and heap structures in an efficient manner.

IV. A Software-Like Solution in Hardware

Algorithm *Space-Saving* is very compact and known to be efficient in software-based systems. We use it as the starting point for implementing frequent items in an FPGA and explore how to best exploit the features available in an FPGA board.

[1] In some configurations not the full 36 kbit can be used.

[2] The semantics for two conflicting write operations is undefined.

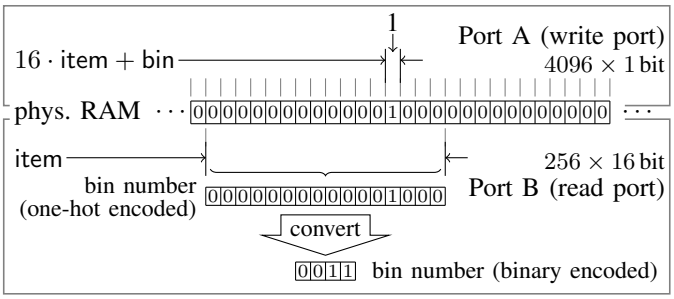


Fig. 3: CAM implementation using dual-ported BRAMs. A single bit is written via BRAM Port A to store an $\langle \mathsf{item}, \mathsf{bin} \rangle$ pair. Reading via Port B reveals the bin number for a given item in one-hot encoded form (16 bits).

We first illustrate two such features, before we package them into a working solution for the frequent item problem, which we call Software-Like later on.

When designing an FPGA circuit, we prefer a min-heap-based bin storage over the linked list-based *Stream-Summary* of [1], even though the latter showed a small advantage in Cormode and Hadjieleftheriou's comparative analysis. The necessary pointer chasing in *Stream-Summary* is relatively expensive in FPGA circuits and should thus be avoided.

A. Content-Addressable Memory: A Hash Table on Steroids

The first task in Algorithm *Space-Saving*, the lookup of bins based on item values (line 2 in Figure 1), is a good candidate for *content-addressable memory (CAM)*, a hardware-accelerated key-value store with strong runtime guarantees. CAMs are a standard device in network processing (*e.g.*, for packet classification) and have recently been proposed also as a tool for frequent item computation [9].

For Software-Like, we build a content-addressable memory based on dual-ported BRAM. It provides a good balance between write and read performance and supports the problem sizes that we are interested in for the frequent item search (see [10] for CAM implementation alternatives).

The key idea of our BRAM-based CAM implementation is the use of differently configured BRAM ports for read and write accesses. Figure 3 illustrates this for a CAM that supports 8-bit keys and 4-bit values (as it could be used for a very small-scale *Space-Saving* implementation with 16 bins and an input alphabet of size $|A| = 256$).

Port A in this illustration is configured to a word size of 1 and used for writing entries into the CAM. To store a pair $\langle \mathsf{item}, \mathsf{bin} \rangle$ in the CAM, we set the bit at address $16 \cdot \mathsf{item} + \mathsf{bin}$ to 1 (top half of Figure 3). Thus, we need (at least) 4,096 single-bit words accessible via Port A.

To look up the bin number of a given item, we access the stored information via Port B which is configured to a word size of 16 bit (in support for 16 bins). As illustrated in Figure 3, the value item is used as an address to retrieve a 16-bit word. This word contains the bin information in *one-hot encoded* form, *i.e.*, a single bit in this word is set and its position

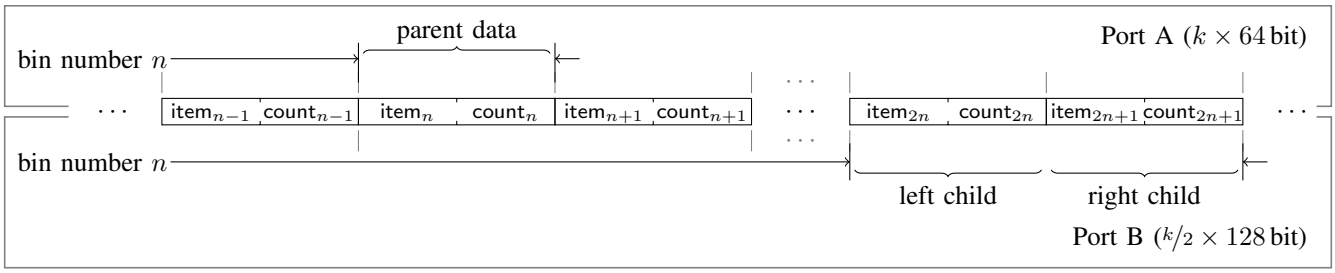


Fig. 4: Implementation of a min-heap using dual-ported BRAM. Bin number n applied to the address lines of both ports will yield record b_n at Port A and its two children b_{2n} and b_{2n+1} at Port B.

encodes the bin number. A standard encoding circuit converts this representation into the common *binary encoding*.

Support for Larger Alphabets. The CAM implementation in Figure 3 has a clear scaling problem. To support, say, a 32-bit alphabet and 100 bins, we would need as many as $2^{32} \cdot 100$ bits (or 50 GB) of BRAM capacity. This explosion can be avoided by chopping item words into smaller sub-words, distributing them over separate CAMs, and recombining CAM outputs upon lookup (refer to [10] for details).

