Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs[1]

Zachary K. Baker and Viktor K. Prasanna
University of Southern California, Los Angeles, CA, USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu

Abstract

The Apriori algorithm is a popular correlation-based data-mining kernel. However, it is a computationally expensive algorithm and the running times can stretch up to days for large databases, as database sizes can extend to Gigabytes.

Through the use of a new extension to the systolic array architecture, time required for processing can be significantly reduced. Our array architecture implementation on a Xilinx Virtex-II Pro 100 provides a performance improvement that can be orders of magnitude faster than the state-of-the-art software implementations. The system is easily scalable and introduces an efficient "systolic injection" method for intelligently reporting unpredictably generated mid-array results to a controller without any chance of collision or excessive stalling.

1 Introduction

Recent advances in storage and data sensing have revolutionized our technological capability for collecting and storing data. Server logs for popular websites, customer transaction data from network routers, credit card purchases, customer loyalty cards, etc. produce terabytes of data in the span of a day. While it is useful as a historical record, effective processing for patterns and trends can make it profitable. Correlation-based data mining is the field of algorithms to process this data into more useful forms, in particular, connections between sets of items. In this paper, we investigate the Apriori algorithm [2], a popular strategy designed for progressively grouping together frequent itemsets in large databases given a particular frequency of occurrence cutoff.

While the computation and data complexity of the Apriori algorithm is very high, little research has been done in efficient implementations for hardware acceleration. We feel that much of the disinterest in addressing these problems lies in the challenge of implementing set membership functions efficiently, as well as in the complexity of control. We address these issues through hybrid systolic array-microcontrolled datapaths and efficient design principles. This paper presents several strategies we have developed for adapting the Apriori algorithm to use in a systolic array [10]. We also present a strategy called "systolic injection," a contribution to the general use of systolic arrays that allows a wide range of previously challenging applications to be implemented efficiently. Through the use of the systolic array we allow for increased frequency performance, decreased interconnect, and simple, easily scalable units. We implement the parallelizable and computation intensive operations within the systolic array and implement the serial and control intensive operations within a microcontroller. The microcontroller also controls data source and sink for the array. Moving the control-intensive operations to the microcontroller, or "software deceleration" [9], allows the operations that take up a small fraction of the total running time to be executed with less hardware than it would require as a complex state machine.

Due to our streaming implementation, the candidate generation phase of the algorithm require orders of magnitude less time than the support calculation. Overall, the hardware approach provides a minimum of a 4x time performance advantage over the fastest non-supercomputer implementations [4, 5, 7], and often provides much higher performance multipliers. The off-chip memory required is moderate beyond the size of the database and the bandwidth between memory and the systolic array is 250MB/s.

While the architecture could easily be implemented in a custom ASIC – in fact, the simple units that make up the systolic array are designed explicitly for ease of ASIC implementation – the use of FPGA allows the user to utilize parameterized designs which allow for variable size item descriptors as well as optimized memory sizes for a particular problem. As well, FPGAs allow the design to be scaled upward easily as process technology allows for ever-larger gate counts.

[1]Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the Xilinx and HP Corporations.

2 Related Work

As far as we know, the Apriori algorithm has not been studied in any significant way for efficient hardware implementation. However, research in hardware implementations of related data mining algorithms has been done [6, 17, 18].

In [6] and [17] the k-means clustering algorithm is implemented as an example of a special reconfigurable fabric in the form of a cellular array connected to a host processor. K-means clustering is a data mining strategy that groups together elements based on a distance measure. The distance can be an actual measure of Euclidean distance or can be mapped from some other data type. Each item in a set is randomly assigned to a cluster, and the centers of the clusters are computed. The elements are then iteratively added and removed from clusters to move them closer to the centers of the clusters. This is related to the Apriori algorithm as both are dependent on efficient set additions and computations performed on all elements of those sets. However, k-means adds the distance computation and significantly changes how the sets are built up.

In [18] a system is implemented which attempts to mediate the high cost of data transfers for large data sets. Common databases can easily extend beyond the capacity of the physical memory, and slow tertiary storage, e.g., hard drives, are brought into the datapath. The paper proposes the integration of a simple computational structure for data mining onto the hard drive controller itself. The data mining proposed by the paper is not Apriori, but rather the problem of exact and inexact string matching, a much more computationally regular problem compared to the Apriori algorithm. However, the work is useful, and will become more so as FPGA performance scales up and significantly exceeds the data supply capabilities of hierarchical memory systems.

We base our comparisons of hardware performance versus various state-of-the-art software implementation [4, 7, 5] as we are unaware of any comparable hardware implementation of the Apriori algorithm. Extensive research exists [8, 12] on parallelizing correlation algorithms, but we focus on single-processor machine performance.

3 Introduction to the Apriori Algorithm

We divide the Apriori [2] algorithm into three sections, as illustrated in Figure 1. Initial frequent itemsets are fed into the system, and candidate generation, candidate pruning, and candidate support calculation is executed in turn. The support information is fed back into the candidate generator and the cycle continues until the final candidate set is determined. In our hardware implementation, the same FPGA configuration is used throughout the various modes of operation, with only small changes in the control of the units. We will first introduce some of the data mining lexicon and then describe the operational phases in more detail.

