# Frequent Item Computation on a Chip

Jens Teubner, *Member, IEEE*, Rene Mueller, *Member, IEEE*, and Gustavo Alonso, *Member, IEEE*

**Abstract**—Computing frequent items is an important problem by itself and as a first step for several data mining algorithms. In this paper we explore how to accelerate the computation of frequent items using *field-programmable gate arrays (FPGAs)* with a threefold goal: increase performance over existing solutions, reduce energy consumption over CPU-based systems, and explore the design space in detail as the constraints on FPGAs are very different from those of traditional software-based systems.
In the paper we discuss three design alternatives, each one of them exploiting different FPGA features and each one providing different performance/scalability trade-offs. An important result of the paper is to demonstrate how the inherent massive parallelism of FPGAs can improve performance of existing algorithms but only after a fundamental redesign of the algorithms. Our experimental results show that, *e.g.*, the pipelined solution introduced in this paper can reach more than 100 million tuples per second of sustained throughput (four times the best available results to date) by making use of techniques that are not available to CPU-based solutions. Moreover, and unlike in software approaches, the high throughput is independent of the skew of the Zipf distribution of the input and at a far lower energy cost.

## 1 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 add up 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 data from memory or disk. Getting data in and out of the system often results in high *latency*, to the point that any algorithmic advantages may become irrelevant. In addition, a modern server CPU consumes over 100 Watts of *electrical power*, not counting necessary peripherals such as memory, disks, or cooling equipment.

In the search for possible solutions, *field-programmable gate arrays (FPGAs)* have been proposed as a way to extend existing computer architectures. They add 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 [1], [2] and/or embedded in critical data paths (network-CPU, disk-CPU) to reduce the load and amount of data that hits the CPU [3].

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, they can 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.

In this paper we tackle a basic data mining operation, the calculation of *frequent items* in a data collection, and show how it can be implemented using FPGAs. We achieve throughput rates of more than 100 million items per second, a rate four times higher than the best published results [4] 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).

Our main contribution is a highly efficient frequent item operator based on FPGAs. 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 ref-

*Authors are with the Systems Group, Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland.*
*E-mail:* `firstname.lastname@inf.ethz.ch`

erence for the performance of existing software solutions, we use the in-depth study of the frequent item problem by Cormode and Hadjieleftheriou [4].

The paper is organized as follows. The upcoming Section 2 formalizes the problem and briefly reviews the known solutions in software, before Section 3 gives a general background in FPGA technology. Sections 4 to 6 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. Different methods to issue queries to the frequent item circuits are presented in Section 7. In Section 8 we discuss the power aspects of the three different approaches in a full-system implementation and compare it with a traditional CPU-based solution. In Section 9, we relate our work to others', before we summarize in Section 10.

# 2 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 *$\phi$-frequent items* are those items in $S$ that occur more than $\phi N$ times. $\phi N$ 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 $\phi$, 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: *$\varepsilon$-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 $(\phi - \varepsilon)N < f_x \leq \phi N$.