B. Min-Heaps in Dual-Ported BRAM

The bin that holds the minimum count value can be found quickly (line 6 in Algorithm *Space-Saving*, Figure 1) if all bins are organized as a *min-heap*. This had also been suggested by Cormode and Hadjieleftheriou. A min-heap is a binary tree where the value stored in a node is never larger than the value stored in any of its children. Thus, the bin with the smallest count value is readily available as the root of the min-heap.

The efficient access of that bin comes at the cost of a small maintenance overhead, which has to be paid after every update of the data structure. After each count increment, the heap property must be validated and the tree re-organized if necessary. To this end, we must compare the modified node with both its children and, if necessary, swap parent and child node and recurse. (Min-heap inserts, consequently, have a $\log k$ worst case complexity.)

In an FPGA implementation, we can again benefit from the dual-ported access mechanism to BRAM blocks. With the proper data layout and BRAM configuration, a node and both of its children can be read or written at the same time and within a single FPGA clock cycle.

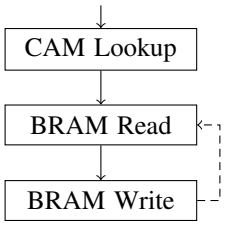
The idea is illustrated in Figure 4. Port A provides the expected type of access to all k nodes of the heap. We store item and count information using 32-bit values each, suggesting a word size of 64 bit on BRAM Port A.

As usual, our heap is represented as an array in which the children of a node at array position n can be found at positions $2n$ and $2n + 1$ (left and right child, respectively). With two siblings always at adjacent locations (and starting at an even location), we can use the configurability of FPGA BRAM to access both of the simultaneously. To do so, we configure Port B to a word size of 128 bit, as illustrated in Figure 4. Since the word size is twice the logical record size, an access to Port B with address n will automatically yield both children of node b_n.

Heap maintenance is a good example where high configurability can compensate for the comparably low clock frequencies of FPGAs. While, in software, separate instructions are required to access each of the heap nodes and to compare them one by one, an FPGA can perform all lookups and comparisons in a single cycle. If necessary, the modified nodes are again written back in just one cycle. In fact, our implementation performs all comparisons *concurrent* to counter increments, such that no cycles are wasted if the heap layout need not be changed.

C. The Pieces Plugged Together

Solving the frequent item problem along the lines of Algorithm *Space-Saving* naturally leads to processing each input item in three stages. Our implementation SOFTWARE-LIKE executes these stages as shown on the right. We implemented them in VHDL using content-addressable memory and a BRAM-based min-heap structure as discussed before. Processing stages are coordinated by a finite state machine implemented in FPGA logic.



First, our circuit consults the content-addressable memory to locate the corresponding bin (implicitly creating a new entry if the item is not currently monitored). Information about the item is then read from BRAM, updated, and written back. Sometimes, it may be necessary to re-organize the min-heap, in which case SOFTWARE-LIKE iterates as indicated with the dashed line (re-organization also triggers changes to the content-addressable memory not shown in the figure).

A process of this type is a good candidate to exploit some of the *parallelism* offered by FPGAs. Our implementation SOFTWARE-LIKE will, for instance, overlap the CAM lookup of an input item with the bin updates triggered by its predecessor. Likewise, we parallelize heap re-organizations and their associated updates to the content-addressable memory. Both optimizations further increase the amount of work that can be achieved per FPGA clock cycle.

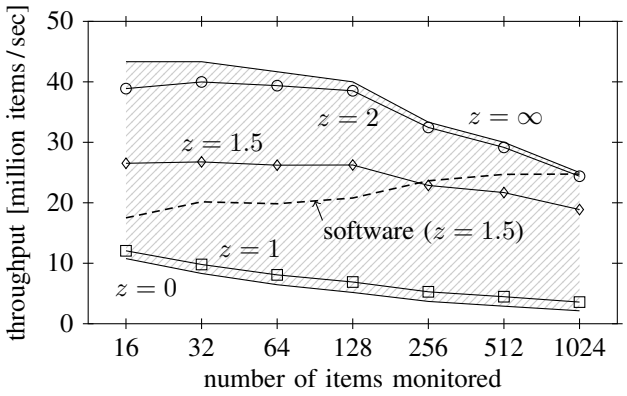


Fig. 5: Performance characteristics of SOFTWARE-LIKE implementation, based on content-addressable memory and dual-ported BRAM, for input data sets that follow Zipf distributions $z \in \{0, 1, 1.5, 2, \infty\}$. Dashed line: performance of the software implementation.

D. Evaluation

We implemented the circuit for SOFTWARE-LIKE using the aforementioned XC5VFX130T Xilinx FPGA. We configured the circuit to monitor between 16 and 1,024 items and measured its throughput with data that follows a Zipf distribution. The throughput we achieved for each configuration and for different values for the Zipf parameter z is reported in Figure 5.