In the literature, an analogy to a shopping cart is used. A *basket* is the set of items purchased at one time, checked out from the library, or otherwise grouped together based on some criteria such as time, customer, etc. A *frequent item* is an item that often occurs in a database. A *frequent itemset*, then, is a set of items that often occurs together in the same basket within the database if it consists of more than one item. The cutoff of how often a set must occur before it is included in the candidate set is the *support*.

In this way, a researcher can request a particular support value and find the items which occur together in a basket a minimum number of times within the database. This guarantees a minimum confidence in the results. A popular example in the literature (possibly apocryphal) is processing the supermarket transactions of working men with young children: when they go to the store after work to pick up diapers, they tend to purchase beer at the same time. Thus, it makes sense statistically to put a beer refrigerator in the diaper aisle.

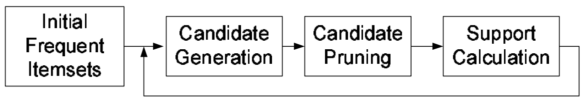

Figure 1: Process flow of the data mining system

Candidate generation is the process in which one generation of candidates is built into the next generation. This building process is from where the *Apriori* name derives. Each new candidate is built from candidates that have been determined *apriori* (in the previous generation) to have a high level of support. Thus, they can be confidently expanded into new potential frequent itemsets. This is expressed formally as follows:

```
$\forall c_1, c_2 \in C_m$ do
  with $c_1 = (i_1, ..., i_{m-1}, i_m)$
  and $c_2 = (i_1, ..., i_{m-1}, i^*_m)$
  and $i_m < i^*_m$
      $c := c_1 \cup c_2 = (i_1, ..., i_{m-1}, i_m, i^*_m)$
```

It should be noted that only ordered sets are utilized, that is, the item codes increase toward the last item in a set. Thus, when c is generated from c_1 and c_2, the sets remain ordered. Candidate generation pairs up any candidates that differ only in their final element to generate the candidate itemsets for the next candidate generation.

The next step of candidate generation guarantees that each new candidate is not only formed from two candidates from the previous generation, but that all subsets that can be created by removing one element are also present in the previous generation, as follows:

```
$\forall c \in C_m$ do
        $\forall i \in c : c - \{i\} \in C_{m-1}$
```

The initial candidate generation proves by design that if we remove either of the last two items (i_m, i^*_m) from the new candidate, we will get candidates from the previous generation, namely, c_1 and c_2. The second step verifies that if we remove any single item from the new candidate, we will find a candidate from the previous generation. This progressive build-up of candidates is the heart of the Apriori algorithm.

The third phase of the algorithm is the support calculation. It is by far the most time consuming and data intensive part of the application, as during this phase the database is streamed into the system. Each potential candidate's support, or the number of occurrences over the database set, is determined by comparing each candidate with each transaction in the database. If the set of items that form the candidate appear in the transaction, the support count for that candidate is incremented, as follows:

```
$\forall t \in T$ do
    $\forall c \in C$ do
        if $c \subset t$
            support(c)++
```

The main problem with the Apriori algorithm is this data complexity. Each candidate must be compared against every transaction set. This gives a large running time for a single generation, $O(|T||C||t|)$, assuming the subset function can be implemented in time $|t|$. However, the parallelism contained in the loops allows for some interesting acceleration in hardware, particularly when implemented as a systolic array.

4 Our Approach

Our general approach is to implement the Apriori algorithm in the most efficient manner possible, utilizing a minimum of hardware and a minimum of time, as well as insuring that utilization of the hardware comparators is near 100%. For some parts of the implementation, namely the support calculation, this is an easy task as checking for set equivalence is a simple operation. However, the candidate generation and pruning operations are significantly more complicated as they introduce new data in the system at unpredictable intervals. We also must ensure that our memories are of minimum size and yet also be sufficient to store necessary data.

Our architecture allows for 560 units on a single device. These results are based on the place-and-route of the full systolic array design on a Xilinx Virtex-II Pro 100 device with 44,000 slices. Hardware usage is 70 slices per systolic unit with resources for up to 16 2-byte item candidate sets. The units are all connected end-to-end in the form of a linear array. Each unit contains memory locations to temporarily store the candidates whose support is being calculated and to allow for stalling. A unit is composed of the candidate memory, an index counter, and a comparator, which allows the output of the candidate memory to be compared with an incoming item. Because all sets utilized in the system are ordered, the equivalence and greater than signals are all that are required to determine set equivalence and subset functions. These pieces are handled by the controller to implement the required functionality in the three computational phases.