## 2.1 Frequent Items in Software

The work of Cormode and Hadjieleftheriou has provided an in-depth comparison of such algorithms [4]. The comparison indicates that the *Space-Saving* algorithm by Metwally *et al.* [5] 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 [4] for details and characteristics of the other frequent item algorithms, which we will not cover here any further.

```
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 $S$ [5].

## 2.2 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, $b_{\mathsf{min}}$, 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 $x$.

The number of bins $k$ reserved for monitoring is a configuration parameter that can be used to trade accuracy for space. As detailed in [5], $\lceil 1/\varepsilon \rceil$ counters are required to find frequent items with an accuracy of $\varepsilon$. 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 [5].

## 2.3 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 $b_{\mathsf{min}}$ (line 6).

To be able to answer *(ii)* efficiently, existing implementations keep their bins physically organized according to their count values. Metwally *et al.* [5] propose the use of a linked list for this purpose. Their data structure, dubbed

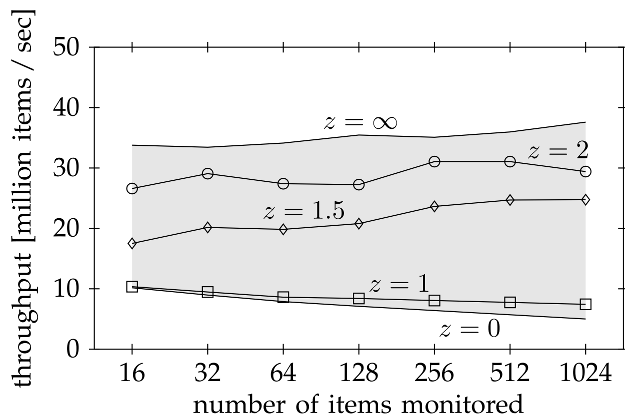



Fig. 2. Performance of *Space-Saving* [4] for different Zipf distributions $z \in \{0, 1, 1.5, 2, \infty\}$.

*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 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. [5] and [4] 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.

### 2.4 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 [4]). 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 [4] 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).

TABLE 1
Selected characteristics of FPGAs used in this paper.

| | XC5VLX110T | XC6VLX550T |
|---|---|---|
| lookup tables (6-to-1 LUTs) | 69,120 | 343,680 |
| flip-flops (1-bit registers) | 69,120 | 687,360 |
| slices (4 LUTs, 2 flip-flops) | 17,280 | 85,920 |
| block RAM (total kbit) | 5,328 | 22,752 |
| block RAM (# of 36 kbit blocks) | 148 | 632 |
| 18-bit multipliers | 64 | 864 |
| release year | 2006 | 2009 |

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 between the performances for $z = 0$ and $z = \infty$. The main cause for this are heap maintenance operations that are more frequent for data with smaller $z$ [14].

## 3 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.

### 3.1 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 recent Xilinx Virtex FPGAs [6]). 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 chips, for instance, include a number of BRAM blocks, each of which provides 36 kbit of fast storage.

Table 1 lists the available resources that typical candidates of the Xilinx Virtex-5 and Virtex-6 series can provide. Managing and properly balancing these resources is one of the challenges that an algorithm designer for FPGAs faces. Thereby, individual resource types are *not* independent. For instance, lookup tables and flip-flop registers are combined into so-called *slices*, and resource imbalances can only be traded *within* each slice.

Assessments in this work are based on two different generations of Xilinx Virtex FPGA chips. Model XC5VLX110T is a popular (though 4-year old) development chip, used, *e.g.*, in the Xilinx XUPV5-LX110T

university research and teaching platform. We used the XUPV5 board for testing and for power consumption measurements in Section 8. XC6VLX550T is a more advanced model from the current Xilinx Virtex-6 series. Fabricated on a 40 nm process, it allows apple-to-apple comparisons with current Intel processor lines. Results in the main body of this work are based on cycle-accurate simulations of a XC6VL550T chip.

FPGAs typically operate at clock frequencies that are significantly lower than those of general-purpose CPUs ($\approx$100 MHz). They are still competitive because tailor-made 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 (few Watts vs. $\approx$100 Watts for a modern CPU).

### 3.2 Block RAM: Dual-Ported and Configurable.

In this paper we exploit the high configurability of the available BRAM. In Virtex 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 heap structures in an efficient manner.

## 4 A SOFTWARE-LIKE SOLUTION FOR FPGAS

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 (our first design will thus be called SOFTWARE-LIKE) and explore how to best exploit the features available in an FPGA board.

When designing an FPGA circuit, we prefer a min-heap-based bin storage over the linked list-based *Stream-Summary* of [5], 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.

### 4.1 Content-Addressable Memory

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 run-time 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 [7].

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 [8] for CAM implementation alternatives).

### 4.2 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 3. 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.

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 them simultaneously. To do so, we configure Port B to a word size of 128 bit, as illustrated in Figure 3. 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

1. In some configurations not the full 36 kbit can be used.
2. The semantics for two conflicting write operations is undefined.

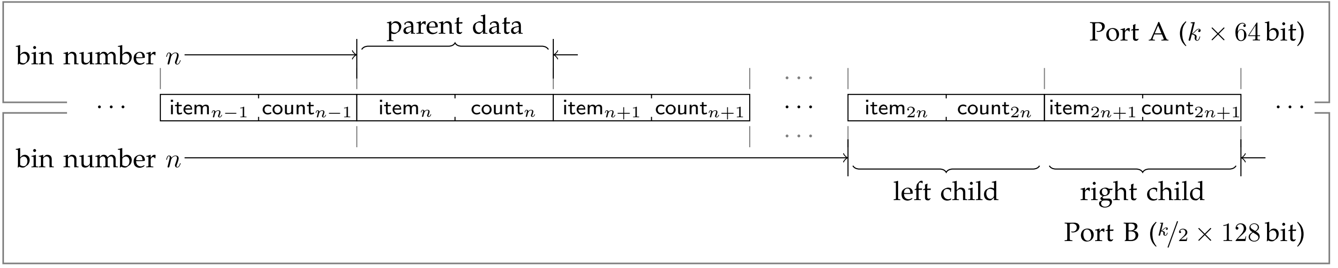
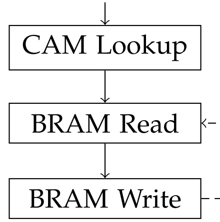



Fig. 3. Implementation of a min-heap using dual-ported BRAM. Bin number $n$ applied to the address lines of both ports will yield record $b_n$ at Port A and its two children $b_{2n}$ and $b_{2n+1}$ at Port B.

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.

### 4.3 Assembling a Frequent Item Circuit

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.

### 4.4 Evaluation

We implemented the circuit for SOFTWARE-LIKE for execution on the XC5VLX110T Virtex-5 FPGA.[3] We configured the circuit to monitor between 16 and 1,024 items

3. Support for content-addressable memories in the Xilinx tool chain is currently restricted to older chip series [9]. This prevented us from evaluating our circuit on the Virtex-6 chip.

