

# An FPGA-Based Accelerator for Frequent Itemset Mining

YAN ZHANG, FAN ZHANG, ZHEMING JIN, and JASON D. BAKOS

University of South Carolina

2

In this article we describe a Field Programmable Gate Array (FPGA)-based coprocessor architecture for Frequent Itemset Mining (FIM). FIM is a common data mining task used to find frequently occurring subsets amongst a database of sets. FIM is a nonnumerical, data intensive computation and is used in machine learning and computational biology. FIM is particularly expensive—in terms of execution time and memory—when performed on large and/or sparse databases or when applied using a low appearance frequency threshold. Because of this, the development of increasingly efficient FIM algorithms and their mapping to parallel architectures is an active field. Previous attempts to accelerate FIM using FPGAs have relied on performance-limiting strategies such as iterative database loading and runtime logic unit reconfiguration. In this article, we present a novel architecture to implement Eclat, a well-known FIM algorithm. Unlike previous efforts, our technique does not impose limits on the maximum set size as a function of available FPGA logic resources and our design scales well to multiple FPGAs. In addition to a novel hardware design, we also present a corresponding compression scheme for intermediate results that are stored in on-chip memory. On a four-FPGA board, experimental results show up to 68X speedup compared to a highly optimized software implementation.

Categories and Subject Descriptors: B.1.2 [Control Structures and Microprogramming]: Control Structure Performance Analysis and Design Aids—*Automatic Synthesis*; B.2.4 [Arithmetic and Logic Structures]: High-Speed Arithmetic; B.5.2 [Register-Transfer-Level Implementation]: Design Aids—*Automatic synthesis*; B.7.1 [Integrated Circuits]: Types and Design Styles—*Algorithms implemented in hardware*; C.1.3 [Processor Architectures]: Other Architecture Styles—*Data-flow architectures, Heterogeneous (hybrid) systems, Pipeline processors*

General Terms: Performance

Additional Key Words and Phrases: Frequent itemset mining, data mining, Eclat, data intensive, co-processor, high performance computing, reconfigurable applications, reconfigurable

## ACM Reference Format:

Zhang, Y., Zhang, F., Jin, Z., and Bakos, J. D. 2013. An FPGA-based accelerator for frequent itemset mining. *ACM Trans. Reconfig. Technol.* 6, 1, Article 2 (May 2013), 17 pages.

DOI:<http://dx.doi.org/10.1145/2457443.2457445>

## 1. INTRODUCTION

Advancements in semiconductor technology, the increasing prevalence of mobile processing and data capture, and the decreasing cost of storage has made large-scale databases increasingly common. For example, web logging, credit card transactions, and genomic sequencing can quickly generate gigabyte- to terabyte-sized datasets. The need for data mining techniques that can process large datasets is becoming increasing urgent.

---

This work is supported by the National Science Foundation, under grant CNS-0844951.

Authors' address: Y. Zhang, F. Zhang, Z. Jin, and J. D. Bakos, Department of Computer Science and Engineering, University of South Carolina; email: [jbakos@cse.sc.edu](mailto:jbakos@cse.sc.edu).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2013 ACM 1936-7406/2013/05-ART2 \$15.00  
DOI:<http://dx.doi.org/10.1145/2457443.2457445>

Correlation-based data mining is a common method to extract information from large-scale databases. A common example is frequent itemset mining (FIM) or association rule mining. Social networking sites use FIM to suggest friends and serve advertisements to users [Fukuzaki et al. 2010]. In a more widely cited example, supermarkets use FIM to determine which products are often purchased together. A popular anecdote describes how one supermarket used FIM on their transaction database and was surprised to discover that customers who purchased diapers were more likely than other customers to also purchase beer [Witten and Frank 2005]. The supermarket later concluded that this behavior was caused by husbands who purchase beer after being sent out at night by their wives to purchase diapers. Discovering the relationship among purchased items can help supermarket management develop promotions and decide which items to place together on the shelf.

FIM is an expensive computation when the input data set is very large or sparse or when performed using a low frequency threshold. As such, there is a need for optimized FIM techniques, including the implementation and optimization of FIM algorithms for parallel and heterogeneous processing platforms. In this article, we implement FIM on an FPGA-based coprocessor. FPGAs are widely used to accelerate scientific and media-related applications, such as numerical linear algebra [Zhou and Prasanna 2008], signal processing [Heighton et al. 2006], and biological sequence alignment [Alachiotis et al. 2011], but are less commonly used for graph-based applications such as frequent itemset mining.

We summarize the contributions of this work as the following.

- As far as we know, this is the first FPGA implementation of the Eclat FIM algorithm.
- Our accelerator can handle large datasets without iterative database loading or logic reconfiguration required by previous FPGA FIM designs.
- Our accelerator design employs a novel on-chip caching and associated data compression technique to reduce frequency of off-chip accesses for intermediate results.
- Our design is scalable to multiple FPGAs.

This remainder of this article is organized as follows. Sections 2 and 3 introduce the background and related work. Section 4 presents the algorithm and the data structure used in our design. Section 5 describes our accelerator design. Section 6 describes a compression scheme designed to take advantage of on-chip memory. Section 7 lists our experimental results and Section 8 concludes the article.

## 2. BACKGROUND

### 2.1. Problem Formulation

Using a supermarket metaphor, items represent merchandise—individual items for sale. A *transaction*, sometimes called a *basket*, is a receipt, or a combination of items purchased together. An *itemset* is a subset of the items that appear on enough receipts to exceed the specified *threshold*. The threshold is often expressed as a percentage of the total number of transactions. For example, a threshold of 0.02 would mean that any item appearing in at least 2% of all transactions is considered frequent. The computational cost of FIM grows as the threshold is decreased. Regardless of implementation, the results given by any traditional FIM algorithm is always exact given a specified dataset and threshold. Most FIM algorithms find frequent itemsets by searching for small-sized frequent itemset candidates and then iteratively growing the candidates to search for larger ones. The FIM algorithm Apriori adopts this approach.

FIM implementations generally sort the transactions in the database according to item number, as this generally results in more performance gain than sacrificed in

the cost of sorting. This simplifies the FIM algorithm and allows duplicate items to be easily filtered out.