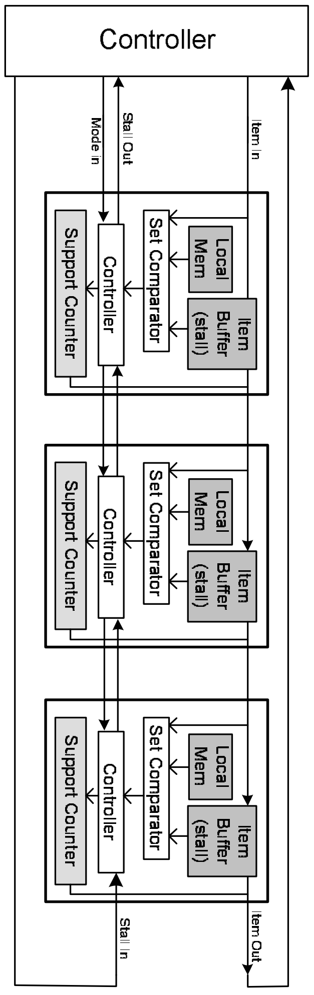

Figure 2: Illustration of the general data mining systolic array

Throughout this section, we will use the following variables:
i, the index of a systolic array unit
m, the number of items in a candidate list, also the generation number as one item is added to the list per generation
$|C_m|$, the number of total candidates in generation m,
c_a, the number of hardware units (c_a may be less than $|C_m|$),
$|T|$, the number of individual baskets in the database
$t_t = \sum_{k=1}^{|T|} |T_k|$, the total number of items in the database,

The elements are connected end-to-end in a one-dimensional array. Data flows in one direction, stall information flows is the opposite direction. We will express the array connections and behavioral descriptions in Standard Temporal Arithmetic (STA) [11]. The STA descriptions will illustrate only the first pass (as if $|C_m| \leq c_a$) for simplicity. All subsequent passes (in the support calculation, and candidate generation phases) are behaviorally identical. The connections between cells B_1 through B_n are as follows:

$$\begin{aligned}
&\forall i.\ 1 \leq i < c_a \\
&\qquad \text{conn}(\text{out}(O_{data}, B_i), \text{In}(I_{data}, B_{i+1})) \\
&\forall i.\ 2 \leq i < c_a + 1 \\
&\qquad \text{conn}(\text{out}(O_{stall}, B_i), \text{In}(I_{stall}, B_{i-1}))
\end{aligned}$$

4.1 Support Calculation

The support calculation is the simplest of the operations in the Apriori algorithm to implement. Its use of two loops with no dependencies allows for high parallelization. The challenge lies in getting the data to the units.

The first step is to load the units with candidates. Data enters at one end of the linear array. After the first candidate is stored in the first unit, the ith candidate set is forwarded along to the ith unit, until all units in the array are full. This requires $m\ |C_m|$ cycles. The time, however, may be split into multiple sections if $|C_m| > c_a$.

The support operation is as follows[2]

$$\begin{aligned}
&\forall t.\ 0 < t \leq m|C_m| \\
&\qquad \text{Val}(\text{In}(I_{data}, B_1)) = \bigwedge_{c=1}^{|C|} \bigwedge_{e=1}^{m} \text{external mem}(c, e) \\
&\qquad \text{if}(\textit{candidate\_num} = \textit{unit\_id}) \text{ then } \text{mem}_i(e) = I_{data} \\
&\qquad \text{Out}(O_{data}, B_i) = \text{In}(I_{data}, B_i)
\end{aligned}$$

$$\begin{aligned}
&\forall t.\ m|C_m| < t - i \leq \sum_{k=1}^{|T|} |T_k| \\
&\qquad \text{Val}(\text{In}(I_{data}, B_1)) = \bigwedge_{k=1}^{T} \bigwedge_{j=1}^{|T_k|} T_k(j) \\
&\qquad \text{if}(I_{data} \neq \text{mem}_i(m_i)) \text{ then } \textit{match} = fail \\
&\qquad \text{else} m_i + + \\
&\forall t.\ mc < t - i \leq \sum_{k=1}^{|T|} |T_k| \text{ and } (t \bmod m) = 0 \\
&\qquad (\textit{support}_i = \textit{support}_i + ((\textit{match}_i\text{= fail})?1{:}0)) \\
&\qquad \text{reset(match)}_i; m_i = 0
\end{aligned}$$

Next, the transactions are streamed through the array. All transaction are sent through, one element per cycle. Each transaction is an ordered set (like the candidate sets), so finding if the candidate is a subset of the transaction basket is similar to a merge sort. As each item arrives, it is compared with the current item. If the items match, the candidate pointer is incremented. If the item in the candidate memory is greater than the incoming item, the counter is not incremented. In this way, a very large transaction basket can be streamed through, and, if the counter pointer equals m by the end of the transaction, the candidate has been determined to be a subset of the transaction basket. If the subset condition is satisfied, the support counter is incremented. If the counter is less than m at the end of the transaction data, the candidate is not a subset of the transaction. In either case, the unit's memory pointer is reset and the process begins again for the next transaction basket, until all transactions pass through all of the units. This requires $t_t + m|C_m|$ cycles, and is thus very efficient. If $|C_m| > c_a$, there is no way to avoid passing the database stream multiple times through the units. This reduces efficiency, but is still a faster strategy than software-based sequential algorithms. The time given if multiple passes are required is as follows: given $|C_m| > c_a$, the total number of passes p is $\lceil \frac{|C_m|}{c_a} \rceil$, and thus the time for streaming transactions is $\lceil \frac{|C_m|}{c_a} \rceil\ (m|C_m| + t_t)$. The extra cycle is required to flush the support data from the linear array. The support data is collected by the controller and stored for the various control operations required to maintain a minimum support level across all candidates.

