A Reconfigurable Platform for Frequent Pattern Mining

Song Sun Michael Steffen Joseph Zambreno
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
{sunsong, steffma, zambreno}@iastate.edu

Abstract

In this paper, a new hardware architecture for frequent pattern mining based on a systolic tree structure is proposed. The goal of this architecture is to mimic the internal memory layout of the original FP-growth algorithm while achieving a much higher throughput. We also describe an embedded platform implementation of this architecture along with detailed analysis of area requirements and performance results for different configurations. Our results show that with an appropriate selection of tree size, the reconfigurable platform can be several orders of magnitude faster than the FP-growth algorithm.

1 Introduction

Frequent pattern (or frequent itemset) mining is widely used in many data mining applications. The goal of frequent pattern mining is to determine which items in a transactional database commonly appear together. One popular approach to frequent pattern mining is the *Apriori* algorithm [1], in which the range of candidate frequent itemsets is narrowed by applying the principle that a superset of items must not be frequent unless any subset of it is frequent. In the *FP-growth* algorithm [8], all transactions in the database are stored as a tree using two scans.

The FP-growth algorithm was originally designed from a software developer's perspective and uses recursion to traverse the tree and mine patterns. Consequently, a direct hardware implementation is not feasible for improved performance. To utilize the advantages of the FP-growth algorithm, a reconfigurable systolic tree architecture for frequent pattern mining was introduced, and a prototype using a Field Programmable Gate Array (FPGA) platform was presented [10]. In this paper, we modify the original scheme introduced in [10] by eliminating the counting nodes, and provide a new count mode algorithm. The embedded system incorporating the FPGA logic is described and the performance of the system is evaluated with three benchmarks.

The remainder of this paper is organized as follows. Section 2 describes related work in the area of data mining acceleration. In Section 3 we briefly introduce the concept of our systolic tree-based architecture, and present a hardware/software platform. In Section 4 we discuss scaling optimization for handling large transactional databases and its adaptation to our scheme. A detailed area and performance study is performed in Section 5, with a comparative analysis of our approach with the original FP-growth software algorithm. Finally, we conclude the paper in Section 6 with a look towards future work.

2 Related Work

A parallel implementation of the Apriori algorithm on FPGAs was first done in [2]. Due to the processing time involved in reading the transactional database multiple times, the hardware implementation was only $4\times$ faster than current software implementation. The same authors further explored the parallelism in Apriori and developed a bitmapped Content Addressable Memory (CAM)-based architecture that provided $24\times$ performance gain over the software version [3]. [11] proposed a hash-based pipelined architecture for hardware enhanced frequent pattern mining. As the Apriori algorithm requires the entire database to be read for every item in the worst-case scenario, further speedup would be limited.

Conclusions from several independent evaluations indicate that FP-growth is currently the optimal association rules mining algorithm [6, 8]. While software solutions exist [9], we are unaware of any existing hardware implementations of FP-growth algorithms besides [10]. Our architecture shares the same advantage as the FP-growth algorithm which only requires two scans of the database.

Table 1. Sample Transactional Database

ID	$Items$	ID	$Items$
1	B,C,D	5	A,B,C
2	B,C	6	A,B,C
3	A,C,D	7	A,B,D
4	A,C,D		

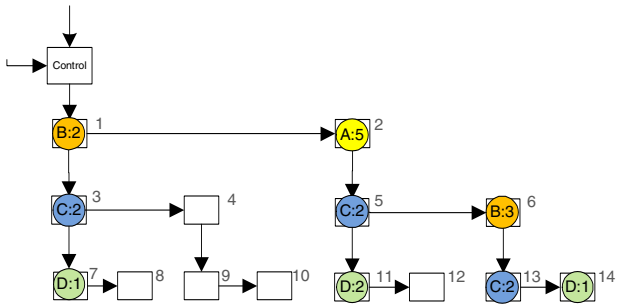

Figure 1. Systolic Tree Architecture

Algorithm: WRITE mode(item t)
$match := 0$; $InPath := 1$;
(1)**if** PE *is empty* **then**
    $store\ the\ item\ t$;
    $count := 1$;
    $match := 1$;
    $stop\ forwarding$;
(2)**if** (t *is in* PE) **and** ($InPath = 1$) **then**
    $match := 1$;
    $count++$;
    $stop\ forwarding$;
(3)**if** ($match = 0$) **then**
    $forward\ t\ to\ the\ sibling$;
    $InPath := 0$;
  else
    $forward\ t\ to\ the\ children$