## 2.2. FIM Algorithms

Frequent itemset mining algorithms have long been a subject of study. In 1993 Agrawal et al. introduced the problem [Agrawal et al. 1993] and later published a seminal paper in which they developed a FIM algorithm named Apriori [Agrawal and Srikant 1994]. Apriori consists of iteratively performing three stages: *candidate generation*, *support counting*, and *candidate pruning*. Apriori uses a breadth-first search to traverse the space of itemset candidates, where each level of the tree represents an itemset generation.

Since its introduction, there has been continued work to develop faster algorithms based on Apriori [Bodon 2006; Goethals and Zaki 2003]. Most of those are based on improving the central data structure for storing candidate itemsets such as the hash table and trie. Bodon showed that the trie approach outperforms the hash table in terms of running time, memory requirements, and sensitivity to parameters [Bodon and Ronyai 2003]. Bodon went on to develop a trie-based implementation [Bodon and Ronyai 2003] that outperformed the Apriori of Borgelt [2003] and Goethals and Zaki [2003].

The Eclat algorithm was first introduced by Zaki for FIM [Zaki 2000]. Unlike Apriori, it uses a depth-first traversal to search the candidate space recursively. However, Eclat does not store candidates so no trie structure is necessary. Eclat is more efficient in memory and provides better performance compared to Apriori.

FP-growth is another popular FIM algorithm introduced by Han et al. [2000]. FP-growth compresses the database into a data structure called a Frequent-Pattern tree (FP-tree) to avoid repetitive database scans, requiring only two scans of the database. Unlike Apriori, FP-Growth does not perform candidate generation and support counting. However, its disadvantage is that the FP-tree can grow very large. This makes FP-Growth an extremely fast algorithm but its high memory requirement makes it impracticable for large databases.

## 3. RELATED WORK

There are several examples in the literature of FPGA-based accelerators for FIM. Perhaps the earliest was an Apriori implementation that mapped Apriori to a systolic array [Baker and Prasanna 2005]. In the systolic array, each processing unit has three different modes corresponding to the three tasks in Apriori: candidate generation, candidate pruning and support counting. This is an elegant solution and maps naturally to an FPGA. However, because there is not enough storage on the FPGA to store all candidates for any realistically sized input size, multiple passes of the algorithm are required to process the dataset. Also, the FPGA must be reconfigured after each pass. The overhead of reprogramming is significant, especially for large datasets. In a later version, the same authors improved the most time-consuming stage, support counting, by introducing Content Addressable Memory (CAM) [Baker and Prasanna 2006]. The bitmapped CAM can parallelize support counting because a multiple-item set can be compared in single cycle instead of multiple cycles as in the earlier paper. Although this approach improved performance, it still requires multiple iterations.

Wen et al. developed an accelerator to address the reconfiguration and multiple-pass problem, using the DHP algorithm [Park et al. 1997] to develop a hash-based systolic architecture [Wen et al. 2008]. Specifically, they used a systolic array for support counting and candidate generation but added a trimming filter and a hash table filter. The hash table is used to speed up the candidate pruning and the trimming filter is used to trim the transaction set and reduce its size after each pass. With the help of both

schemes, they achieved the same performance with 25 cells of their systolic array as with 800 cells used in Baker et al.'s approach.

Thoni et al. developed a further improvement to the systolic array-based Apriori, where each cell in the systolic array uses a bitmap memory that stores multiple candidates and achieves a 40% improvement over Baker's version [Thoni and Strey 2009]. This approach is similar to using Apriori and the bitvector data structure. However, multiple passes are still necessary for datasets of realistic size.

Sun et al. [2008] and Sun and Zambreno [2011] developed an FPGA-based hardware implementation of FP-growth. In their approach, they build the FP-tree using a systolic tree. Due to limited logic resources, a Virtex 5 XC5VLX330 (the largest 5th generation Xilinx FPGA) can fit a systolic tree with no more than  $depth = 5$  and  $children = 5$ . To accommodate a larger FP-tree, the authors suggest a database projection algorithm to divide the tree into many small trees that can fit in single FPGA. This extra preprocessing step requires overhead that results in low performance for some datasets.

#### 4. ALGORITHM AND DATA STRUCTURE

Due to the limitations of systolic array-based FIM designs—especially for mining large databases—we take a radically different approach. First, instead of using a systolic array, our processing element (PE) design consists of a FSM-based controller connected to multiple on-chip memories. The controller performs the Eclat algorithm by performing a depth-first search using an on-chip stack. The controller generates its own sequence of addresses for accessing off-chip memory when performing support counting as it traverses the search tree. Our design uses a vertical bitvector representation for both the input dataset and candidates.

##### 4.1. Eclat Algorithm

Eclat is an abbreviation for Equivalence CLass Transformation. While Apriori is a breadth-first search in the sense that all frequent candidates of each generation are processed before the algorithm moves to the next generation, Eclat uses a depth-first approach where candidates are recursively generated.

Algorithm 1 describes Eclat. Line 1 initializes the set of candidates to 1-itemsets. Line 2 to line 11 is the recursive function call to find frequent itemsets. Lines 4 to 10 are the loop to scan all input candidates. In lines 6 to 8, a pair of parents' candidates generates a new candidate  $p$ , and then the support is calculated. If the candidate is frequent, new candidates will be generated and added to candidate set  $C^{i+1}$  and function Eclat() is recursively called.

##### 4.2. Data Representation

The original implementations of Apriori used a traditional *horizontal representation* for the transactions. In this case, each transaction is represented as a sorted list of items. The vertical representation is the transpose (corner turn) of this representation, where each item is associated with a list of corresponding transactions that contain the item. In other words, the database is stored according to item rather than transaction number. Vertical representation generally offers one order of magnitude performance gain relative to the same algorithm using the horizontal format because it reduces the volume of I/O operations and avoids repetitive database scanning.