$$\begin{aligned}
&\text{for } t = \sum_{k=1}^{|T|} |T_k| + 1 \\
&\qquad \text{Out}(O_{data}, B_i) = Val(\textit{support}_i)
\end{aligned}$$

[2]The STA notation may be unfamiliar to the reader. The symbol $\bigwedge_{j=1}^{|T_k|} T_k(j)$ implies that a single element of data is generated in each cycle. Here, the transaction T_k produces its jth element as j progresses from 1 to the number of elements in the transaction.

$$\forall t.\ \sum_{k=1}^{|T|} |T_k| + 1 \le t - i \le \sum_{i=1}^{|T|} |T_k| + 1 + c_a$$
$$\text{Out}(O_{data}, B_i) = \text{In}(I_{data}, B_i)$$

4.2 Candidate Generation

We will now describe and analyze the candidate generation operation. We again utilize the functionality of the systolic array to realize the set comparison operation, but now, instead of streaming the transaction data through the array, we stream the candidate sets. This allows each candidate set to be compared against all of the other candidates sets. A sample run of the algorithm is given in Table 2 to provide an idea as to how many candidates are evaluated in each generation.

As in the support calculation, the first step here is to stream the new candidates into the linear array, with each candidate being written sequentially into the unit memories. This requires $m|C_m|$ cycles, as in the support calculation. Next, the candidates are passed through the linear array again, in order to be compared against all of the other candidates. The candidate subset information must be determined as the candidates flow through the array, and the data from those comparisons must be delivered to the controller within the confines of the systolic array structure. This is more a complex issue that the simple counter used in the support operation, requiring that the individual subset results be delivered in an identifiable form to the controller.

We determine the equivalence of sets in much the same way as we determined if the candidates were subsets of the transaction baskets in the support calculation. However, in this phase the two sets must be identical only until their penultimate item, at which point the two items are compared. If the item in the candidate memory is greater, it is then injected into the data stream[3].

The power of the systolic array lies in the ability of the designer to minimize interconnect within the device. Because we are attempting to create an area efficient implementation as well as a time efficient one, we must minimize the datapath. If c_a units are all processing a stream at the same time, at the end of m cycles any or all of the units could attempt to insert a new candidate item. It is unnecessary to send out the entire candidate set as the controller already has the candidates sets in its memory. It is only necessary to forward some sort of flag that signifies that a match has been found, and the controller can put the new candidates together before the next phase.

[3]We leave the discussion of injection for Section 4.4. For the moment we will assume that we can inject a delay into the stream and provide space for the unit to report that a potential new candidate match has been found.

Unfortunately, as any or all of the units can produce a flag at every m cycles, it is impossible to distinguish which of the units produced the flag, and when it was produced. Thus, we must forward more information. The datapath width for streaming the candidate and transaction information is equal to the width of the candidates in memory. Since the final element of the candidate memory is all that is necessary to create the new candidate, we choose to forward only the last element, appended to the candidate data that actually caused the match. This operation has no more expense[4] than forwarding a simple flag as the datapath is already in place, and makes possible the decoding in the controller as well. As discussed in Section 4.4 we can reliably inject a single element into the end of a candidate list without causing trouble anywhere else in the pipeline. This simplifies the controller as well, as the first m elements of the new candidate generation appear just before the new suffixes that will be appended to the candidate to form the new generation. This works as follows:

$U =$ number of new candidates before pruning
$t^* \overset{\text{def}}{=} t -$ stalls accumulated for item i at time t
$\forall t.\ 0 < t \le 2m|C_m|$

$$\text{Val}(\text{In}(I_{data}, B_1)) = \bigwedge_{c=1}^{|C|} \bigwedge_{e=1}^{m} \textit{external mem}(c, e)$$

if(candidate_num $=$ *unit_id*) then $\textit{mem}_i(e) = I_{data}$

$\forall t.\ 0 < t - i \le m|C_m|$
$$\text{Out}(O_{data}, B_i) = \text{In}(I_{data}, B_i)$$

$\forall t.\ m|C_m| < t^* - i \le 2m|C_m| + U$
if($I_{data} \ne \text{mem}_i(m_i)$) then *match*$_i$ = fail
else $m_i + +$
$\forall t.\ m|C_m| < t^* - i \le 2m|C_m| + U$
and $((t^* - i) \bmod m) = 0$
if(*match*$_i$ = fail) then generate$_i$
reset(match$_i$); $m_i = 0$

Figure 3 illustrates the hardware structures used in this phase. The running time of this phase suffers from the same hardware limitation problems that trouble the support phase. However, as the candidate data is orders of magnitude smaller than the penalty for having to stream the candidates through multiple times is much less of a problem. Again, the number of the times the candidates are streamed through the array is given by $\lceil \frac{|C_m|}{c_a} \rceil$.