Performance Characteristics. Two characteristics are most apparent in the graph:

(i) The achieved throughput is very sensitive to the distribution of the input data. Skewed data (large z) can be processed about three times faster than uniformly distributed input.

(ii) Throughput decreases when the number of monitored items is scaled up.

We already observed characteristic *(i)* when we evaluated an implementation of *Space-Saving* in software. The data dependence is mainly caused by necessary re-organizations of the data structure used to monitor items, a min-heap in the implementation we consider here. Non-uniform data distributions reduce the likeliness that such re-organizations are necessary. In Table I (Section II) we looked at the necessary heap swaps in detail. The same conclusions we drew for the software-based implementation also hold for the SOFTWARE-LIKE counterpart.

Signal Propagation Delays. Characteristic *(ii)*, the performance degradation of SOFTWARE-LIKE with an increasing number of monitored items is an artifact specific to FPGAs. To build larger content-addressable memories, an increasing amount of BRAM blocks have to be wired together into a single functional unit. The growing complexity of this unit leads to longer *signal paths* and, hence, to longer *signal propagation delays*.

configuration	lookup tables		flip-flops		BRAMs		clock freq.
64 bins	3,111	3 %	2,161	2 %	29	9 %	125 MHz
128 bins	3,363	3 %	2,485	3 %	37	12 %	120 MHz
256 bins	3,863	4 %	2,439	2 %	53	17 %	100 MHz
512 bins	4,919	6 %	2,639	3 %	85	28 %	90 MHz
1,024 bins	6,775	8 %	3,157	3 %	149	50 %	75 MHz

TABLE III: FPGA chip resource consumption for the SOFTWARE-LIKE implementation (based on content-addressable memory and dual-ported BRAM). BRAM is the critical resource for this setup.

The longest signal path determines the maximum frequency at which the overall circuit can operate. While we were able to clock our smallest circuit instance (16 bins) at a rate of up to 130 MHz, a clock rate of 75 MHz was the maximum for the instance with 1,024 bins. This directly results in the observed performance degradation. This is a problem specific to FPGAs. Our experiments using software (Figure 2) do not show the same performance degradation for large algorithm configurations.

Chip Resource Requirements. The min-heap to hold monitored items and the content-addressable memory are the primary chip resource requirements of this solution to the frequent item problem, and both boil down to the consumption of BRAM blocks. Only a little amount of logic is required, on the other hand, to implement the state machine that drives processing. As shown in Table III, BRAM blocks are the main chip resource that the implementation consumes. The amount of available BRAM blocks in the chip thus limits the number of bins that we can instantiate to 1,024.

Larger chips (such as the Virtex-5 XC5VLX330T or the upcoming Virtex-6 VC6VLX760) include more BRAM blocks than the hardware we have available and could host configurations with 2,048 or even 4,096 bins. However, circuit complexity and the resulting signal propagation delays are only going to become worse when we increase configuration sizes further.

Summary. The use of content-addressable memory and a BRAM-based bin storage may be seen as a straightforward translation of the existing software algorithm into hardware. It is not surprising that SOFTWARE-LIKE also inherited a critical deficiency that is already known to exist in software-based implementations. Heap maintenance makes the performance of the implementation *data dependent*. This may be prohibitive in scenarios that depend on predictable behavior even when the nature of the input data is not predictable.

V. PARALLELIZE, DON'T SORT

We can address these deficiencies by leveraging some of the FPGA features that do not apply to a straightforward implementation of an algorithm that was designed for execution in software. In particular, we can use the *parallelism* that is inherent to FPGAs to replace the data-dependent heap structure by an alternative that is less sensitive to the input

data distribution. We are going to refer to the resulting circuit as PARALLEL-LOOKUPS.

A. Finding the Bin with Minimal count

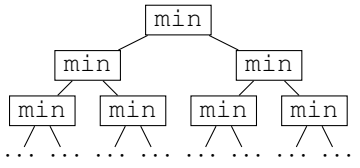
More specifically, we use parallelism to find the identifier min of the bin b_{min} that holds the smallest count value. The search for this bin was the motivation to use a heap structure in the implementation SOFTWARE-LIKE.

The corresponding circuit can easily be constructed using a VHDL description. Assuming only two bins b_a and b_b (identifiers a and b; count values count_a and count_b, respectively) to choose from, a component to implement the functionality could be coded as (<= denotes signal assignment in VHDL):

```
architecture min (a       : in integer,
                  count_a : in integer,
                  b       : in integer,
                  count_b : in integer,
                  min     : out integer,
                  count_min : out integer) is
begin
  min <= a when count_a < count_b else b;
  count_min <=
    count_a when count_a < count_b else count_b;
end;
```

A VHDL compiler will translate this description into a comparison of count values, followed by two multiplexers that drive the output ports of the component, min and $\text{count}_{\text{min}}$ (we detailed the inner workings of a similar operation in [11]).