Table I depicts the horizontal format for an example dataset. Table II shows the equivalent vertical format. Table III shows the equivalent vertical bitvector format, which is a variation of the vertical format where a binary sequence is stored for each item where each bit represents a transaction and is set to one if the item is contained

**Algorithm 1:** Eclat

---

**Input:**  $C^1$  – Frequent Candidate for generation 1  
**min\_sup** – Minimum support  
**Output:**  $U_i C^i$  – All Frequent Candidates

---

```

1: Initialize  $C^i$  to  $C^1$ 
2: Eclat( $C^i$ ) :
3:    $C^{i+1} = \emptyset$  ;
4:   for all  $c_j^i \in C^i$  do
5:     for all  $c_k^i \in C^i$ , with  $k > j$  do
6:       if  $(|c_j^i \cap c_k^i| = i - 1$ 
7:          $p = c_j^i \cup c_k^i$ 
8:         support_counting( $p$ );
9:         if support( $p$ )  $\geq$  min_sup
10:           $C^{i+1} = C^{i+1} \cup p$ ;
11:   for all  $C^{i+1}$  not empty do Eclat( $C^{i+1}$ )

```

---

Table I. Example Horizontal Dataset Representation

Transaction ID	Item			
1	A	B	C	
2	A	C	E	
3	A	E	F	G
4	B	C	E	G
5	D	F		
6	A	C	D	E

Table II. Corresponding Vertical Tidset Representation

Item	Transaction ID			
A	1	2	3	6
B	1	4		
C	1	2	4	6
D	5	6		
E	2	3	4	6
F	3	5		
G	3	4		

within the corresponding transaction. The vector width is fixed for all items. For dense transaction data, the size of vertical bitvector is smaller than the vertical tidset format's. This is an advantage in terms of both run time and memory space. The vertical bitvector format becomes inefficient in terms of memory space for sparse datasets. However, support counting with bitvectors can be performed using bitwise logical operations, which allow for a large amount of exploitable fine-grain parallelism. For this reason we use this format in our accelerator.

## 5. ACCELERATOR ARCHITECTURE

Figure 1 illustrates our accelerator framework, which is comprised of a controller, two FIFOs, and two simple functional units that perform a bitwise AND and a population count operation.

Table III. Corresponding Vertical Bitvector Representation

Item	Transaction IDs					
	1	2	3	4	5	6
A	1	1	1	0	0	1
B	1	0	0	1	0	0
C	1	1	0	1	0	1
D	0	0	0	0	1	1
E	0	1	1	1	0	1
F	0	0	1	0	1	0
G	0	0	1	1	0	0

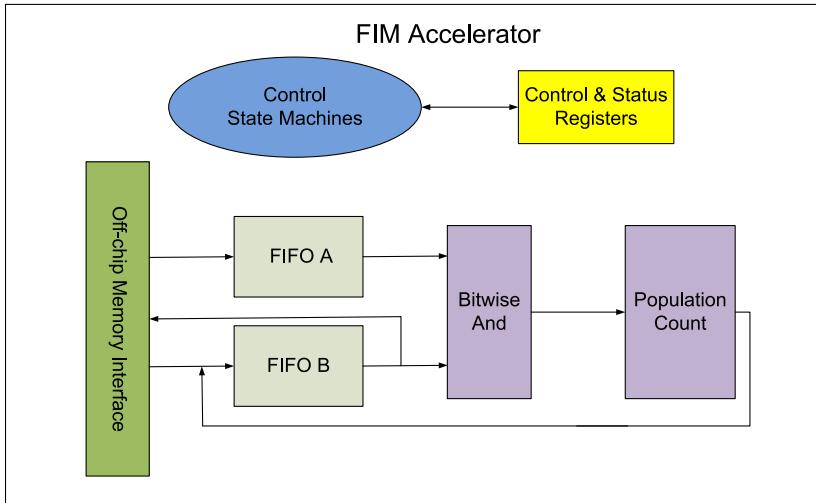


Fig. 1. The FIM accelerator architecture.

Figure 2 and Table IV illustrates how the controller and FIFOs implement Eclat. The accelerator reads bitvectors as a sequence of words from external memory and buffers them in FIFO A and FIFO B. The FIFOs provide source and intermediate bitvectors. Their contents are fed into a data path that is comprised of a bitwise AND followed by a population count unit and an accumulator. The population count unit counts the number of one-bits and accumulates this value into a value that represents the support of the current candidate. The result of the bitwise AND is also fed back into FIFO B. The intermediate result stored on FIFO B can also be stored into off-chip memory and used by a future candidate.

Figure 2 considers an example four-item database. The search tree that is rooted to a candidate consisting of item 1 is shown in Figure 2. The second level of the tree shows search states that correspond to candidates {1,2}, {1,3}, and {1,4}. The third level shows search states that correspond to candidates {1,2,3}, {1,2,4}, and {1,3,4}. There are two candidates that have a sub-threshold appearance rate in the database, which are crossed out in the figure.

Table IV shows all the steps taken during the depth-first search. The second and third columns show the contents of FIFO A and B. The third column is the result stored back in FIFO B. The fourth column indicates if the candidate is valid (has an appearance rate greater than the threshold). The fifth and sixth tell us whether the result should be output or flushed. Below we describe each step in detail.

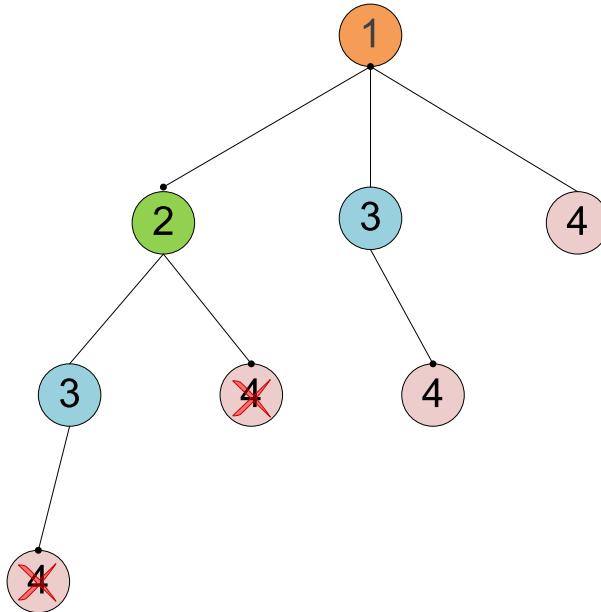


Fig. 2. An example for FIFO and memory control.