[4]There is no expense at the cycle level, but the injection system does add a multiplexer into the forwarding datapath.

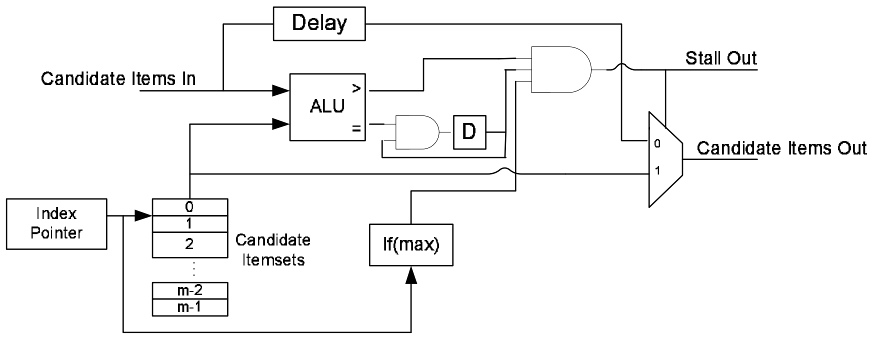

Figure 3: Architecture detail for the hardware control implementation during the candidate generation phase

If there are C_{m+1} new candidates generated while passing through the array, each of those new candidates only stalls the pipeline by one cycle, and thus the total number of cycles for the complete candidate generation phase is $\lceil\frac{|C_m|}{|C_a|}\rceil\,(2m|C_m|) + U$.

4.3 Candidate Pruning

The last highly comparison-intensive phase of the algorithm is to determine the apriori existence of subsets of the new candidates within the current generation. In this phase all of the new candidates are fed through the system again so that all possible candidate pairs can be processed. Our objective is to find if subsets of the new candidates match an element of the previous generation of candidates. One could, of course, simply create all potential subsets, stream them through the linear array, and remove any new candidates that did not have a matching ancestor for each subset. However, this would require significantly more time than our approach, by a factor of $m-1$.

Our approach is to send all candidates through the system only once, keeping a record of two pieces of data, a) if only one item is missing from a given ancestor candidate, and b) which item is missing. Because a factor of m fewer candidates are streamed through the array, we have significantly less work to do, and a total time of $(\lceil\frac{|C_m|}{c_a}\rceil\,|c_a| + 1)\,m$ cycles. We implement this by maintaing a two bit failure counter in the array unit, The key to this phase is that one failure must occur (the entire new candidate cannot be in the old candidate, by definition). Two failures, however, signify that the old candidate is not, in fact, a single-element distance from the new candidate. If the memory pointer arrives at the $(m-1)$th candidate set item with only one missing set item, it injects the missing set item into the stream. The basic subset hardware is not sufficient here, though. In comparing only one element per cycle, there is a potential the memory index in the unit may not be incremented before the

$U =$ number of new candidates before pruning

$t^* \overset{\text{def}}{=} t -$ stalls accumulated for item i at time t

$\forall t.\ 0 < t \leq 2m|C_m|$

$$\text{Val}(\text{In}(I_{data}, B_1)) = \bigwedge_{c=1}^{|C|}\bigwedge_{e=1}^{m} \textit{external mem}(c,e)$$

$$\text{if}(\textit{candidate\_num} = \textit{unit\_id}) \text{ then } \textit{mem}_i(e) = I_{data}$$

$\forall t.\ 0 < t - i \leq m|C_m|$

$$\text{Out}(O_{data}, B_i) = \text{In}(I_{data}, B_i)$$

$\forall t.\ m|C_m| < t^* - i \leq 2m|C_m| + U$

$\quad\text{if}(I_{data} \neq \text{mem}_i(m_i))$ then *failures* += 1

$\quad$else m_i++

$\forall t.\ m|C_m| < t^* - i \leq 2m|C_m| + U$ and $((t^* - i) \bmod m) = 0$

$\quad\text{if}(\textit{failures} > 1)$ then generate$_i$

$\quad\text{reset}(\text{match}_i);\ m_i = 0$

The controller collects the missing item information as it exits the array and determines if there is a failure at each of the first $m-1$ positions. Because the failing match information immediately trails the candidate it refers to, this is a simple bitmapping operation. However, if the $c_a < |C_m|$, that information has to be stored by the controller for the next pass.

4.4 Stalling Systolic Array

The previous two sections are based on the availability of a method to inject results into the datastream. Injection is important in this sort of application because any or all of the items in the array can produce data at any time over consecutive cycles. Unfortunately, this is not as simple as inserting some sort of "shadow" register to provide a delay given a downstream stall, although this is part of the solution. Without some method of controlling the data, this could result in a tangle of data collisions. Consider the candidate generation phase; $(|C_m|)^2$ comparisons are made, and were each of the c_a units to produce a result (a worst case that is not actually possible) and attempt to forward it in some way, the final unit in the chain would either have c_a collisions, or we would have to find some way to buffer out c_a elements within the units. Candidate generation and pruning is an unpredictable, entirely data-controlled operation. However, we can stall the pipeline and cause the pipeline itself to act as an efficient buffer. This is possible in data mining whereas it is less attractive in applications such as string matching because we have some flexibility as to when data is sent into the array.

