Efficient Parallel 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 and foundational member of the correlation-based datamining kernels used today. However, it is a computationally expensive algorithm and running times can stretch to days for large databases, as database sizes can reach from Gigabytes and computation requires multiple passes.

Through the use of a new type of systolic array architecture, time required for processing can be dramatically reduced. Our array architecture implementation on a Xilinx Virtex-II Pro 100 provides a 4x performance improvement over the fastest software implementation available running on a dual 2.8 GHz Xeon system with 3 GB RAM. The system is easily scalable and introduces an efficient "systolic injection" method for intelligently reporting unpredictably generated mid-array results to a controller without chance of collision or excessive stalling. The off-chip bandwidth required is 250MB/s, allowing for practical implementation with off-the-shelf memory controllers. The off-chip memory requirements are negligible.

1 Introduction

Recent advances in storage and data sensing has 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 that is useful as historical record but not as useful as it could be were it effectively processed for patterns and trends. Correlation-based datamining is the field of algorithms to process this data into more useful forms, in particular, connections between sets of items. In this paper, we

[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.

investigate the Apriori algorithm [2], a popular strategy designed for progressively grouping together frequent itemsets in large databases given a particular frequency 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 doing this work 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 [11]. We also present a strategy called "systolic injection," a significant technical contributions 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 packaged units. We implement all data and computation intensive operations within the systolic array and implement all serial and control intensive operations within a microprocessor, namely the Power PC 405 integrated with the Virtex-II Pro device. The device choice is important, as the systolic array is dependent on its data source and sink, and for the "software deceleration" [10] for control-intensive operations that take up a small fraction of the total running time of the algorithm. Due to our streaming implementation, the candidate generation phase of the algorithm require orders of magnitude less time than the support calculation, and overall requires roughly 25% of the time required by the fastest non-supercomputer implementation [5]. The off-chip memory required is negligible beyond the size of the database and the bandwidth between memory and the systolic array is only 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.

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 datamining algorithms has been done [6, 12, 20, 21].

In [6, 20] the k-means clustering algorithm in 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 datamining 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 any manner of other data types. Each item in a set is randomly assigned to a cluster, the centers of the clusters are computed, and then elements are added and removed from clusters to more efficiently 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, but adds the distance computation and significantly changes how the sets are built up. Besides differing in the overall algorithm, the structure of the computation is also significantly different, as the system requires the use of global memory, in which each unit's personal memory is accessible by the host controller. By avoiding global connections that violate the principles of systolic design, we can increase overall system clock frequency and ease routing problems.

In [21] 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. This paper proposes the integration of simple computational structure for datamining onto the hard drive controller itself. The datamining 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 an efficient software implementation [5] using a trie approach as we are unaware of any comparable hardware implementations of the Apriori algorithm. Extensive research exists [9, 13] on parallelizing correlation algorithms, but we focus on single machine performance.

3 Introduction to the Apriori Algorithm

We break the Apriori [2] algorithm into three sections, as illustrated in Figure 1. Initial frequent item sets are fed into the system, and candidate generation, candidate pruning, and candidate support 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. We will first introduce some of the datamining lexicon and then describe the operational phases in more detail.

In the literature, an analogy to a shopping cart is used: the set of items purchased at one time, checked out from the library, or otherwise grouped together based on some critera such as time, customer, etc. is referred to as a *basket*. The items within the basket can be the entire transaction, or there may be multiple transactions within the basket. A *frequent itemset* is the a set of one or more items that often occur in a database one item, and 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, guaranteeing 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, if not socially responsibly, to put a beer refrigerator in the diaper aisle.