It is easy to see that component `min` can be composed into a tree that performs bin search for an arbitrary number of input bins. To process an input of k bins, $k-1$ `min` elements are needed to construct a search tree of height $\lceil \log_2 k \rceil$ (*e.g.*, using VHDL structural modeling):



tree for $k = 8$

With such a search tree, it is no longer necessary to organize bins in a particular way (such as a min-heap) or to invest time in sorting. Using the tree, we do not require any particular bin order and can exploit parallelism for efficient bin access.

B. Parallel Item Search

The item lookup corresponding to line 2 of Algorithm *Space-Saving* can be implemented in very much the same way. The corresponding circuit is sketched in Figure 6. An array of simple logic components [=] compares the input item x to all currently monitored items item_i in parallel. In case of a match, the bit-wise 'and' operator [&] emits the address b_i of the matching bin or zero otherwise. Since an item can be found at most in one bin, collecting the output of all [&] using a bit-wise 'or' operation (indicated as the n-ary operator `or` in Figure 6) yields the address b_x of the bin that currently monitors x—or zero if the item was not found.

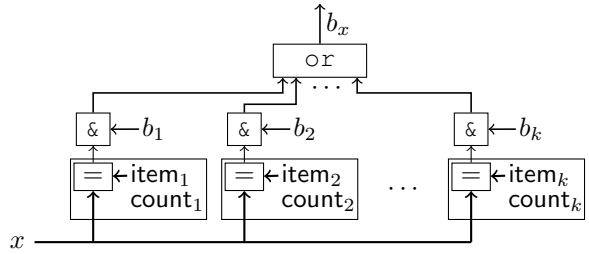


Fig. 6: All bins b_i are compared with the input item x in parallel to determine whether x is currently monitored.

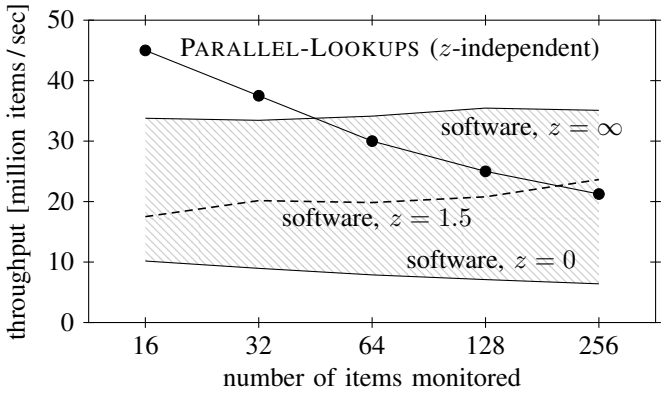


Fig. 7: Throughput of FPGA implementation PARALLEL-LOOKUPS (—●—). Performance of software implementation shown for reference (▨/---).

All sub-tasks are particularly efficient to perform on FPGAs. Bit-wise 'and' operations are supported by the fast carry logic gates. Comparison components as well as the `or` operator can use the full six LUT inputs for fast, resource-efficient processing. We can, for instance, implement the n-ary `or` operation by composing 6-to-1 lookup tables into a tree of height $\lceil \log_6 n \rceil$.

Parallel searches for items and count values in PARALLEL-LOOKUPS can leverage the *asynchronous* processing mechanisms offered by FPGAs. Component `min` as well as the building blocks for parallel item search ([=], [&], and bit-wise `or`) all use *combinatorial logic* only. The performance of an asynchronous circuit built from such operators only depends on the low-level *signal propagation delays* inside the chip. In earlier work [11], we demonstrated how this can considerably improve the performance of an FPGA circuit.

C. Evaluation

By using parallelism we avoid the heap maintenance required in SOFTWARE-LIKE. An immediate consequence can be seen in the performance chart shown in Figure 7. The throughput of the PARALLEL-LOOKUPS circuit has become *independent* of the input data distribution (contrast to the software solution, which we also plotted into Figure 7 for reference). Data skew no longer affects the performance of

configur.	lookup tables		flip-flops		BRAMs		clock freq.
32 bins	6,470	7 %	3,892	4 %	24	8 %	150 MHz
64 bins	11,890	14 %	5,973	7 %	24	8 %	120 MHz
128 bins	22,623	28 %	10,134	12 %	24	8 %	100 MHz
256 bins	45,267	55 %	18,455	22 %	24	8 %	85 MHz

TABLE IV: FPGA chip resource consumption and clock frequencies of implementation PARALLEL-LOOKUPS.

the hardware circuit, a benefit that is particularly valuable if the distribution of the input data is not known in advance.

Performance Characteristics. As mentioned before, the performance of PARALLEL-LOOKUPS is primarily determined by signal propagation delays inside the chip. When scaling up the number of bins, two effects result in increasing propagation delays:

(i) With increasing heights of `min` and `or` trees, more lookup tables have to be traversed. Each lookup table adds a fixed propagation delay.

(ii) The high overall fanin of the two trees complicates on-chip signal routing. The logic synthesizer thus has to resort to sub-optimal routing strategies with high routing delays.

Both factors affect the maximum clock frequency at which we can operate the hardware circuit (growing content-addressable memories caused the same effect in the previous circuit). While we could run the 16-bin configuration at 180 MHz, routing delays forced us to clock our largest instance (256 bins) at no more than 85 MHz.