Table IV. FIFO Control Sequence for Example in Figure 2

Step	FIFO B	FIFO A	Result (in FIFO B)	Valid	Output	Flush
1	1	2	1 & 2	Yes	Yes	No
2	1 & 2	3	1 & 2 & 3	Yes	Yes	No
3	1 & 2 & 3	4	1 & 2 & 3 & 4	No	No	Yes
4	1 & 2	4	1 & 2 & 4	No	No	Yes
5	1	3	1 & 3	Yes	Yes	No
6	1&3	4	1 & 3&4	Yes	Yes	Yes
7	1	4	1&4	Yes	Yes	Yes

*Step 1.* FIFO B loads bitvector 1 and FIFO A reads bitvector 2 from off-chip memory. The intermediate result is the bitwise AND of bitvectors 1 and 2, which we represent as 1&2. For example, if there are 6 total transactions, item 1 appears in transactions 1, 2, 3, and 4, item 2 appears in transactions 1, 2, 4, and 6, and threshold = 2, the result is 111100 AND 110101 = 110100 and has a population count of 3, thus the candidate is valid. This result is stored back to FIFO B and output to off-chip memory as an intermediate result that may later be loaded for a parallel branch.

*Step 2.* FIFO B uses the previous intermediate result 1&2. FIFO A loads bitvector 3 from off-chip memory. The result is the bitwise AND of bitvector 1&2 and bitvector 3, represented as 1&2&3. This candidate is also valid so it is sent into FIFO B and output to off-chip memory.

*Step 3.* FIFO B uses the previous step's result 1&2&3. FIFO A loads bitvector 4 from off-chip memory. The result is represented as 1&2&3&4. The candidate is invalid and is not sent to off-chip memory. Since this is a leaf, FIFO B is flushed.

*Step 4.* FIFO B loads intermediate result 1&2 from off-chip memory and FIFO A loads bitvector 4 from off-chip memory. The result is 1&2&4. The candidate is invalid

Table V. Resource Utilization Summary

Resource	128-bit Accelerator		256-bit Accelerator		Full Design	
	Utilization	Percentage	Utilization	Percentage	Utilization	Percentage
Combination ALUTs	6,908/ 203,520	3.4%	7,994/ 203,520	3.5%	38,522/ 203,520	19%
Logic registers	5,991/ 203,520	2.9%	7,033/ 203,520	3.5%	44,145/ 203,520	22%
Block Memory bits	1,382,144/ 15,040,512	9%	1,688,320/ 15,040,512	11%	5,939,472/ 15,040,512	39%
DSP block 18 bit elements	8/768	1%	8/768	8%	24/768	3%

so it is not stored to off-chip memory. The recursion has now reached the leaf so FIFO B is flushed and the design moves to next branch of the tree.

*Step 5.* FIFO B loads bitvector 1 and FIFO A loads bitvector 3 from off-chip. The result is 1&3. The candidate is valid. FIFO B is flushed but does not store the last intermediate result to off-chip memory since it will not be needed in later branches (based on the property of Eclat trees where branch lengths decrease from left to right).

*Step 6.* FIFO B uses previous result 1&3. FIFO A loads bitvector 4 from off-chip. The design performs the bitwise AND to compute the result 1&3&4. The candidate is valid. This branch terminates so FIFO B is flushed.

*Step 7.* FIFO B loads bitvector 1 and FIFO A loads bitvector 4 from off-chip. The result is 1&4. The candidate is valid. The design stores the result back to FIFO B. The design has now completed its workload and reinitializes the FIFOs.

The FIFOs are flushed to clear intermediate results that are not needed in the next step of computation. There are two situations where this occurs. First, the DFS search reaches the leaf node, and the next step will start a new branch. Step 6 and Step 7 in the example shown in Table IV depict this situation. The second situation is when the search result is invalid, which forces the algorithm to prune the current branch. Steps 3 and Step 4 in the example shown in Table IV depict this situation.

Our implementation processes the bitvectors in a slightly different way than the original Eclat. For instance, in Step 2, our design evaluates (1 & 2) bitwise AND 3, while the traditional Eclat algorithm would have evaluated (1&2) bitwise AND (1&3) to determine the support of 1&2&3.

Table V summarizes the resource utilization of our design. Note that the memory and FIFO control during the recursive tree search is implemented in hardware as a finite state machine with a stack.

### 5.1. Memory Interface

Our platform is a GiDEL PROCStar III PCIe add-in card. It contains four Altera Stratix III 260 FPGAs. Each FPGA is connected to three independent off-chip DRAM memory banks, bank A, bank B, and bank C. Banks A and B have a peak bandwidth of 4 GB/s and bank C has a peak bandwidth of 2.1 GB/s.

GiDEL's design software can generate customized interfaces to these banks. These customized interfaces may be configured to have, for example, seven 32-bit internally arbitrated ports or three 256-bit internally arbitrated ports. This provides a tradeoff between having few wide ports or a larger number of narrow ports.

Near-maximum memory throughout can be achieved with any port width, but only when accessing consecutive words from the memory. The arbitration logic causes pre-emption when multiple processing elements access the ports concurrently, which results in interruptions in the stream of memory requests and degrades performance. As a result, we use a 256-bit port, the widest possible port size.