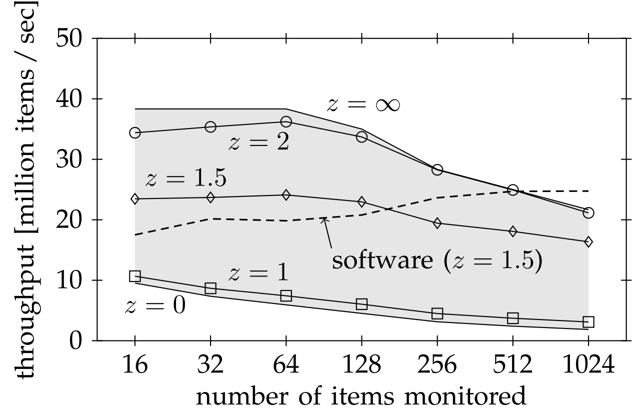



Fig. 4. Performance characteristics of SOFTWARE-LIKE implementation running on Virtex-5 chip. Input data sets follow a Zipf distribution $z \in \{0, 1, 1.5, 2, \infty\}$. Dashed line: performance of the software implementation ($z = 1.5$).

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 4.

*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.

*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

TABLE 2
FPGA chip resource consumption for the SOFTWARE-LIKE implementation (Virtex-5 chip). BRAM is the critical resource for this setup.

| config. | lookup tbls | | flip-flops | | BRAMs | | clock freq. |
|---|---|---|---|---|---|---|---|
| 64 bins | 1,236 | 1 % | 609 | 1 % | 13 | 8 % | 115 MHz |
| 128 bins | 1,583 | 2 % | 678 | 1 % | 21 | 14 % | 105 MHz |
| 256 bins | 2,126 | 3 % | 811 | 1 % | 37 | 25 % | 85 MHz |
| 512 bins | 3,007 | 4 % | 1,072 | 1 % | 69 | 46 % | 75 MHz |
| 1,024 bins | 4,911 | 7 % | 1,589 | 2 % | 133 | 89 % | 65 MHz |

together into a single functional unit. The growing complexity of this unit leads to longer *signal paths* and, hence, to longer *signal propagation delays*.

The longest signal path determines the maximum frequency at which the overall circuit can operate. While we were able to clock the smaller circuit instances (16 to 64 bins) at a rate of up to 115 MHz, a clock rate of 65 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 2, 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-6 chip mentioned in Table 1) include more BRAM blocks than the hardware we used for this experiment and could host configurations far beyond 1,024 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. SOFTWARE-LIKE requires between 3 cycles/item ($z = \infty$) and 35 cycles/item ($z = 0$ and 1,024 bins).

# 5 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.

## 5.1 Finding the Bin with Minimal count

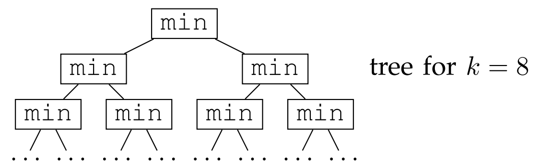
For locating the identifier min of the bin $b_{\mathsf{min}}$ that holds the smallest count value we use a parallel circuit. The search for this bin was the motivation to use a heap structure in the implementation SOFTWARE-LIKE.

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 $\mathsf{count}_a$ and $\mathsf{count}_b$, 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 $\mathsf{count}_{\mathsf{min}}$ (we detailed the inner workings of a similar operation in [10]).

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):




This search tree is going to be laid out spatially on the FPGA chip. It may be seen as a direct hardware implementation of a *reduce operation* [11], which has become a standard design pattern for parallel (software) algorithms.

Thanks to the search tree, it is no longer necessary to organize bins in a particular way (such as a min-heap) or to invest time into sorting. Using the tree, we

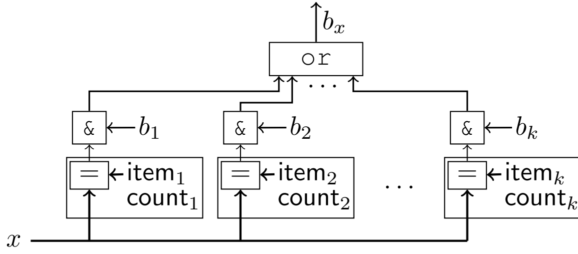



Fig. 5. All bins $b_i$ are compared with the input item $x$ in parallel to determine whether $x$ is currently monitored.

do not require any particular bin order and can exploit parallelism for efficient bin access.

## 5.2 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 5. 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 5) yields the address $b_x$ of the bin that currently monitors $x$—or zero if the item was not found.

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 [10], we demonstrated how this can improve the performance of an FPGA circuit.

## 5.3 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 6 (based on our Virtex-6 chip). 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 6 for reference). Data skew no longer affects the performance of the hardware circuit, a benefit that is particularly valuable

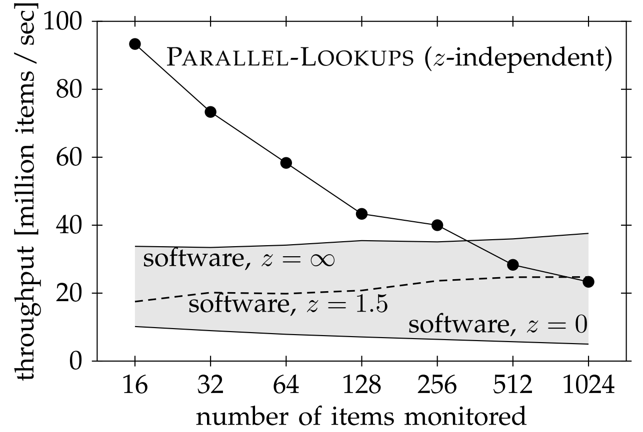



Fig. 6. Throughput of FPGA implementation PARALLEL-LOOKUPS (—●—) on Virtex-6. Performance of software implementation shown for reference (□/---).

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 fan-in 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 280 MHz, routing delays forced us to clock our largest instance (1,024 bins) at no more than 70 MHz.

The finite state machine that controls the operation of PARALLEL-LOOKUPS requires three cycles for each input tuple (independent of the data distribution). This yields a throughput of 93 million items per second for the 16-bin configuration, but only 23 million items per second for the configuration with 1,024 bins.

Note that the observable throughput results from a combination of the achievable clock frequency and the number of clock cycles spent for each item. In the SOFTWARE-LIKE implementation, this number was data dependent, and 3–35 cycles were required per input item ($z = 0$; 1,024 bins). In PARALLEL-LOOKUPS, by contrast, each item always takes 3 cycles to process.

*Chip Resource Requirements.* The high degree of parallelism directly affects the amount of logic resources required. As shown in Table 3, lookup tables are the

TABLE 3
Virtex-6 chip resource consumption and clock frequencies of implementation PARALLEL-LOOKUPS.

| configur. | lookup tables | | flip-flops | | BRAMs | | clock freq. |
|---|---|---|---|---|---|---|---|
| 64 bins | 11,899 | 3 % | 4,352 | 1 % | 1 | 1 % | 175 MHz |
| 128 bins | 23,696 | 6 % | 8,512 | 1 % | 1 | 1 % | 130 MHz |
| 256 bins | 47,744 | 13 % | 16,832 | 2 % | 1 | 1 % | 120 MHz |
| 512 bins | 93,850 | 27 % | 33,472 | 4 % | 1 | 1 % | 85 MHz |
| 1024 bins | 168,472 | 49 % | 66,752 | 9 % | 1 | 1 % | 70 MHz |

critical resource type in PARALLEL-LOOKUPS. The 1,024-bin design, for instance, consumed almost half of the available lookup tables in our Virtex-6 chip.

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 1,024 bins, we see a throughput reduction by more than a factor of four (because we had to operate the large configuration at a lower clock speed). The main cause for this performance degradation is the growing complexity of the on-chip wiring required to implement the two search trees.

## 6 PIPELINING FOR SCALABILITY

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.

### 6.1 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 7 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).