The finite state machine that controls the operation of PARALLEL-LOOKUPS requires four cycles for each input tuple (independent of the data distribution). This yields a throughput of 45 million items per second for the 16-bin configuration, but only 21.25 million items per second for the configuration with 256 bins.

Chip Resource Requirements. The high degree of parallelism directly affects the amount of logic resources required. As shown in Table IV, the limited availability of lookup tables, the actual resource that performs computation, now prevents us from scaling up our implementation beyond 256 monitored items (we considered only powers of 2).

In addition, the nature of PARALLEL-LOOKUPS may prevent further scale-ups. Large search trees already cause significant routing problems and large signal propagation delays. These problems do not go away with larger or faster boards.

A difference to the SOFTWARE-LIKE design is that we now have to use flip-flop registers to store bin data. Flip-flops are tightly woven into the FPGA logic and can thus be accessed fully in parallel. The contents of BRAM blocks, by contrast, need to be requested explicitly by address, and at most one word can be fetched from each BRAM block per clock cycle and BRAM port.

Summary. The notable improvement over the earlier SOFTWARE-LIKE implementation is that the throughput of PARALLEL-LOOKUPS is independent of the input data distribution. For data with a small skew, this resulted in a clear improvement of the net throughput.

On the down side, the performance degradation for large circuit configurations has become worse. Between 16 and 256 bins, we see a throughput reduction by more than a factor of two (because we had to operate the large configuration at a lower clock speed).

```
1  foreach stream item x ∈ S do
2      i ← 1 ;
3      while i < k do
4          if b_i.item = x then
5              b_i.count ← b_i.count + 1 ;
6              continue foreach ;
7          else if b_i.count < b_{i+1}.count then
8              swap contents of b_i and b_{i+1} ;
9          else
10             i ← i + 1 ;
       /* replace last bin if x was not found */
11     b_k.count ← b_k.count + 1 ;
12     b_k.item ← x ;
```

Fig. 8: Algorithm *Array*. Keep all processing and communication local to ensure scalability.

VI. PIPELINING FOR SCALABILITY AND THROUGHPUT

The search trees in PARALLEL-LOOKUPS showed a drop in execution performance with growing sizes of the hardware circuit. The main cause is that the necessary on-chip wiring has to interconnect many storage bins that are far apart on the chip die. This leads to long signal paths and complex routing. A circuit that keeps all wirings and computations local has better chances to show good scalability.

A. Algorithm Array

We can obtain such locality when we organize all bins as an *array* in which each bin is only connected to its two immediate neighbors. The algorithm now has only a restricted view on the currently monitored information and must implement its functionality by communication along the array.

An algorithm that implements these restrictions is shown in Figure 8 as Algorithm *Array*. The algorithm passes each new item x linearly along the array, compares x with each bin, and updates the bin's count value if a match was found (lines 9–10 and 4–6).

As item x travels along the array, lines 7 and 8 in Algorithm *Array* test the local order among adjacent bins and, if necessary, swap bin contents to bring the bin with the smaller count value to the right (we assume small bin indexes to be to the "left" and large indexes to be to the "right"). A consequence is that the traveling item pushes the bin with the smallest count value toward the right end of the array.

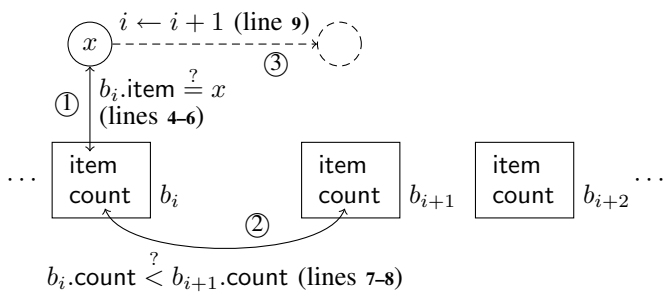


Fig. 9: The three processing steps of Algorithm *Array*.

Once item x reached the end of the array and no match was found, we can be certain that the last bin holds the smallest count value. According to Algorithm *Space-Saving*, this is the bin where we place the new item, as done in lines 11 and 12 in Figure 8.

We illustrate the three processing steps of Algorithm *Array* in Figure 9. In Step ① (algorithm lines 4–6), item x is compared to the local bin and the count value incremented if necessary. Otherwise, Step ② (lines 7–8) compares count values of the current bin and its right-next neighbor and swaps the two bin contents if need be. If neither action was performed, item x moves on to consider the next bin (Step ③, lines 9–10).

Properties of the Algorithm. Algorithm *Array* is semantically equivalent to Algorithm *Space-Saving*. The two searches by item and count value are implemented within one sequential read over the monitored data. As such, the processing time for a single input item is guaranteed to be $\mathcal{O}(k)$, independently of the input data distribution.

Moreover, the bin swapping mechanism in lines 7 and 8 provides the functionality of *bubble sort* inside Algorithm *Array*. Traveling items will gradually *sort* the monitored bins in descending order of their count values. This order is typically requested by users or higher-level algorithms (such as a-priori). Though strictly speaking this order is not guaranteed by *Array*, we found it to hold in all practical cases. If not, our hardware implementation PIPELINE required only few additional cycles to establish a fully sorted result.