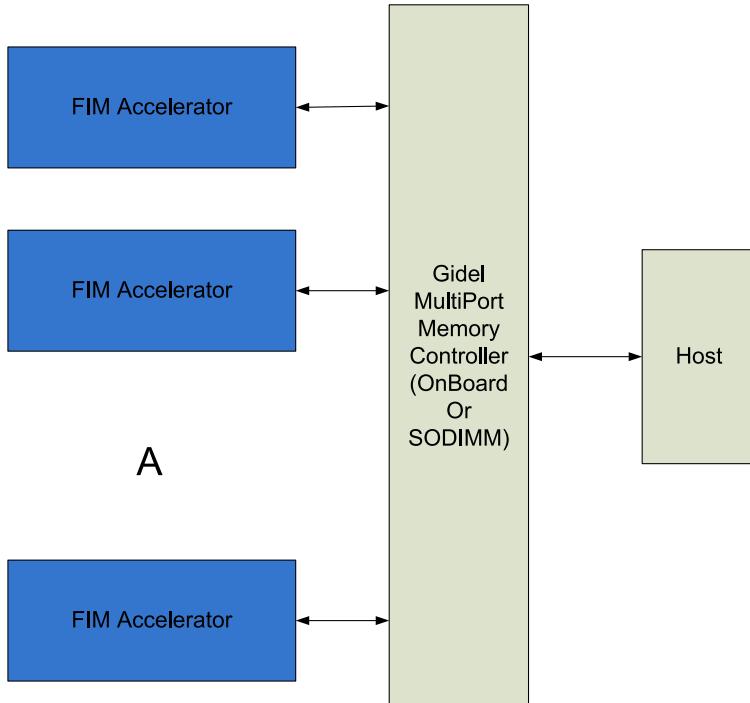


Fig. 3. Parallelizing multiple FIM Accelerators.

### 5.2. Parallelizing the Accelerator Architecture

Figure 3 shows our top-level architecture, showing how multiple processing elements and the host CPU are connected to GiDEL’s memory system. We instance two 256-bit accelerators for Bank A and B and a 128-bit accelerator for Bank C (an inherent limitation of the platform) on each of the four FPGAs.

As shown in Figure 4, in order to distribute the workload to the pool of PEs, we dynamically assign each of the topmost branches of the depth-first search to each processing element. In other words, each PE is assigned a branch with a different starting item. Since the running time of each branch varies, a new branch is assigned to each PE as it completes its workload. We repeat this process until all branches have been processed.

### 6. USE OF ON-CHIP SCRATCHPAD MEMORY

The performance of our accelerator is limited by memory bandwidth. Thus, for a given workload, a reduction in the required number of off-chip memory accesses will improve performance. In this section, we describe an optimization where intermediate results across branches are compressed and stored on-chip so that they do not need to be loaded from off-chip. These intermediate results are kept on chip to be used later to construct future candidates.

There are two primary types of data exchanges between a processing element and off-chip memory: *source transactions*, in which the original bitvectors from the database are read, and *intermediate results*, in which previously computed bitvectors corresponding to candidates of  $k$  items that had previously been stored to off-chip memory are read back to construct candidates of  $k + 1$  items.

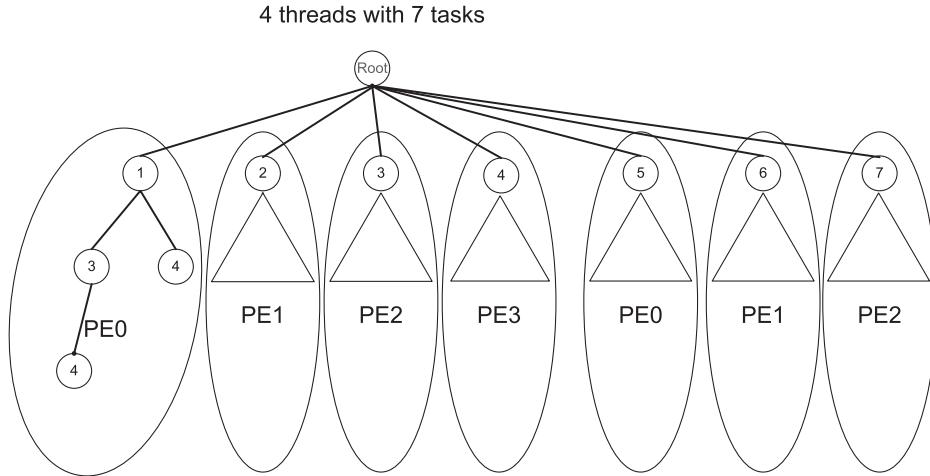


Fig. 4. Distributing tasks to accelerators.

Storing some of these intermediate results in a customized on-chip cache would reduce off-chip transactions. Further, an efficient lossless compression scheme for the intermediate results would allow more of them to be stored on-chip. In this section, we describe a method for compressing the intermediate results in a way that can be performed in hardware.

Our compression scheme exploits the fact that intermediate bitvectors, especially those farther from the root of the search tree, are sparse in sense that they have relatively few one-bits. To exploit this, we employ a compressed sparse storage format. We can further improve the performance of this compression technique by storing the intermediate bitvectors using differential encoding.

This approach has two characteristics that lend themselves to FPGA implementation. The first is how the compression and decompression can be integrated directly in the memory interface. On a CPU, these operations would interfere with the primary FIM computations, potentially causing cache pollution and additional off-chip memory accesses. Second, the design has complete control over the scratchpad memory, ensuring that the compressed intermediate values and the table that tracks them are always stored on chip. On a CPU-based implementation, there's no direct way to guarantee that specific arrays are kept on chip without unintended write backs to main memory. Thus, the potential benefits of implementing the same compression scheme in software would not offset the overheads it requires.

### 6.1. Differential Encoding

Assume two intermediate candidate bitvectors that are adjacent on the same branch of the search tree:

$$A = 1\&2, \quad (1)$$

$$B = 1\&2\&3. \quad (2)$$

Because they were generated consecutively they will differ only slightly, allowing us to replace one of these bitvectors with a difference vector:

$$C = A \text{ xor } B. \quad (3)$$

**Algorithm 2:** Bitvector Compression and Storage

---

**Input:** A — previous generation bitvector  
 B — Latest generation bitvector  
 $C_s$  — Capacity of scratch memory  
 $C_o$  — Occupied scratch memory size  
 $C_t$  — Compression Threshold for compressed bitvector  
**Output:**  $C_c$  — Compressed bitvector A

---

```

1:   C = A xor B;
2:    $C_c$  = Compression(C);
3:   if (sizeof( $C_c$ ) >  $C_t$ )
4:       Store A to off-chip and discard  $C_c$ 
5:   elseif (  $C_o + C_c > C_s$ )
6:       Store A to off-chip and discard  $C_c$ ;
7:   else
8:       Store  $C_c$  to on-chip and discard A;
9:       save the address of A to address table
10:  end if;
```

---

Storing C on-chip instead of A would increase the sparsity and improve compression. A can be reconstructed from C, using:

$$A = C \text{ xnor } \bar{B}. \quad (4)$$

Using Formulas (3) and (4), we can generate C and reconstruct A using little hardware logic. The on-chip memory will not introduce any delay as long as we pipeline the compression and decompression.