One intuition is to stall the entire pipeline, and allow the collisions to clear. This is necessary to avoid data colliding in an unrecoverable way. However, in the situation of a large systolic array such as this one, global stalls are problematic for several reasons. First, it is inconvenient to have proliferation of global routing resources. The clock is the only element absolutely necessary and thus we desire to limit global routes to it. Second, several hundred *stall* signals coming together into a priority queue to stall downstream units that might collide with the requesting unit is unnecessary and wasteful. This approach would require a great deal of hardware resources. Third, stalling the whole pipeline would significantly reduce the overall utilization of the pipeline and increase execution time. We choose to pursue a more intelligent stalling approach, only stalling the previous unit and letting the stall signal propagate toward the first unit in the array. Each upstream unit handles the stall signal in manner appropriate for its current state. This allows each injection of data to only delay the pipeline by a single cycle, which is the key to the exact running time analysis in the previous sections.

Let us assume that unit i has found a matching candidate set and needs to signal this to the controller by injecting an item into the data stream. The item is forwarded to unit $i+1$, while $i-1$ receives the stalling signal. However, because $i-1$ does not receive and process the stalling signal until it has already started receiving $i-2$'s data, unit i must accept the data from $i-1$, otherwise that data would be lost. This one-element collision and matching stall continues to the first element in the chain, without causing any data loss. This strategy can be extended to allow for the potential for multiple stalls to occur at the same time in the pipeline. The key element here is that a stalled unit can accept no more than one additional piece of data. Moreover, the stalled unit will not do computation on either of the pieces of data in its memory. Thus, it cannot produce additional stalls.

Figure 4 illustrates the data movement through the pipeline and the data storage behavior within a unit.

In Table 1 we illustrate all possible combinations for a given unit. $stall_i$ is the variable declaring that unit i needs to inject a result in the next cycle. $\text{In}_{\text{stall}} = 1$ signifies that a stall has been requested from unit $i+1$, and $stall\_mem$ is the memory bit that defines if there is a standing request for a stall that has been delayed due to a coincidental stall

In_{stall}	stall_mem	generate	$\text{Out}_{\text{stall}}$	stall_mem*
0	0	0	0	0
0	0	1	1	0
0	1	d	1	0
1	0	0	1	0
1	0	1	1	0
1	1	d	1	1

Table 1: Behavior of the stalling/injection system. The *generate* signal becomes a "don't care" when the unit is already in a stalling state.

in the current unit. A unit cannot perform computation (and possibly request a stall) if it is currently has a stall request outstanding in its stall memory.incoming stall request from unit $i+1$. The output columns are the new stall request signal (routed to unit $i-1$) and the new value of the stall memory.

The intention of exhaustively enumerating the possible situations a unit can encounter serves to demonstrate to the reader that there is no way for the unit to have internal data collisions. Moreover, the progressive stalling can function effectively in a system.

5 Results

Due the lack of comparable hardware implementations, we compare against the fastest software results available running benchmark databases in standardized formats. We will base our results on a run of the T40I10D100K (15 MB) and T10I4D100k (4 MB) datasets [1] given various support levels. We compare against the APRIORI-BRAVE, Borgelt, and Goethals implementation results provided in [4]. These results are based on program execution on a 2.8 GHz dual Xeon processor machine with 3GByte RAM.

The synthesis tool for our designs is Synplicity Synplify Pro 7.2 and the place-and-route tool is Xilinx ISE 6.2. The target device is the Virtex II Pro XC2VP100 with -6 speed grade. The results are based on the placed-and-routed design. We implement only the systolic array for the results as the controller requires very little bandwidth and only moderate interaction with the array and should not significantly affect the performance results.

Table 2 has runtime information for the support calculation section of the system. As the support calculation takes an order of magnitude more time than any other of the functional segments in our implementation, it is a good approximation of the overall system performance. With only one device we can beat the much faster clocked dual device Xeon machine by at least 4x, and often by far greater margins. More importantly, the system has essentially perfect system scaling, as each additional FPGA further reduces the number of passes of the database through the system. All that is required is 20 bits of data commu-