Figure 2. WRITE Mode Algorithm

3 Pattern Mining using Systolic Trees

3.1 Overview

In VLSI terminology, a *systolic tree* is an arrangement of pipelined processing elements in a multi-dimensional tree pattern [4]. Implementation of a systolic tree is then application dependent for tree structure and processing element implementation. The structure of the systolic tree in frequent pattern mining introduced in [10] has a depth of W and K children for each node. Each node in the original FP-tree corresponds to a Processing Element (PE), resulting in the total number of processing elements in a tree as $K^W + K^{W-1} + ... + K + 1 = \frac{K^{W+1}-1}{K-1}$.

The systolic tree built from the database in Table 1 is shown in Fig. 1 where K=2 and W=3. The architecture has three main modes of operation: *write*, *scan*, and *count*. The systolic tree is built in *write* mode using the algorithm in Fig. 2. Input items are streamed from the root node in the direction set by the write mode algorithm. The root node is the control PE and acts as the interface to the rest of the tree. The support count of a candidate itemset is extracted in both *scan* and *count* mode. We refer to this process as candidate itemset *dictation*.

3.2 Systolic Tree Creation

When the tree is being built, all PEs are in *write* mode. An item is loaded into the control PE each cycle which in turn transfers each item to the general PEs. If the item is already contained in a node, the corresponding *count* value will be increased. Otherwise, an appropriate empty PE will be located. The algorithm for *write* mode in each PE is given in Fig. 2. The input of the algorithm is an item t and the $match$ flag is set when the item in PE matches t. The $Inpath$ flag is cleared when the PE does not contain any item of the current transaction. For example, PE 1 under the control PE in Fig. 1 does not contain the item B in the transaction {A,B,C} because the previous write of item A in the transaction cleared the $Inpath$ flag for PE 1. The item B is then stored in PE 6 and followed a path of PE 1,2,5,6. After all items are sent to the systolic tree, a control signal that declares the end of a transaction and the start of a new one is sent to the control PE. The signal is then broadcasted to all PEs which reinitialize $match$ and $Inpath$ flags for the next transaction.

3.3 Candidate Itemset Dictation

The FP growth algorithm generates frequent itemsets by recursive enumeration. However, a recursive implementation is impractical in an FPGA implementation. The approach used in systolic tree architecture is what we call candidate itemset dictation. When we want to check whether a given itemset is frequent or not, it is sent to the systolic tree and the count of an itemset will be returned by the root node of the systolic tree. This operation must be performed after the systolic tree is built and PEs are in *scan* mode.

In our systolic tree structure there is only one path tracing back from any PE to the control PE since each PE has a unique parent. The main principle of dictation is that any path containing the requested candidate itemset will be reported to the control PE. Note that the path may contain more items than the queried itemset. To clarify the dictation algorithm, we deem there are two doors in each PE to

Algorithm: SCAN mode(item t)
open the bottom door;
match := 0; $IsLeaf$:= 0;
(1)**if** PE *is empty* **then**
    stop forwarding;
(2)**if** (t *is in* PE) **and** (*Bottom door is open*) **then**
    match := 1;
    $IsLeaf$:= 1;
    $forward$ *t to the sibling*;
(3)**if** $t <$ *the item in* PE **then**
    $IsLeaf$:= 0;
    close the bottom door;
    $forward$ *t to the sibling*;
(4)**if** $t >$ *the item in* PE **then**
    $IsLeaf$:= 0;
    $forward$ *t to the sibling*;
    $forward$ *t to the child if the bottom door is open*;

Figure 3. SCAN Mode Algorithm

control the flow of data between neighboring PEs. The right door is always open and the bottom door is locked when no data should be sent to the children. The door policy is described in Fig. 3.