Note this example shows only two generations. In general, if only the last generation's result is stored in FIFO B in a nonencoded representation, all previous generation's intermediate results can be encoded relative to it.

## 6.2. Compression

The total on chip memory in our FPGA is limited, consisting of less than 2MB of total capacity, and generally, it is not even possible to use a large portion of this capacity for a single, monolithic memory (due to limitations in the routing fabric). Also, the proposed bitvector compression technique will perform poorly in the earlier generations (i.e., closer to the tree root), where there are more one-bits in the bitvectors.

Each bitvector will have a width equal to the number of transactions, which is typically thousands of bits or more. As a result, bitvectors are stored as a sequence of consecutive words (equal in width to the memory port) in the on-chip FIFOs. In our compressed sparse format, we store only the words that contain at least one one-bit. Each word is accompanied with an index that specifies where that word originally appeared in the original bitvector.

As shown in Algorithm 2, in order to maximize the number of intermediate vectors stored on chip, we only store vectors whose compression ratio, using our sparse format, exceeds a specified compression threshold. Note that this compression threshold is separate from the FIM threshold, which is still used as the pruning criteria in the FIM algorithm.

In line 1 the algorithm calculates the difference of bitvectors A and B and store them as C. In line 2, the algorithm discards the all zero words and annotates the bitvector with the word indices. In lines 3 and 5, the algorithm checks the compression ratio and the amount of remaining on-chip memory to fit the compressed bitvector. If the compression ratio is too high or if there is insufficient remaining memory capacity,

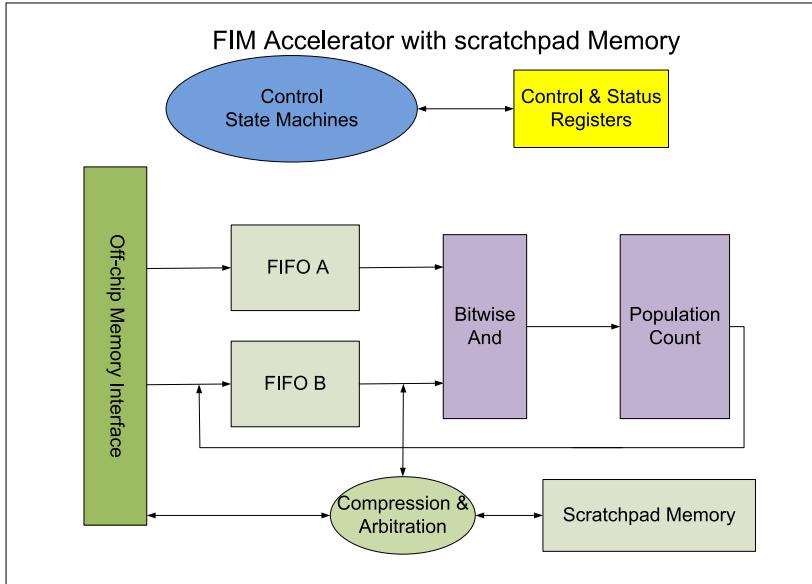


Fig. 5. The FIM accelerator architecure enhanced with scratchpad memory.

bitvector  $A$  will be sent off-chip in noncompressed format. Otherwise, bitvector  $A$  is discarded and the compressed data is stored in on-chip memory.

The decompression of  $C_c$  and reconstruction of  $A$  are similar so we don't show this procedure here.

After adding the on-chip memory, which we call *scratchpad memory*, and related logic, the architecture of our accelerator is shown in Figure 5. As shown, scratchpad memory, compression and arbitration logic are added between FIFO B and off-chip memory.

## 7. EVALUATION

In this section we describe the performance results of our processing element design without the scratchpad memory. In the next section we provide an analysis of performance improvement after adding the scratchpad memory optimization.

### 7.1. Hardware Platform and Software Environment

The FPGA card's block diagram is shown in Figure 6. As shown in Figure 6(a), each FPGA is connected to two 2 GB of SODIMM and one 256 MB DDR2 onboard DRAM. The DDR2 DRAM is bank A of FPGA, operating at 667MHz. The two SODIMMs are bank B and C. Bank B operates at 667MHz, while bank C operates at 360MHz.

As shown in Figure 6(b), each FPGAs is connected to its immediate neighbor on the left and right via the L/R bus (though we do not utilize this bus in our design). Each FPGA is also connected to a main bus, link bus, and local bus. The local bus not only connects to all devices, but also the PCIe bridge.

We run our comparative results of Borgelt's Eclat on a Dell PowerEdge R710 server. The server contains two Intel Nehalem Xeon 5520 CPUs. The CPU runs at 3.2GHz and the system memory is 16 GB. The software baseline is written in C++ and compiled by g++ 4.4 with optimization level 3 (-O3). The parallel version for 16-thread all runs at this server. The parallel is compiled using g++ 4.4 with OpenMP 4.0.