```
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. 7. Algorithm *Array*. Keep all processing and communication local to ensure scalability.

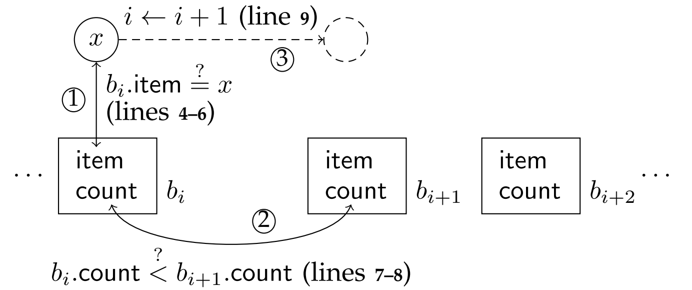



Fig. 8. The three processing steps of Algorithm *Array*.

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.

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 7.

We illustrate the three processing steps of Algorithm *Array* in Figure 8. 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

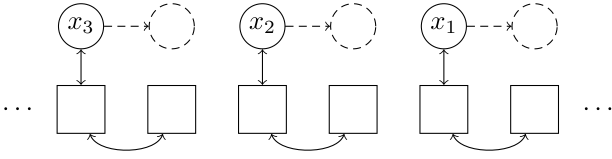



Fig. 9. Pipeline parallelism. Multiple input items $x_i$ follow one another and are processed in parallel.

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 coincides with the one typically requested by users or higher-level algorithms (such as a-priori [12]).

### 6.2 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.

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 9. 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.

### 6.3 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.

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.

### 6.4 Evaluation

We ran our implementation PIPELINE in the Virtex-6-based testbed for various numbers of bins. As can be seen in Figure 10, the throughput of this design remains almost constant for a large range of configurations. The only significant outlier to a throughput of 110 million items per second is the configuration with 1,024 bins. All configurations outperform the software solution by a factor of four and more.

This favorable scalability is a consequence of the pipeline-based design of the circuit. All circuit complexity, including longest signal paths, stays within each instance of the processing unit that we programmed. Signalling between instances is short, cheap, and independent of the length of the pipeline.

*Chip Resource Requirements.* Resource requirements for the PIPELINE implementation (listed in Table 4) 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

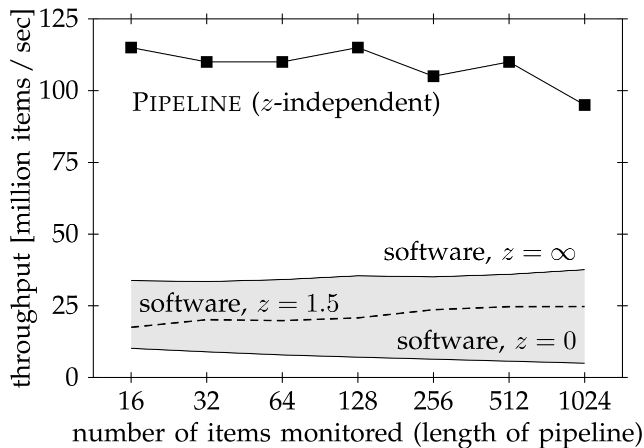



Fig. 10. Performance of hardware circuit PIPELINE on Virtex-6 chip (–■–). Performance of software implementation shown for reference (□/- - -).