B. Pipeline Parallelism

If implemented on a regular CPU, Algorithm *Array* is obviously inefficient. While the original *Space-Saving* algorithm can be implemented with $\mathcal{O}(1)$ (approximated; using a *Stream-Summary*) or $\mathcal{O}(\log k)$ (approximated; using a min-heap) complexity, the time needed to process an item in *Array* depends linearly (*i.e.*, $\mathcal{O}(k)$) on the configuration size k. On the other hand, all sub-tasks involved are simple and, as we will see in a moment, *Array* can be parallelized well, which makes it a good basis for an implementation in an FPGA.

The bin array can be viewed as a pipeline. Each item progressively traverses this pipeline and changes state only locally. Multiple items can follow one another in the pipeline and will not interfere with each other as long as they keep sufficient distance. An FPGA can process all such items in parallel.

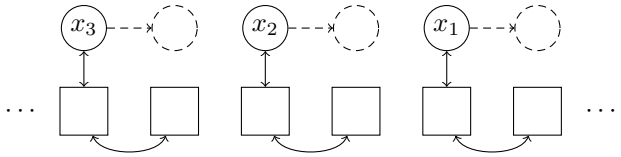


Fig. 10: Pipeline parallelism. Multiple input items x_i follow one another and are processed in parallel.

Any processing step in Algorithm *Array* operates on two adjacent bins at most. Items will thus never interfere with each other if they are separated by at least one bin; *i.e.*, up to $k/2$ items can traverse an array of length k simultaneously. This is illustrated in Figure 10. The six bins on the bottom of that figure represent a subset of the bin array. Items x_1 through x_3 follow each other with one bin separation. The three steps in Algorithm *Array* (illustrated using arrows as before) can only reach bins in a way that will not cause interference.

Analysis. Assuming sufficient resources for parallelism (as it is the case in FPGAs), pipelining makes the throughput of Algorithm *Array* independent of the length of the bin array (*i.e.*, of the number of items monitored). This is an even stronger guarantee than the $\mathcal{O}(1)$ complexity of *Space-Saving* which assumes constant-time hash lookups.

Pipeline parallelism could, in theory, also be implemented in software on top of general-purpose CPUs. But even if CPU resources were available in sufficient quantity (several tens or hundreds of CPUs), such an implementation would suffer from a significant overhead to synchronize threads and communicate data between them. In addition, threads would all compete for the same cache lines and thus experience high cache miss rates. None of these problems are an issue in an FPGA-based implementation.

C. Implementation Details

Our actual hardware implementation PIPELINE performs all steps ①–③ as one processing unit, which makes each item's progress in the pipeline fully predictable (hence, avoids expensive synchronization). To ensure the correctness of the implementation (and not miss a bin match), we compare x with the contents of b_i and b_{i+1}.

When a match is found for an item x, PIPELINE continues traversing the array (as opposed to aborting the **while** loop in line 6), but x is replaced by a special *void* item that will never match and will not be put into the last bin of the pipeline. This increases the regularity of our circuit, and we benefit from the sort operations that are also performed during the processing of *void* items.

The latter two implementation details also make the throughput of PIPELINE *data independent*. The execution of a processing unit takes the same amount of time, no matter how skewed the input data. Like the PARALLEL-LOOKUPS design in the previous section, the implementation PIPELINE features predictable data throughput.

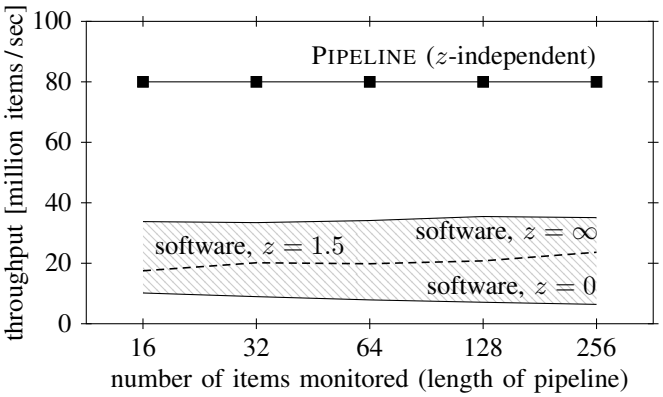


Fig. 11: Performance of hardware circuit PIPELINE (—■—). Due to pipeline parallelism, the throughput sustained is independent of the length of the pipeline. Performance of software implementation shown for reference (▨/- - -).

configuration	lookup tables		flip-flops		clock freq.
32 bins	8,720	10 %	8,312	7 %	80 MHz
64 bins	16,887	20 %	6,880	8 %	80 MHz
128 bins	32,023	39 %	12,033	14 %	80 MHz
256 bins	62,260	76 %	22,335	27 %	80 MHz

TABLE V: Resource consumption of PIPELINE.