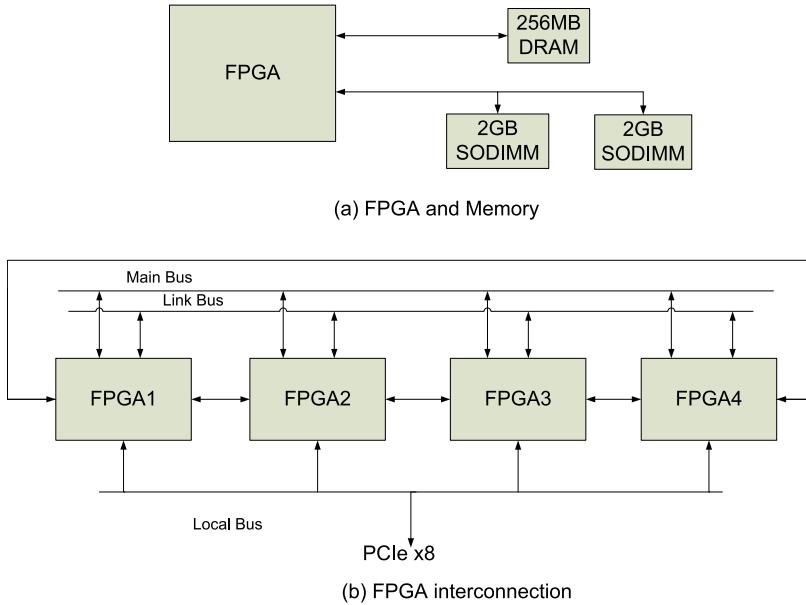


Fig. 6. Gidel board FPGA and memory framework.

Table VI. Test Datasets Characteristics

Dataset	Size(MB)	Average Length	Transaction Number	Number of Items
T40I10D0_3N500K	63	40	500K	299
T40I10D0_3N1000K	133.9	40	1000K	300
T60I20D0_5N500K	103.8	60	500K	500
T90I20D0_5N500K	155.3	90	500K	499

In our FPGA implementation, we attach an accelerator to each memory bank. Bank A and bank B are configured for a 256-bit word while bank C is limited to a 128-bit word. The design runs at 200MHz.

## 7.2. Test Data

Most of the FPGA FIM implementations in the literature use Bodon's Apriori implementation [Bodon and Ronyai 2003] as their software baseline. However, we compare to Borgelt's implementation [Borgelt 2003] since it is a state-of-the-art software Eclat implementation, is still under active development and being improved, and achieves substantially higher performance than Bodon's implementation.

Table VI summarizes our test datasets. These are synthetic datasets generated by IBM's Synthetic Data Generator [IBM 2012].

## 7.3. Performance Results

Due to limitations of the memory controller, memory Bank A and Bank B supports a word width of 256 bits while Bank C only supports a word width of 128 bits. Table VII shows our performance results for a single 128-bit and 256-bit processing element, as well as the performance of the entire coprocessor card (eight 256-bit processing elements and four 128-bit processing elements across four Stratix III EP3SE260 FPGAs) relative to Borgelt's Eclat implementation running on our Xeon 5520. Note that the 256-bit accelerator running at 200MHz has a capacity of 6.4GB/s (in terms of

Table VII. Accelerator Performance

Dataset	Threshold	Serial SW baseline (s)	16-threads OpenMP implementation(s)	256-bit PE		128-bit PE		Overall		
				Running time(s)	Speedup	Running Time(s)	Speedup	Running Time(s)	Speedup compared to 1-thread	Speedup Compared to 16-thread
T40I10D0_3N500K	0.01	819.5	87.7	166.41	4.92	287.82	2.85	21.33	38.41	4.12
T40I10D0_3N1000K	0.02	270.6	33.4	56.81	4.76	99.63	2.72	7.132	37.94	4.68
T60I20D0_5N500K	0.02	393.5	43.8	84.77	4.64	146.96	2.68	12.52	31.43	3.49
T90I20D0_5N500K	0.05	685.3	78.2	156.29	4.39	270.71	2.53	23.43	29.25	3.38

width x clock rate), but the maximum sustainable memory bandwidth offered by the DRAM is 3.2 GB/s. This effectively reduces our hardware utilization. While this is an opportunity to reduce our core clock rate, we opted to continue to target 200 MHz in order to keep the design fast enough to target platforms with higher off-chip memory bandwidth.

As shown, the 256-bit PE achieves a speedup of 4 to 5 compared with the baseline and the 128-bit PE achieves a speedup of 2–3. The entire system achieves a speedup of 30 to 40.

#### 7.4. Design Space Exploration

As described in Section 6, storing compressed intermediate bitvectors on chip can potentially improve accelerator performance by reducing the number of intermediate results that need to be read from off-chip memory. However, several important parameters will influence the effectiveness of this approach. These include the following.

- *Word size*. Our compression technique relies on eliminating any word that contains only zero bits. Using longer words increases the likelihood that there is a one-bit within the word, but will reduce the number of bits required to index the words, reducing the overhead of the metadata required by the compression scheme. Note that the word size used in the compression scheme does not necessarily need to match the word size of the memory port. In order to explore the effect of the compression word size, we decoupled the *memory word size* from the *compression word size*.
- *Scratchpad memory size*. Larger scratchpads will allow more intermediate values be kept on chip but might require longer access times. Instancing larger scratchpads will also make it challenging to meet timing requirements during place-and-route.
- *Compression threshold*. Higher compression threshold values will prevent denser, less compressible intermediate results from filling the on chip memory but will also prevent more frequently accessed intermediate results from lower in the tree from being stored.

In order to explore the effect of these parameters we will test our design under using combinations of these parameter values. In order to rapidly explore the design space we use a software model. Using the model, we estimate the performance improvement by comparing the number of memory accesses required with the scratchpad with the number required without the scratchpad. We report the ratio of these values as the potential speedup.