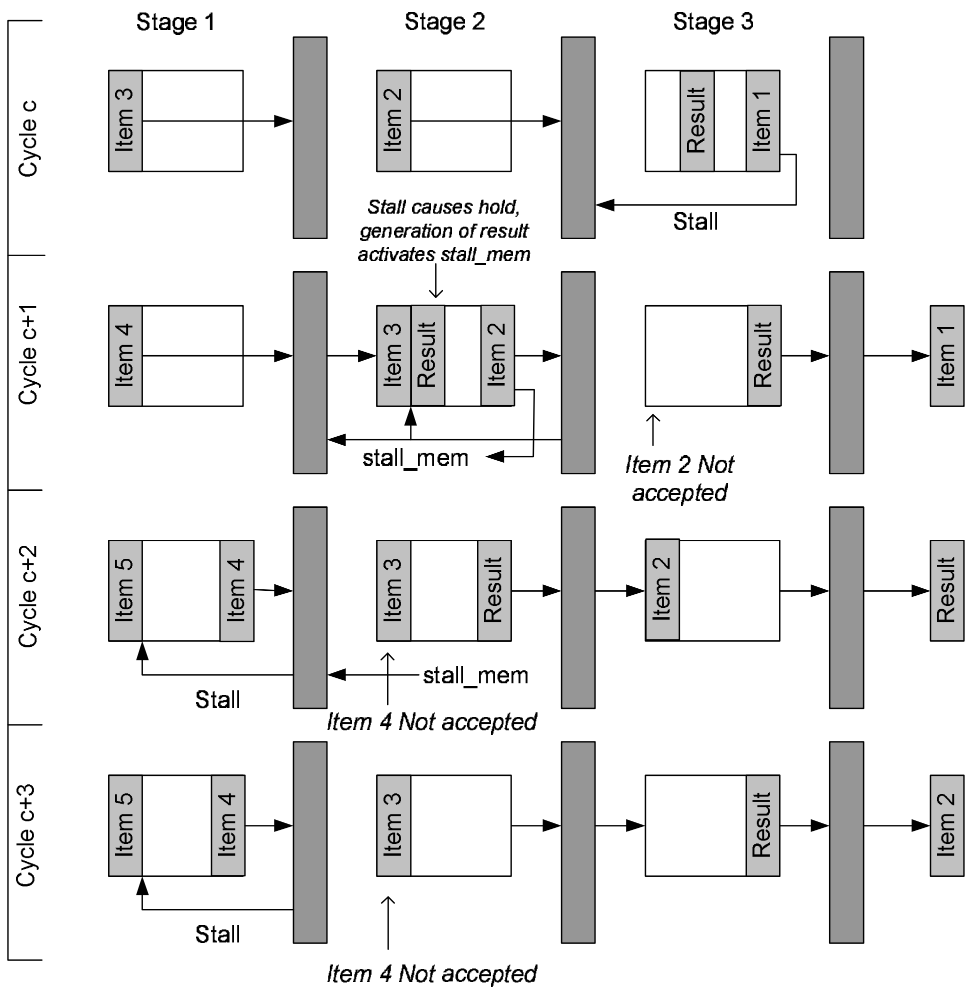

Figure 4: Systolic injection occurs in Stage 3 (cycle 1), and Stage 2 (cycle 2) but does not cause unrecoverable collisions

m	$\|C_m\|$	num. passes	time (sec)
1	100000	179	5.7
2	100000	179	5.7
3	99974	179	5.7
4	55041	99	3.2
5	28622	52	1.7
6	21642	39	1.2
7	18158	33	1.1
8	10181	19	0.6
9	7499	14	0.4
10	5251	10	0.3
11	3929	8	0.3
12	2323	5	0.2
13	1122	3	0.1
	total time for support calculation: 26 seconds		

Table 2: Performance analysis for $|T|$ = 4000000 at 112 MHz, n_a = 560. Number of candidates based on T40I10D100K dataset.

nication at 112 MHz between the devices for passing the 2-byte item datapath, stall signal, flag signal, and mode descriptor. Figure 5 illustrates the scaling performance of the design. Scaling is linear with number of devices until 64 devices. While we only do timing analysis and simulation of the systolic array in this paper, we plan on implementing the controller on either the PPC-405 on the Virtex-II Pro or in one of the SRC high-end platforms [15].

Figures 6 and 7 give comparisons to three state-of-the-art microprocessor-based implementations. We consider the T40I10D100K (15 MB) and T10I4D100k (4 MB) datasets. The two sets are standard testbench databases from FIMI [1]. The biggest difference between the two is the average number of elements in a basket, T40I10D100K having an average of 40 elements in a basket over 100,000 entries and the T10I4D100k having an average of 10 elements over the 100,000 entries. This increased basket size increases the chance that there will be correlations between the otherwise random data. This can

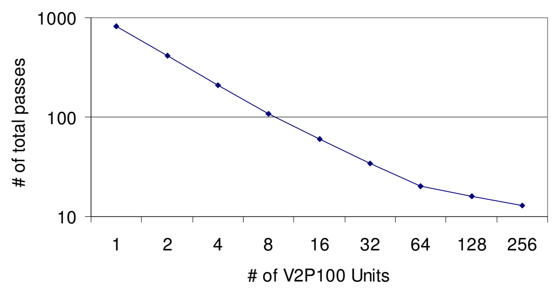

Figure 5: Scaling behavior for datamining system. Performance scales linearly until 64 devices are connected, or some 35k individual systolic array units.

be seen in the differences in overall timings between Figure 6 and Figure 7. The results in Figure 7 demonstrate that the T40I10D100K database has longer average timings due to the increased number of correlations between items to process.