TABLE 4
Virtex-6 resource consumption of PIPELINE.

| configuration | lookup tables | | flip-flops | | clock freq. |
|---|---|---|---|---|---|
| 64 bins | 16,846 | 4 % | 5,377 | 1 % | 110 MHz |
| 128 bins | 34,108 | 9 % | 10,529 | 1 % | 115 MHz |
| 256 bins | 71,302 | 20 % | 20,804 | 3 % | 105 MHz |
| 512 bins | 135,424 | 39 % | 41,441 | 6 % | 110 MHz |
| 1024 bins | 269,285 | 78 % | 82,657 | 12 % | 95 MHz |

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 8 and 9"), 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.

Resource analysis also reveals why we see a performance drop for the configuration with 1,024 bins. Since flip-flops and lookup tables have to be combined into *slices* on the physical chip, actual chip space usage is higher than the dominating resource type alone (here: lookup tables). In our particular case, the 1,024-bin design occupied 81 % of all available FPGA slices. It is known that chip utilization beyond 70–80 % will likely lead to contention on the on-chip routing interconnect [13]—with negative consequences on the achievable clock frequency and data throughput.

*Scalability.* A virtue of the pipeline-style circuit design is its scalability with circuit sizes. Our design has now truly become bound by the available amount of chip resources. With larger chips we would expect to see PIPELINE scale beyond 1,024 bins.

In an earlier report on this work [14], we were conducting experiments on a Virtex-5-generation chip and with an older version of the VHDL synthesis software. With that combination we had seen a constant maximum clock frequency of 80 MHz (maximum throughput of 80 million items/sec) for all configurations tested. The current 11.4 release of the Xilinx synthesis software uses more advanced optimization heuristics. They cause the slight throughput variations that can be seen in Figure 10.

Scalability beyond the available space of a single chip could be achieved by hardware solutions that daisy-chain multiple FPGA chips (such as the BEE3 [15] or Cube [16] systems). We would expect that PIPELINE would nicely map to such environments, though the necessary hardware 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. Our circuit sustains a throughput of 110 million tuples per second over a large range of size configurations. This corresponds to a processing time of 1 cycle/item at a clock rate of 110 MHz.

What makes PIPELINE most attractive, however, is its high potential to scale to large bin configurations, where the only limit is the available chip space.

## 7 QUERIES

While our main focus so far was the achievable throughput on the input side, real-world applications will likely also demand efficient *read access*—preferably without interference with update processing. Das *et al.* [17] recently demonstrated that a concurrent query workload can severely harm the efficiency of a parallel frequent item solution.[4]

Different query mechanisms are conceivable to support read access in our three FPGA design alternatives.

### 7.1 SOFTWARE-LIKE Design

In our SOFTWARE-LIKE design, the key information (items and their associated count value) is kept in a min-heap implemented using *Block RAM*. Answering a user query thus means to read out the Block RAM content.

The min-heap maintenance routine in Section 4.2 takes advantage of the *dual ported* feature of BRAM and reserves both ports to implement node swapping. Thus, in order to read out bin content in response to a user query, the input processing logic must release at least one of the two BRAM ports (and suspend its own work for that). In effect, each bin readout will cause a fixed processing delay and reduce overall update throughput.

4. In what Das *et al.* termed "independent design," a single query every 50,000 updates will already eat up almost 100 % of the available CPU time. No comparisons are given in [17] for their proposed "Coop" and "Shared" parallelization strategies.

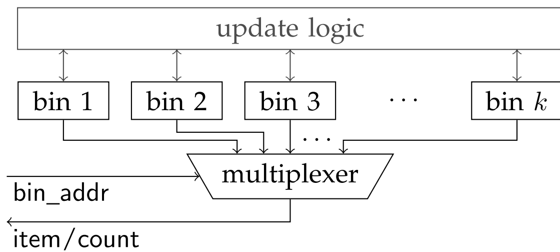



Fig. 11. Bin data storage in flip-flop registers (as done in designs PARALLEL-LOOKUPS and PIPELINE) allows bin readouts independent of the update process, *e.g.*, using a $k$-to-1 multiplexer.

In particular, each readout of the full BRAM content will take at least $k$ FPGA cycles in a $k$-bin design, plus few cycles to suspend and resume input processing.

The effective circuit slow-down thus depends on query frequency and the value of $k$. Taking the scenario of Das *et al.*, one query every 50,000 input items will slow down a 256-bin SOFTWARE-LIKE circuit by $\approx 0.6\,\%$.

From an implementation perspective, support for readout causes only little overhead. In particular, the small additional circuitry will not considerably affect resource consumption, which was dominated by the use of BRAM. The implementation that we measured in Section 4 already included readout support.

### 7.2 Register-Based Designs

Things look differently in the remaining two FPGA designs. PARALLEL-LOOKUPS and PIPELINE use flip-flop registers to store bin data. Registers are tightly woven into the FPGA fabric and can—conceptually—be accessed by an arbitrary number of concurrent readers.

This property opens up the possibility to perform readouts fully independent of any update processing, hence, without impact to update throughput. A basic implementation is illustrated in Figure 11: a reader states the index of a bin on the address line bin_addr of a $k$-way multiplexer and can then read the corresponding item and count values on the multiplexer output. The designs we presented in Sections 5 and 6 both contained exactly this circuitry.

*Semantics.* Though slim and elegant to implement, the readout strategy shown in Figure 11 has a severe semantical problem: there are no consistency guarantees for readouts that occur concurrent to update processing. Depending on the input data, inconsistencies might include wrong count values, duplicate or conflicting values for the same item,[5] or even missing bin data.