The $IsLeaf$ flag is set if the PE matches the last item in the queried candidate itemset. A PE with $IsLeaf$ set is responsible for reporting the number of the candidate itemset to its parent PE. Since the item is sent one by one, the flag $IsLeaf$ is cleared if another item in the candidate itemset which is larger than the stored item passes through the PE. If the input item is smaller than the stored item, the bottom door should be closed. The rationale behind this is that the item in the child can never be larger than that of its ancestor in a path. Therefore, when a bottom door in a PE is closed, the path passing through it will never contain the candidate itemset. If the input item is larger than the stored item, it should be forwarded to all open doors because the path may contain items which are not in the candidate itemset and the candidate itemset is contained in the path.

3.4 Candidate Itemset Count Computation

Once all items in a candidate itemset are sent to the systolic tree, a control signal is broadcasted to the whole systolic tree indicating a change to *count* mode. In *count* mode the PEs report the support count of the candidate itemset to its unique parent. Figure 1 shows the first child's input interface is always connected to its parent while others accept input from the sibling. The PE which is not directly connected to its parent sends its count to the left sibling. The parent PE collects the support counts reported by the children PEs and

Algorithm: COUNT mode(int $CountRight$, int $CountBottom$)
$CountSent$:= 0;
$CountChild$:= $CountRight + CountBottom$;
if ($CountSent$ = 0) **and** ($IsLeaf$ = 1) **then**
    $CountSent$:= 1;
    $forward$ ($CountMyself + CountChild$)
        to its parent direction;
else
    $forward$ ($CountChild$) *to its parent direction*;

Figure 4. COUNT Mode Algorithm

sends them to its own parent. The *count* mode algorithm in each PE is given in Fig. 4, where the support counts are transferred to each PE's parent in a pipelined fashion. The inputs of the algorithm are two count values sent by its sibling and child respectively. The $CountMyself$ variable is the count for the item at the node. The $CountSent$ flag is set when the local number has been reported to the parent PE. Only the PEs with the $IsLeaf$ flag set can report its local $CountMyself$ value while other PEs transfer the count values sent by the child and sibling. The control PE adds up all count values and sends it to the output signal.

Since each node has K degrees and the depth of the tree is W, the *count* mode signal reaches the right-bottom node after $(K-1) \times W$ cycles. It takes another $(K-1) \times W$ cycles for the count values to be broadcasted from the right-bottom node to the control node.

3.5 Initial Implementation

In the initial VHDL implementation, different sizes of trees are generated automatically by specifying the depth and degree parameters. Xilinx ISE 9.1.03i was used for synthesizing various configurations of our architecture for a Xilinx Virtex-4 XC4VFX140 with package FF1517 and -10 speed grade. The maximum delay for different combinations of K and W is shown in Fig. 5. The clock frequency drops dramatically when the node degree or the tree depth increases due to the increased number of interfaces and interconnections between PEs.

3.6 Platform Architecture

Similar to the FP-growth algorithm, two scans of the transactional database are required. In the first scan both the set of frequent items and the support count of each frequent item is collected. This task is implemented in the software component of our system as shown in Fig. 6. To mine the frequent itemset, each candidate frequent itemset is dictated

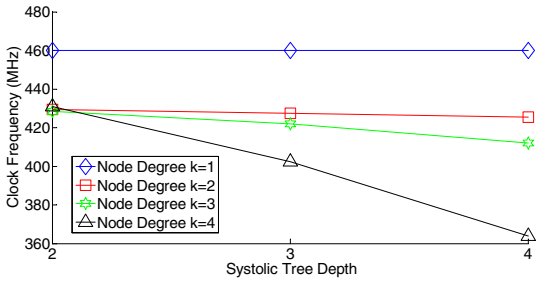

Figure 5. Systolic Tree Clock Frequency

by the software module into the systolic tree. The software module and systolic tree communicates through a Processor Local Bus (PLB). The systolic tree then reports the support count of the itemset back to the software module. The process of producing candidate itemsets is critical to the runtime of the whole system. When selecting candidate itemsets to dictate, we discard those itemsets which are not frequent. One solution is to scan the systolic tree once and record infrequent itemsets with a support count less than a threshold. However the time to traverse the systolic tree and space required to store the infrequent itemsets may be unacceptable, especially when the database is very large and sparse. Consequently, a brute-force method is used to dictate every candidate itemset. For example, there will be 2^n candidate itemsets if the number of frequent items is n.