A key observation is that Borgelt implementation is by far the fastest software implementation for the T10I4D100k dataset whereas the Bodon implementation is the fastest in the T40I10D100K. In both cases, our hardware approach provides consistently faster results, independent of the database characteristics. Moreover, the difference in performance increases significantly as the minimum support level decreases.

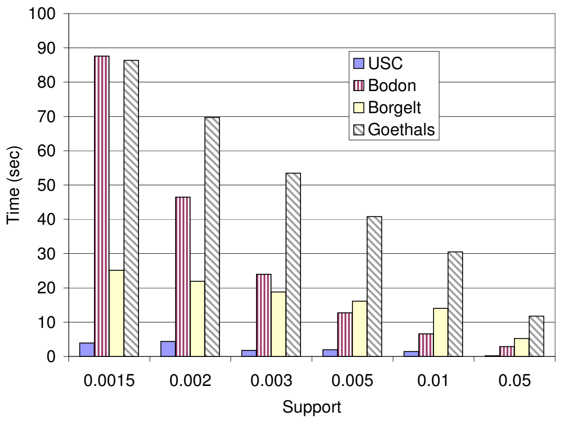

Figure 6: Performance comparison against various microprocessor-based implementations for T10I4D100K dataset.

6 Conclusion

We have shown that FPGA implementations of the Apriori algorithm can provide significant performance improvement over software-based approaches. We are also interested in implementing some of the more recent (and more control-intensive and memory-intensive) approaches in hardware, including hash-based strategies such as DHP [13] and trie-based approaches [4]. It may be possible to increase the bandwidth of the system by processing several sub-partitions of a set in parallel as in [16]).

We are also interested in leveraging our experience with high-performance string matching [3] for autonomous pattern generation for network security [14].

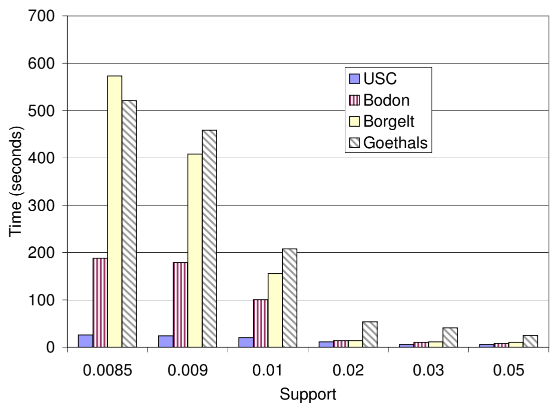

Figure 7: Performance comparison against various microprocessor-based implementations for T40I10D100K dataset.

References

[1] Frequent Itemset Mining Dataset Repository, 2004. `http://fimi.cs.helsinki.fi/data/`.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994.

[3] Z. K. Baker and V. K. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the 14th Annual International Converence on Field-Programmable Logic and Applications (FPL '04)*, 2004.

[4] F. Bodon. A Fast Apriori Implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[5] C. Borgelt and R. Kruse. Induction of Association Rules: Apriori Implementation. In *Proceedings of the 15th Conference on Computational Statistics*, 2002.

[6] M. Estlick, M. Leeser, J. Szymanski, and J. Theiler. Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware. In *Proceedings of the Ninth Annual IEEE Sym-*

posium on Field Programmable Custom Computing Machines 2001 (FCCM '01), 2001.

[7] B. Goethals. Survey on frequent pattern mining. Technical Report: Helsinki Institute for Information Technology, 2003.

[8] E.(Sam) Han, G. Karypis, and V. Kumar. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3), 2000.

[9] P. James-Roxby, G. Brebner, and D. Bemmann. Time-Critical Software Deceleration in an FCCM. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[10] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, 1979.

[11] N. Ling and M.A. Bayoumi. *Specification and Verification of Systolic Arrays*. World Scientific Publishing, 1999.

[12] J.E. Moreira, S.P. Midkiff, M. Gupta, and R. Lawrence. Exploiting parallelism with the Array package for Java: A case study using Data Mining. In *Proceedings of SuperComputing (SC) '99*, 1999.

[13] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[14] M. Qin and K. Hwang. Frequent Episode Rules for Internet Anomaly Detection. *Proceedings of the IEEE International Symposium on Network Computing and Applications (IEEE NCA'04)*, 2004.

[15] SRC Computers, Inc. `http://www.srccomputers.com`.

[16] T. Hayashi and K. Nakano and S. Olariu. Work-Time Optimal k-merge Algorithms on the PRAM. *IEEE Trans. on Parallel and Distributed Systems*, 9(3), 1998.

[17] C. Wolinski, M. Gokhale, and K. McCabe. A Reconfigurable Computing Fabric. In *Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA '02*, 2004.

[18] Q. Zhang, R. D. Chamberlain, R. Indeck, B. M. West, and J. White. Massively Parallel Data Mining using Reconfigurable Hardware: Approximate String Matching. In *Proceedings of the 18th Annual IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, 2004.