### 7.3 Consistent Readouts with PARALLEL-LOOKUPS

Our PARALLEL-LOOKUPS circuits was designed to keep the internal bin data consistent at any one moment in time. Inconsistencies are likely to happen, however, as soon as the readout spans a number of cycles during which data might change underneath. Two approaches are conceivable to prevent inconsistent readouts from happening:

*suspend input processing* Much like in the SOFTWARE-LIKE design, we could suspend the processing of input during data readouts.

*atomic snapshots* Take an atomic snapshot of all bin data, then return it to the user over time.

An implementation that uses the former alternative will suffer the same performance penalty that we discussed for the BRAM-based design before (about $k$ cycles for a full readout). Implementation-wise, only little logic has to be added to stop input processing during readouts.

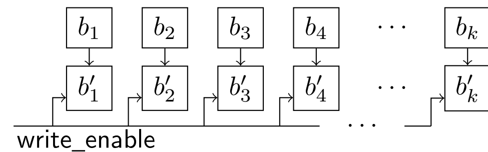



Fig. 12. Atomic snapshots. Raising the write_enable signal will atomically copy all $b_i$ into snapshot registers $b'_i$.

#### 7.3.1 Atomic Snapshots

The second alternative, *atomic snapshots*, is more interesting from an architectural perspective. The gathering of such snapshots is well supported by FPGA hardware. Figure 12 illustrates the idea.

Next to each of the active data bins $b_1, \ldots, b_k$, we place *snapshot registers* $b'_1, \ldots, b'_k$, which can hold a snapshot copy of the respective bin. The data input lines of the snapshot registers are wired to the output of the corresponding active bins. This output will be copied into the snapshot registers at the instant when the write_enable input line is raised. Thus, by connecting all snapshot registers to a common write_enable signal, a complete snapshot can be taken within a single FPGA clock cycle.

The idea of snapshotting works well in principle and is straightforward to implement. Unfortunately, its price in terms of FPGA resource consumption is significant. Snapshot copies will double the number of flip-flop registers required. Since snapshots are "unrelated logic" with respect to the main processing circuit, the required flip-flops will have to be placed on their own FPGA slices and thus significantly increase the overall chip space consumption.

### 7.4 Readouts with PIPELINE

Implementation strategy PIPELINE once again offers interesting advantages. As an item travels through the circuit, the state it leaves behind will be a consistent set of item/count pairs (we exploited this property to concurrently process multiple items back-to-back in a single circuit).

5. Duplicate items might occur if an item, say $y$, forces the eviction of item $x$ from its current bin $b_i$; some items later, $x$ might again occur in the input stream and now be placed in some other bin $b_j$.

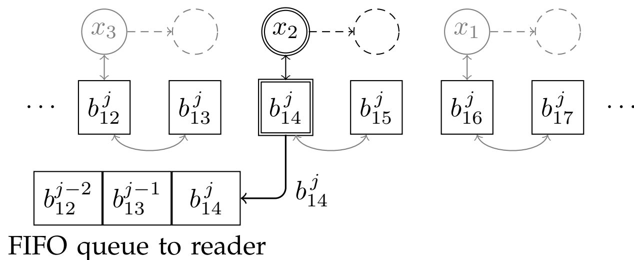



Fig. 13. Readout for design PIPELINE. $b_i^j$ denotes data of bin $b_i$ as it was written in clock cycle $j$.

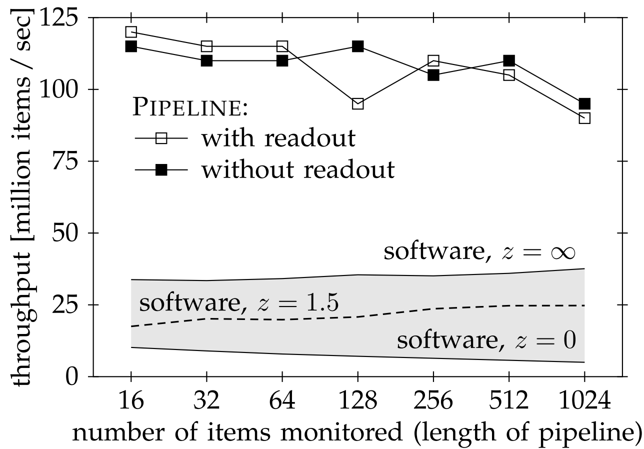



Fig. 14. Performance of hardware circuit PIPELINE with and without explicit readout functionality.

This means that, at any moment, we can "attach" to an item that enters the circuit, follow it while it travels through the processing pipeline, and emit each left-behind item to an outside reader.

Figure 13 illustrates this idea. Travelling item $x_2$ was attached for a data readout. As it passes each bin $b_i$ in clock cycle $j$, it copies the left-behind bin content $b_i^j$ to a FIFO queue. This procedure populates the FIFO with consistent information for all bin contents. All items that were processed before and including $x_2$ will be reflected in the data written to the FIFO, but none of those that are processed after $x_2$.

*Scalability and Performance.* By design, data readouts based on item attachment do not affect or slow down input processing. The necessary circuitry to implement the functionality could, however, have negative impact on the on-chip signal routing or lead to high resource consumption.

Figure 14 illustrates the achievable throughput that we measured PIPELINE circuits with or without the additional readout logic. Overall, the add-on functionality does not significantly affect data throughput: observable throughput changes are within the increments in which we measured throughput (5 million items per second). In fact, for several circuit instances, the optimizer found slightly better physical designs with the readout functionality in place.