The software controller also interfaces with the host workstation and is responsible for transforming the transactional database onto the systolic tree. The FPGA-based hardware component of the embedded platform is responsible for building the systolic tree while receiving the transactions sent by the software module, and extracting the support count of the candidate itemsets dictated by the software module.

4 Tree Scaling by Database Projection

Both the FP-growth and our systolic tree approach use the tree structure to compress the representation of transactions in the database. The size of the tree is dependent on the characteristics of the database where a dense tree leads to a smaller tree size. In the case of insufficient memory for storing the entire tree, the database must be divided into a multiple of smaller databases with fewer frequent items. In [10], the authors show that the largest commercially-available Virtex-5 FPGA can accommodate arbitrary frequent itemsets of size four or less. Without the technique of *database projection*, a transactional database with thousands of items cannot be mined in a reasonable space. Also, the brute-force candidate itemset dictation will take an intolerable amount of time when the number of frequent items

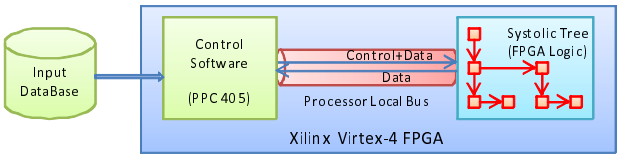

Figure 6. Platform System Architecture

is large. Therefore it is unreasonable to assume that a tree-based representation can fit in the available hardware resources (either memory or logic) for any arbitrary database.

Two methods for database projection are introduced in [8] for FP-trees: *parallel projection* and *partition projection*. There is a time-space trade off between parallel projection and partition projection. Partition projection typically uses less memory and I/O requests but requires each projected database to be mined sequentially. On the other hand, parallel projected databases can be processed in parallel reducing the run time, but at the cost of increased memory consumption.

5 Evaluation

In this section, we study the performance of the systolic tree-based frequent pattern mining algorithm combined with both the parallel projection and partition projection techniques. We also compare our simulation results to an FP-growth based software solution.

5.1 Area and Performance Requirements

From Section 3 there are $\frac{K^{W+1}-1}{K-1} = O(K^W)$ nodes in a systolic tree. The depth of the tree has exponential influence on the total size. For example, there will be around 10B nodes if W and K are both 10. Based on the synthesis results from our experiments with ISE 9.1.03i targeting a Xilinx Virtex-4 FPGA, each node requires 50 LUTs. Therefore the FPGA chip should at least contain 500B LUTs to accommodate this tree. Approximately 25% of the available slices will be used when K and W are both four. The value of K and W are highly related to the number of frequent items n. To record all transactions in the database, K and W are usually no less than n. However, if the support count threshold is very large, not all transactions are needed in the tree. The support count of each candidate itemset extracted from the systolic tree is approximately equal to the true support count.

As shown in Section 3.5, the clock frequency drops sharply with an increase of W and K. However W and K cannot be greater than 4 due to the space limitation of the Virtex-4 FPGA. During the build phase of the systolic tree, one item is transferred into the tree every cycle (at the

360 MHz minimum clock frequency as reported in Fig. 5). If the size of the database after pruning infrequent items in each transaction is B, it will take $\frac{B}{throughput}$ seconds to build the systolic tree on the FPGA where $throughput = clock\ frequency \times itembandwidth$.

The time to return the support count of a candidate itemset with C items is composed of three segments. The first segment is the time to deliver the items in the candidate set. Since the candidate items can be sent to the systolic tree in a heavily pipelined fashion, the time for this part is approximately C clock cycles. The second segment is the time for an item propagated from the control node to the farthest node. Since each node has K degrees and the depth of the tree is W, the time for the second segment is $(K-1) \times W$ cycles. The final segment is the time to collect the support count from the nodes where the last item in the candidate itemset resides. As discussed in subsection 3.4, the time for this step is $2 \times (K-1) \times W$ cycles. The first two segments correspond to the time used by the *scan* mode. The last segment corresponds to the time used by the *count* mode. In summary, the number of cycles required to return the support count of a candidate itemset with C items is $C+(K-1)W+2(K-1)W = C+3KW-3W = O(KW)$ cycles.