Table VIII. Scratchpad Memory Speedup for Dataset T40I10D0\_3N500K with FIM Threshold 0.01

Word Length (bit)	Scratchpad Capacity (Kb)	Compression Threshold									
		0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95
16	512	1.45	1.43	1.45	1.47	1.5	1.53	1.55	1.56	1.56	1.56
16	1024	1.73	1.78	1.82	1.85	1.86	1.87	1.87	1.83	1.77	1.56
16	2048	1.73	1.78	1.82	1.87	1.92	1.96	1.99	2.01	2.02	2.02
32	512	1.27	1.3	1.3	1.28	1.26	1.28	1.3	1.31	1.34	1.38
32	1024	1.27	1.35	1.44	1.55	1.66	1.74	1.77	1.74	1.72	1.75
32	2048	1.27	1.35	1.44	1.55	1.66	1.74	1.78	1.81	1.86	1.95
64	512	1.01	1.04	1.06	1.08	1.09	1.1	1.11	1.1	1.16	1.22
64	1024	1.01	1.04	1.08	1.13	1.2	1.27	1.36	1.47	1.61	1.65
64	2048	1.01	1.04	1.08	1.13	1.2	1.27	1.36	1.47	1.62	1.76
128	512	1	1	1	1	1	1.01	1.02	1.04	1.06	1.08
128	1024	1	1	1	1	1	1.01	1.04	1.09	1.17	1.32
128	2048	1	1	1	1	1	1.01	1.04	1.09	1.17	1.32

Table IX. Performance Summary

Dataset	Threshold	Serial Software Baseline(s)	16-thread(s)	Initial Memory Load time(s)	256-bit Speedup	128-bit Speedup	Scratchpad Speedup (1Mb)	Overall Runtime(s)	Overall Speedup	Speedup Compared to 16-thread
T40I10D0_3N500K	0.01	819.5	87.7	0.316	4.92	2.85	1.77	12.053	67.99	7.28
T40I10D0_3N1000K	0.02	270.6	33.4	0.584	4.76	2.72	1.25	5.705	47.43	5.85
T60I20D0_5N500K	0.02	393.5	43.8	0.476	4.64	2.68	1.43	8.754	44.95	5.00
T90I20D0_5N500K	0.05	685.3	78.2	0.436	4.39	2.53	1.41	16.614	41.25	4.71

In Table VIII, we show how scratchpad and compression parameters affect performance for dataset T40I10D0\_3N500K. In general, performance is best with smaller word sizes. Surprisingly, performance improves only slightly with a larger on-chip memory and achieves higher performance with higher compression threshold.

Our results are shown in Table IX. In this table, we also list cost of the initial memory load from host to on-board DDR memory (PCIe DMA performance). When using a fully populated design (12 PEs) with the scratchpad memory and compression and parameters determined in this section (1Mb size, word size 32, compression threshold selected based on optimal result), the final speedup compared to serial version ranges from 42 to 68. Using the OpenMP framework described in our previous work [Zhang et al. 2011], we can also compare our result with a 16-thread X86 implementation. The speedup against 16-thread parallel version ranges between 4.71 and 7.28.

## 8. CONCLUSION

In this article we present a novel hardware accelerator for frequent itemset mining. Unlike previous work, our design is compact and can process large datasets. The performance of our accelerator is not limited by the logic resource but by memory bandwidth. Our design is scalable and suitable for large scale data mining. We also present

a compression scheme to take advantage of the on-chip memory to reduce the total volume of data that need be exchanged with off-chip DRAM during runtime.

Due to the inherited limitation of the data structure bitvector, this architecture is not efficient for datasets with very low threshold (threshold <1%). This also leaves the possibility for further work to improve the design.

## REFERENCES

- Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*. 487–499.
- Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
- Alachiotis, N., Berger S. A., and Stamatakis, A. 2011. Accelerating phylogeny-aware short DNA read alignment with FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'11)*.
- Baker, Z. K. and Prasanna, V. K. 2005. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'05)*. 3–12.
- Baker, Z. K. and Prasanna, V. K. 2006. An Architecture for efficient hardware data mining using reconfigurable computing systems. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'06)*.
- Bodon, F. 2003. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*.
- Bodon, F. 2006. A survey on frequent itemset mining. Tech. rep., Budapest University of Technology and Economics.
- Bodon, F. and Ronyai, L. 2003. Trie: An alternative data structure for data mining algorithms. *Math. Comput. Model.* 38, 7–9, 739–751.
- Borgelt, C. 2003. Efficient implementations of Apriori and Eclat. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*.
- FIMI Repository. 2003. Frequent itemset mining dataset repository. <http://fimi.ua.ac.be/data>.
- Fukuzaki, M., Seki, M., Kashima, H., and Sese, J. 2010. Finding itemset-sharing patterns in a large itemset-associated graph. In *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. Vol. II, 147–159.
- Gidel Ltd. 2009. *PROStarIII Data Book*. Version 1.0.
- Goethals, B. 2002. Survey on frequent pattern mining. Tech. rep., Helsinki Institute for Information Technology.
- Goethals, B. and Zaki, M. J. 2003. Advances in frequent itemset mining implementations: Introduction to FIMI03. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*.
- Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, New York, NY, 1–12.
- Heiton, J. 2006. Designing signal processing systems for FPGAs. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'06)*.
- IBM. 2012. IBM synthetic data generator. <http://sourceforge.net/projects/ibmquestdatagen/>.
- Park, J. S., Chen, M. S., and Yu, P. S. 1997. Using a hash-based method with transaction trimming for mining association rules. *IEEE Trans. Knowl. Data Eng.* 9, 5, 813–825.
- Sun, S. and Zambreno, J. 2011. Design and analysis of a reconfigurable platform for frequent pattern mining. *IEEE Trans. Parallel Distrib. Syst.* 22, 9, 1497–1505.
- Sun, S., Steffen, M., and Zambreno, J. 2008. A reconfigurable platform for frequent pattern. In *Proceedings of the International Conference Reconfigurable Computing and FPGAs (ReConfig'08)*.
- Thoni, D. W. and Strey, A. 2009. Novel strategies for hardware acceleration of frequent itemset mining with the Apriori algorithm. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'09)*.
- Wen, Y. H., Huang, J. W., and Chen, M. S. 2008. Hardware-enhanced association rule mining with hashing and pipelining. *IEEE Trans. Knowl. Data Eng.* 20, 6.
- Witten, I. and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufman, 27.

- Zaki, M. J. 2000. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.* 12, 3.
- Zhang, Y., Zhang, F., and Bakos, J. 2011. Frequent Itemset mining on large-scale shared memory machines. In *Proceedings of the IEEE International Conference on Cluster Computing*. 585–589.
- Zhou, L. and Prasanna, V. K. 2008. Scalable hybrid designs for linear algebra on reconfigurable computing systems. *IEEE Trans. Comput.* 57, 12.

Received May 2012; revised September 2012; accepted December 2012