TABLE 5
Readout functionality and chip resource consumption (PIPELINE design; 1,024 bins).

| resource | without readout | with readout | change |
|---|---|---|---|
| lookup tables | 269,285 | 266,327 | − 1 % |
| flip-flops | 82,657 | 100,444 | + 22 % |
| slices | 69,433 | 69,482 | ± 0 % |

*Chip Resource Requirements.* A look at the consumed resources confirms that variations are mainly an effect due to different optimizations taken by the physical design optimizer. Interestingly, as shown in Table 5, the design requires less lookup tables if readout functionality is added.

*Implementation Aspects.* In Table 5, only the amount of flip-flop registers used by the designs differs significantly between the two alternatives. This is a consequence of a particular implementation aspect.

The way to implement attachment is to use a *multiplexer* that consecutively retrieves the contents of each bin (at the pace of the traveling item it is attached to). Implementing such a multiplexer naïvely as illustrated in Figure 11 would cause new trouble with signal propagation delays. Essentially, such a multiplexer will lead to the same high fan-in problems that ruined scalability in the PARALLEL-LOOKUPS design of Section 5.

This can be avoided by *staging* the multiplex operation into a tree of multiplexers of which each only has a reasonably small fan-in. During readout, all multiplexers write their output into a local flip-flop registers from where it is passed on to the next stage in the following clock cycle. In effect, the content of each bin is *pipelined* to the root of the staging tree. Once again the use of pipeline parallelism helps to guarantee scalability. Once again the cost of it is some increase in the number of flip-flop registers consumed.

## 8 POWER CONSIDERATIONS

FPGAs can also provide advantages over general purpose CPUs in terms of power consumption. In this section we determine the power consumption of the FPGA designs and compare it with the power required by a traditional CPU-based system. We embed the frequent item circuits into a complete FPGA system on our Xilinx XUPV5-LX110T board (Figure 15). We use a gigabit Ethernet link to send the stream data into the circuit. To this extend we implement a UDP/IP engine on the FPGA. The engine receives UDP datagrams and extracts 32-bit tuples from the datagrams. A dedicated command datagram is used to trigger the readout of the bins. The bin data, *i.e.*, item and count, are sent back as UDP datagrams from our UDP/IP engine. This setup allows us to (1) verify that indeed data can processed at full network speed and (2) get a full end-to-end system from which we can obtain meaningful results for the

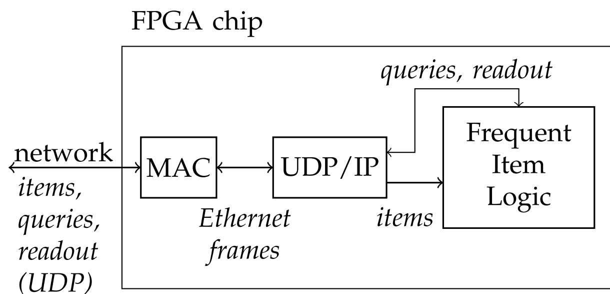



Fig. 15. Frequent item logic connected to a gigabit Ethernet network for processing data sent via UDP.

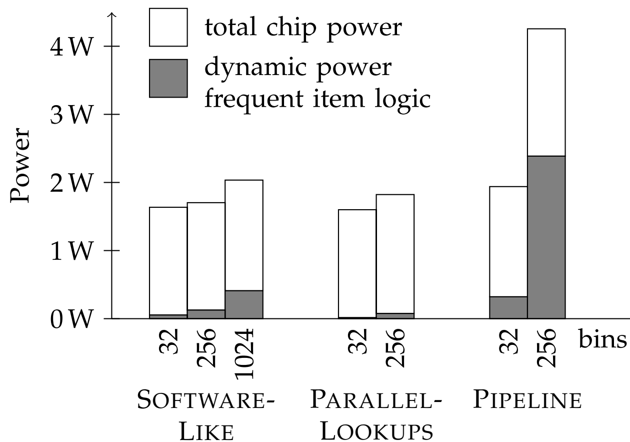



Fig. 16. Comparison of chip power consumption of different FPGA implementations.

total consumption that can be compared to traditional computer systems.

For a 1 Gb/sec link the upper bound of the data rate is 119 MiB/sec or equivalently 31.25 million items/sec. In order to handle this load using the PIPELINE implementation it is sufficient to clock the frequent item logic at 31.25 MHz. The throughput numbers shown earlier in Table 4 are significantly higher (> 95 million items/sec). On hardware with additional physical network interfaces, this would allow us to process data from up to three concurrent input streams.

Figure 16 shows the power estimates for the FPGA chip determined by the Xilinx XPower Analyzer. The total chip power includes both static power dissipation (due to quiescent current losses) as well as dynamic power (caused by signal switching). In Figure 16 we highlight (shaded) dynamic losses that caused by the frequent item circuit alone. It can be seen that with the increasing number of bins the total chip power as well as the power for the frequent item logic increases due to the larger circuit size. The large number of RAM blocks used in the SOFTWARE-LIKE implementations consumes more power than the register-based PARALLEL-LOOKUPS version. The comparatively high power is required for the PIPELINE implementation can be explained by relatively large chip utilization and the higher circuit activity. For 256 bins, PIPELINE requires 97 % of the total Virtex-5 chip space compared to 68 % for PARALLEL-LOOKUPS. For the former several bins can be updated in a single clock cycle whereas for the later at most one bin is active in an given cycle. The total power consumption is less than 4.5 W for all FPGA designs. This power calculation includes only the FPGA chip. We measure the actual power consumption of the entire FPGA board using a power meter attached to the wall plug. For all designs we measure an overall power consumption of 11 W for the entire board.