For a database containing n frequent items, there is a total of 2^n candidate itemsets to be dictated. The number of cycles required to extract all frequent itemsets is around $2^n(C + 3KW - 3W)$. Usually K and W are approximately equal to n, and the longest candidate itemset contains n items. The total time used to extract the frequent itemset from a database with n frequent items is at most $2^n(n + 3n^2 - 3n) = 2^n(3n^2 - 2n)$ clock cycles.

5.2 Partition Method Analysis

A systolic tree with K node degrees and W depth cannot have the number of frequent items in every transaction be larger then $min(K, W)$. For a transactional database with n frequent items, we have the following observations.

Observation 1. *The number of sub-databases with no more than $N = min(K, W)$ frequent items is $2^{n-N} + N - 2$ where $n > N$.*

Proof. Suppose n frequent items are ordered in descending frequency order as $i_1, i_2, i_3, \cdots, i_n$. The items in each transaction are also reordered in descending order. Those transactions ending with item i_1 can be put into one sub-database. This database contains only item i_1. Because we have the support count of each item, we can ignore this sub-database. The sub-databases ending with the first $N+1$ frequent items contain no more than N frequent items in each transaction. These sub-databases are projected by $\{i_2\}, \{i_3\}, ..., \{i_{N+1}\}$.

The number of sub-databases ending with i_{N+2} is 2. One sub-database is composed of those transactions ending with $\{i_{N+1}i_{N+2}\}$. Other transactions ending with i_{N+2} constitute another sub-database. The number of sub-databases ending with i_j is $2^{j-(N+1)}$. Any projected itemset includes i_j and a subset of items $A = \{i_{N+1}, \cdots, i_{j-1}\}$. The number of subsets of A is $2^{(j-1)-(N+1)+1} = 2^{j-(N+1)}$. The total number of sub-databases with no more than $N = min(K, W)$ frequent items is calculated by:

$$N + \sum_{j=N+2}^{j=n} 2^{j-(N+1)} = N + \sum_{k=1}^{k=n-(N+1)} 2^k = N + 2^{n-N} - 2$$

Observation 2. *The number of clock cycles used for mining frequent itemsets are $2^N(3N^2 - 2N)$ in parallel projection since all sub-databases can be mined simultaneously, where $N = min(K, W)$.*

Observation 3. *The number of clock cycles used for mining frequent itemsets is equal to $2^N(3N^2-2N) \times (2^{n-N}+N-2)$ in partition projection, where n is the number of frequent items and $N = min(K, W)$.*

Proof. The sub-databases in partition projection are mined sequentially. Without considering the time used to project the transaction into other sub-databases, the number of clock cycles used to extract all frequent itemsets is computed by:

$$(2^N(3N^2 - 2N)) \times (2^{n-N} + N - 2)$$

5.3 Comparison with FP-growth

Usually the size of the tree is proportional to the number of frequent items. The number of nodes in an FP-tree is at most 2^n, where n is the number of frequent items. The number of nodes used in a systolic tree is $\frac{K^{W+1}-1}{K-1}$. If we let K and W be equal to n, the nodes used in our systolic tree is $(\frac{n}{2})^n$ times larger than that of FP-tree in the worst case scenario. The nodes in a systolic tree contain data but also operation code making the size larger than the nodes in FP-tree.

While the mining time using a systolic tree is only determined by the number of frequent items, different FP-trees with the same number of frequent items have different mining time. We used a Java implementation of the FP-growth algorithm from [5]. The running time of the software algorithm is collected from a PC with a Pentium D 3GHz CPU and 2GB RAM. The three benchmark datasets are from [7]. The mining time of the FP-growth algorithm is closely related to the number of nodes in the FP-tree and not from the number of frequent items.