We constructed our circuit to perform *two* processing units within each clock cycle. That is, after each clock cycle we move each item x_j forward by two bins (and perform up to two swaps). Once more this increases the regularity of our circuit, which now performs the same work on the same bins during every clock cycle. As a result, we can accept one item from the input with every clock tick.

D. Evaluation

We ran our implementation PIPELINE in the Virtex-5-based testbed for various numbers of bins. For all configurations, we were able to push the clock frequency of the circuit up to 80 MHz. This frequency is mostly dictated by complexity of each instance of the processing unit that we programmed and by the longest signal path inside the unit. All processing units operate entirely independent. Hence, the maximum clock frequency is independent of the number of units, *i.e.*, independent of the number of bins monitored.

Performance Characteristics. This can also be seen in Figure 11, where we illustrated the throughput of PIPELINE. As can be seen, the throughput is independent of the pipeline length. As mentioned before, our circuit can accept one input item per clock cycle, resulting in a net throughput of 80 million items per second.

Chip Resource Requirements. Resource requirements for the PIPELINE implementation (listed in Table V) scale similarly to what we saw previously for PARALLEL-LOOKUPS, though the absolute numbers are somewhat higher.

Again we use flip-flops to keep state (a requirement for the highly parallel access that we perform). In addition to storage for the bin contents, this time we also need additional flip-flops to hold all items that concurrently traverse the pipeline ("the circles in Figures 9 and 10"), which explains the additional flip-flop consumption.

The pipelined circuit also has to perform more computational work for each bin. Each processing unit (as mentioned before) consists of several comparisons (equality and inequality) and two counter increments, as well as a swap logic; and one processing unit has to be processed per bin and clock cycle. PIPELINE's consumption of lookup tables thus is appreciably higher than what we saw for PARALLEL-LOOKUPS, but also yields a two- to four-fold throughput improvement.

Scalability. A virtue of the pipeline-style circuit design is its scalability with circuit sizes. We already discussed how the locality of computations within PIPELINE and the independence of processing units leads to good scalability. The circuits we generated all achieved a throughput of 80 million items per second. FPGAs with more chip real estate would allow us to increase the length of the pipeline beyond the 256 bins that we considered so far. Currently there is no indication why 512-bin or 1,024-bin circuits on such hardware should not be able to sustain 80 million items per second, too.

Even larger pipeline instances could be obtained by daisy-chaining multiple FPGA chips, a setup that would naturally be supported by the pipeline structure of our circuit. Obviously, this would necessitate specialized multi-chip hardware, which is typically harder to obtain than single-chip solutions.

Summary. The PIPELINE implementation clearly wins the throughput race among all frequent item solutions we are aware of. Independent of the input data distribution, our circuit sustains a throughput of 80 million tuples per second.

What makes PIPELINE even more attractive, however, is its high potential to scale to significantly larger bin configurations on suitable, but already mainstream, hardware. On such hardware, we expect that PIPELINE is going to show a throughput of 80 million items per second also for pipelines with 512 or 1,024 bins (again only considering powers of two).

VII. RELATED WORK

In their survey article, Cormode and Hadjieleftheriou [5] give an excellent overview over existing (software-based) techniques to answer the frequent item problem, including groups of algorithms that we did not mention here (such as quantile or sketch-based algorithms). We refer to their paper for related work on general frequent item techniques.

We would like to mention the work of Bandi *et al.* [9] explicitly here, however. They too suggested the use of content-addressable memory to determine frequent items. To this end, they assume a specialized hardware component originally designed for network processing. Compared to the CAM we instantiated inside an FPGA, such a hardware solution provides higher storage capacities as well as support for

ternary CAM. Ternary CAMs allow the use of *wild cards* in the search key. Bandi *et al.* use this feature extensively and use content-addressable memory as their *only* access mechanism to an otherwise unordered bin storage (no heap or similar data structure on the side). In summary, they were able to achieve throughput rates slightly under 1 million input items per second. Like software-based implementations or our SOFTWARE-LIKE circuit, this implementation is sensitive to value distributions in the input data.

The structure of Algorithm *Array* has many similarities to *systolic arrays*, a concept that emerged as a design guide for hardware circuits in the late 70s. Kung and Leiserson [12] discovered systolic arrays as a very efficient, yet simple-to-manufacture type of circuits to perform matrix multiplications. Later, Kung and Lohman [13] and Hurson *et al.* [14] used the same principle to implement basic database functionality.

The idea of systolic arrays was also picked up by Baker and Prasanna [15], who implemented parts of the a-priori algorithm with a technique they termed *systolic injection*. The main use of their circuit is the calculation of support for candidate item sets. In an array of processing units, each unit is initialized with one candidate set. Then, data is streamed through the array (one transaction after the other), and each unit counts the number of transactions that contain the candidate set.

Finally, this work is embedded into our own research in the context of the *Avalanche* project at ETH Zurich. As FPGAs go mainstream, we assess how their potential can be leveraged to accelerate core database operations. *Glacier* [8], the query compiler in *Avalanche*, uses metrics such as resource consumption and performance to optimize FPGA circuits that implement database tasks, and could decide among either of the three presented FPGA circuits in this work.