As a reference for the software implementation we use a mobile Core2 Duo T9550 2.66 GHz processor, as it provides a good trade-off between performance and power. For this CPU Intel specifies a thermal design power (TDP) of 35 W which can serve as an upper bound for the total power consumption. Compared to the 4.5 W total power for the FPGA this results in a 7.8-fold power reduction. We also measure the total power consumption of 43 W for the Core2 Duo system (Lenovo W500) at the wall plug when running the software implementatation [4]. This corresponds to an overall power reduction of $4\times$ and shows that FPGAs can provide a significant power advantage even over an already power-optimized mobile platform.

## 9 RELATED WORK

Cormode and Hadjieleftheriou [4] 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. In a more recent work, Das *et al.* [17] parallelized the calculation of frequent items for multi-core systems. Their achieved performance ($\approx$ 5 million items per second) demonstrates that *Space-Saving* is difficult to accelerate using commodity multi-core CPUs.

We would like to mention the work of Bandi *et al.* [7] explicitly. 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

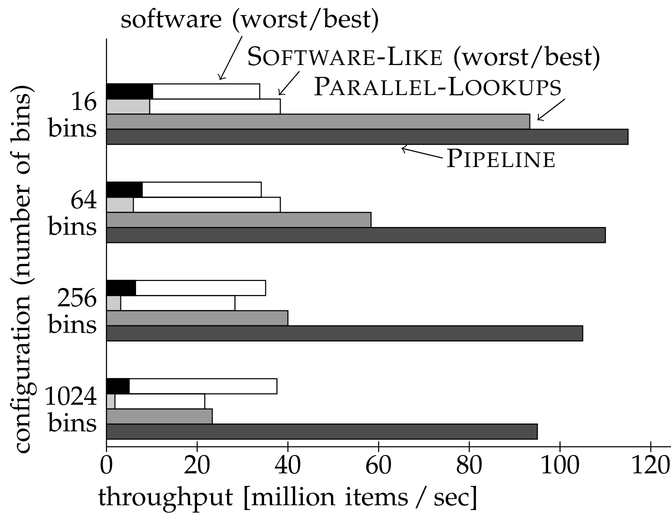



Fig. 17. Throughput comparison of all approaches discussed.

a design guide for hardware circuits in the late 70s. Kung and Leiserson [18] discovered systolic arrays as a very efficient, yet simple-to-manufacture type of circuits to perform matrix multiplications. Later, Kung and Lohman [19] and Hurson *et al.* [20] used the same principle to implement basic database functionality.

The idea of systolic arrays was also picked up by Baker and Prasanna [21], 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.

## 10 Conclusions

In this paper we presented three different FPGA designs that implement the search for frequent items: SOFTWARE-LIKE, PARALLEL-LOOKUPS, and PIPELINE. The different designs illustrate the possibilities and trade-offs inherent in FPGAs, as well as their advantages over software-based solutions. Figure 17 summarizes the throughput of each one of the techniques discussed in the paper, including the software implementation of *Space-Saving* [5], [4] as a baseline.

Our results show 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 110 million items per second. Moreover, the throughput is independent of the distribution of the input data, a major difference over software-based solutions. PIPELINE showed far better performance than any known software-based solution.

More interesting, however, are the different notions of *parallelism* that we studied in this work and how they behave with with respect to scalability. The frequent item problem is a known challenge for parallel execution [17] and straightforward parallelization strategies like PARALLEL-LOOKUPS do not exhibit the desirable scalability with problem sizes. The last of our three designs, circuit PIPELINE, is based on *pipelining* and excels in both scalability and performance.

## References


[1] D. Greaves and S. Singh, “Kiwi: Synthesis of FPGA circuits from parallel programs,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

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

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

[4] G. Cormode and M. Hadjieleftheriou, “Finding frequent items in data streams,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[5] 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, Sep. 2006.

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

[7] 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, Jun. 2007, pp. 247–256.

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

[9] *Content-Addressable Memory v6.1*, Xilinx Inc., Sep. 2008.

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

[11] G. E. Blelloch, “Prefix sums and their applications,” in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann, 1993.

[12] 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)*, Sep. 1994, pp. 487–499.

[13] A. DeHon, “Balancing interconnect and computation in a reconfigurable computing array (or, why you don’t really want 100 % lut utilization),” in *Proc. of the Int’l Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 1999, pp. 125–134.

[14] J. Teubner, R. Mueller, and G. Alonso, “FPGA acceleration for the frequent item problem,” in *Proc. of the 26th Int’l Conference on Data Engineering (ICDE)*, Long Beach, CA, USA, Mar. 2010.

[15] J. Davis, C. Thacker, and C. Chang, “BEE3: Revitalizing computer architecture research,” Microsoft Research, Tech. Rep. MSR-TR-2009-45, 2009.

[16] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. H. W. Leong, “CUBE: A 512-FPGA cluster,” in *Proc. of the Southern Programmable Logic Conference (SPL)*, São Carlos, Brazil, 2009.

[17] S. Das, S. Antony, D. Agrawal, and A. E. Abbadi, “Thread cooperation in multicore architectures for frequency counting over multiple data streams,” *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, Aug. 2009.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.