To fully appreciate the power of systolic tree architecture, we compare the mining time of the systolic tree and

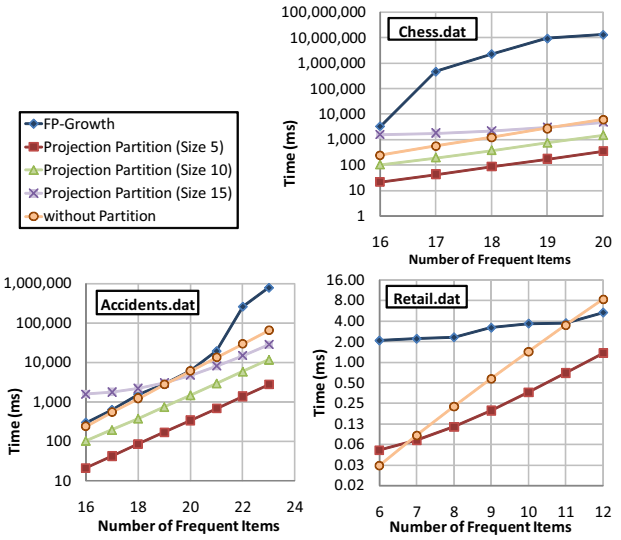

Figure 7. Mining Time of Benchmarks

FP-tree for several benchmarks shown in Fig. 7. The mining time of the partition projection approach is taken as the mining time of systolic tree since the mining time of parallel projection is heavily determined by the selected tree size. The maximum number of frequent items is no more than 25 due to the limitations of fitting a larger FP-tree into 2GB of memory. For the database Chess.dat, the mining time of the systolic tree is always faster then the FP-growth without any need to partition the database. The mining time of FP-growth algorithm is highly dependent on the structure of the FP-tree. The time to mine different FP-trees with the same number of frequent items may vary greatly, while the mining time of the systolic tree is solely determined by the number of frequent items. The higher performance of 2230× for chess.dat at 20 frequent items is because the tree for FP-growth is not as compact compared to Accidents.dat and Retail.dat resulting in longer running time for the FP-growth algorithm. Systolic tree also outperformed FP-Growth for the Accidents.dat benchmark without using any partitioning and has a speedup of 12× at 23 frequent items. FP-growth can implement the Retail.dat benchmark in a compact tree allowing for efficient runtime but cannot outperform the systolic tree for frequent items less then 12.

6 Conclusion and Future Work

We have proposed a reconfigurable architecture which uses a systolic tree to mine frequent patterns. This architecture implements small processing elements that map well to an FPGA-based embedded system. Both the hardware space requirement and the frequent pattern mining time are entirely dependent on the size of the systolic tree. With carefully selected tree size, the mining time of systolic tree can be orders of magnitude faster than the FP-tree. Due to the structural characteristic of our proposed systolic tree architecture, the tree size can not be very large. Our future work is to consider how to shrink the size of the tree at an architectural level. In addition *scan* and *count* mode currently only support one dictation at a time. Future work will investigate supporting multiple dictations at a time with additional pipelining to further increase throughput.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Managament of Data*, pages 207–216, May 1993.

[2] Z. Baker and V. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proceedings of the IEEE symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–12, April 2005.

[3] Z. Baker and V. Prasanna. An architecture for efficient hardware data mining using reconfigurable computing systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 67–75, April 2006.

[4] J. Bentley and H. Kung. A tree machine for searching problems. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 265–266, 1979.

[5] F. Coenen. The LUCS-KDD implementation of the FP-growth algorithm. available at http://www.csc.liv.ac.uk, February 2003.

[6] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.

[7] B. Goethals. Frequent itemset mining dataset repository. available at http://fimi.cs.helsinki/fi/data, 2008.

[8] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, January 2004.

[9] I. Pramudiono and M. Kitsuregawa. Parallel FP-growth on PC cluster. In *Proceedings of the Pacifica-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2003.

[10] S. Sun and J. Zambreno. Mining association rules with systolic trees. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 143–148, Sept. 2008.

[11] Y.-H. Wen, J.-W. Huang, and M.-S. Chen. Hardware-enhanced association rule mining with hashing and pipelining. *IEEE Transactions on Knowledge and Data Engineering*, 20(6):784–795, 2008.