VIII. SUMMARY

FPGAs are poised to become an ideal complement to traditional CPUs in many-core architectures. FPGAs have a very low power consumption and can be tailored to perform data processing tasks more efficiently than software-based solutions by taking advantage of the characteristics of FPGAs and the inherent parallelism.

In this paper we have presented three different FPGA designs that implement the search for frequent items: SOFTWARE-LIKE, PARALLEL-LOOKUPS, and PIPELINE. The different designs illustrate very well the possibilities and trade-offs inherent in FPGAs, as well as their advantages over software-based solutions. Figure 12 summarizes the throughput of each one of the techniques discussed in the paper, including the software implementation of *Space-Saving* [1], [5] as a baseline.

At the time of writing this paper we did not have hardware available that would allow us to consider larger configurations with more bins. Yet, the results and trends presented indicate that FPGAs are clearly competitive as the basis for implementing frequent item search. PIPELINE, the best design in terms of performance provides a throughput of 80 million items per second. Moreover, the throughput is independent of the distribution of the input data, a major difference over software-based solutions. Considering that much larger designs are possible with modern FPGAs, the implementation PIPELINE showed far better performance than any known software-based solution.

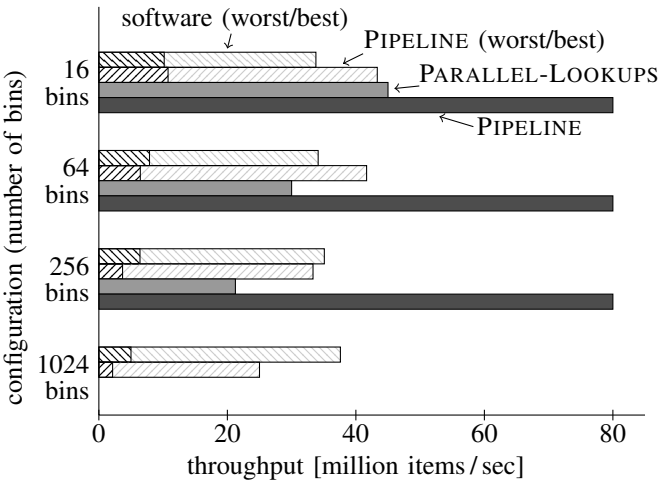


Fig. 12: Throughput comparison of all approaches discussed.

Work in Progress. As part of the *Avalanche* project at ETH Zurich, we envision a hybrid setup where operators running on CPUs and on FPGAs complement each other. In the particular case of frequent item calculation, such operators could actually work together on the same task. We are currently investigating an architecture where we use the pipeline-based circuit discussed in this paper to process the bulk of the input data. At the end of the pipeline, items could then be handed off to a general-purpose CPU. Faced with only a small remaining load and dealing with the lower frequency items, this CPU could then provide scalability to very large bin configurations.

We are also exploring configurations where the frequent item operator on the FPGA is applied directly to real time data streams from the network, exploiting both the high data rates demonstrated in this paper and the low latency I/O systems available in the FPGA [8].

REFERENCES

[1] A. Metwally, D. Agrawal, and A. E. Abbadi, "An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 3, pp. 1095–1133, Sept. 2006.

[2] D. Greaves and S. Singh, "Kiwi: Synthesis of FPGA Circuits from Parallel Programs," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[3] Kickfire, http://www.kickfire.com/.

[4] Netezza Corp., http://www.netezza.com/.

[5] G. Cormode and M. Hadjieleftheriou, "Finding Frequent Items in Data Streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[6] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proc. of the 20th Int'l Conference on Very Large Data Bases (VLDB)*, Sept. 1994, pp. 487–499.

[7] *Virtex-5 FPGA User Guide*, Xilinx Inc., May 2009.

[8] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires—A Query compiler for FPGAs," *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, Aug. 2009.

[9] N. Bandi, A. Metwally, D. Agrawal, and A. E. Abbadi, "Fast Data Stream Algorithms Using Associative Memories," in *Proc. of the 2007 ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007, pp. 247–256.

[10] *An Overview of Multiple CAM Designs in Virtex Family Devices. Application Note 201*, Xilinx Inc., Sept. 1999.

[11] R. Mueller, J. Teubner, and G. Alonso, "Data Processing on FPGAs," *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, Aug. 2009.

[12] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proceedings*, Knoxville, TN, USA, Nov. 1978, pp. 256–282.

[13] H. T. Kung and P. L. Lohman, "Systolic (VLSI) Arrays for Relational Database Operations," in *Proc. of the 1980 ACM SIGMOD Int'l Conference on Management of Data*, Santa Monica, CA, USA, May 1980, pp. 105–116.

[14] A. R. Hurson, C. R. Petrie, and J. B. Cheng, "A VLSI Join Module," in *Proc. of the 21st Hawaii Int'l Conference on System Sciences*, Kailua-Kona, HI, USA, 1988, pp. 41–49.

[15] Z. K. Baker and V. K. Prasanna, "Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs," in *Proc. 13th Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2005, pp. 3–12.