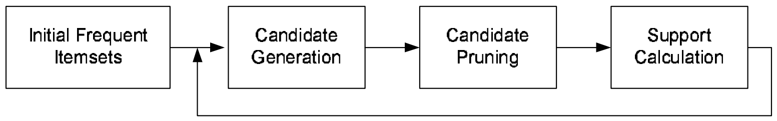

Figure 1: Process flow of the data mining system

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

```
$\forall\ f_1, f_2 \in F_k$ do
  with $f_1 = (i_1, ..., i_{k-1}, i_k)$
  and $f_2 = (i_1, ..., i_{k-1}, i_k^*)$
  and $i_k < i_k^*$
      $f := f_1 \cup f_2 = (i_1, ..., i_{k-1}, i_k, i_k^*)$
```

It should be noted that only ordered sets are utilized. Thus, when f is generated from f_1 and f_2, the sets remain ordered. Candidate generation pairs up any candidates that differ only in their final element to generate 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 every subset of it is also present in the previous generation, as follows:

$\forall i \in f : f - \{i\} \in F_k$

Thus, our initial candidate generation proves by design that if we remove either of the last two items (i_k, i_k^*) from the new candidate, we will get candidates from the previous generation, namely, f_1 and f_2. The second step proves that if we remove any of the other items from the new candidate, we must 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 it is during this phase the database is streamed into the system. Each potential candidate's support, or 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 make up 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 $c \in t$
            support(c)++
```

The main problem with the Apriori algorithm is this data complexity. Each candidate must be compared against every transaction data, and candidate generation must see the entire database transaction set. This gives a large running time for a single generation, $O(|T||C||t|)$, assuming the subset function can be implemented in constant 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 insure that our memories are of minimum size and yet also be sufficient to store all data necessary.

Our architecture allows for roughly 560 units on a single device, assuming perfect routing and placement (this is based on place and route of the full systolic array design on a Xilinx Virtex-II Pro 100 device with 44,000 slices. Hardware usage is roughly 70 slices per unit with resources for up to 16 2-byte item candidate sets). This units are all connected end-to-end in the form of a linear array, and contain memory locations to temporarily store the candidates whose support is being calculated and to allow for stalling. The hardware is composed of the candidate memory, an index counter,

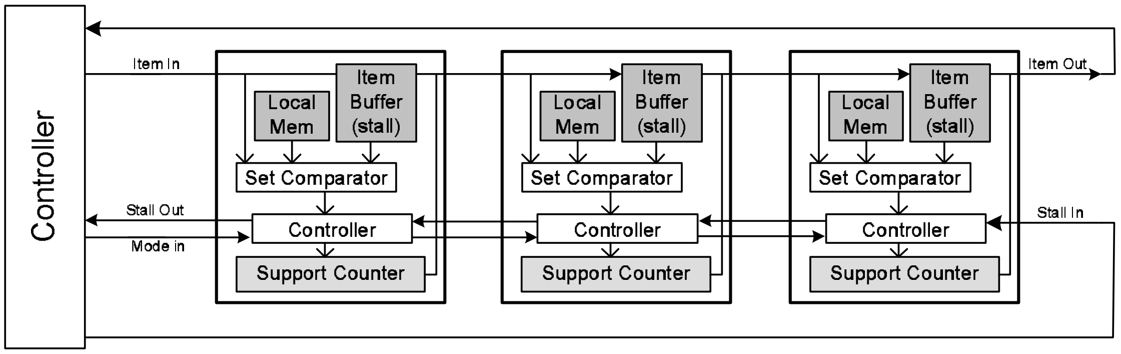

Figure 2: Illustration of the general data mining systolic array

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 based on the current mode to implement the required functionality in the three computational phases.

Throughout this section, we will use the following variables:
m, the number of items in a candidate list, also the generation number as one item is added to the list per generation
c_t, the number of total candidates,
c_a, the number of hardware units (c_a may be less than c_t),
t_t, the total number of items in the database,
t_b, the number of individual baskets in the database

4.1 Support

The support calculation is by far the most simple of the operations in the Apriori algorithm to implement given its use of two loops with no dependencies. Figure 3 illustrates the hardware implementation of the support functionality.

The first step is to load the units with candidates. Data enters at one end of the linear array, and, after placing the first candidate in the first unit, is forwarded to the next unit and so on, until all units in the array are full. This requires $m\,c_t$ time overall. The time, however, may be split into multiple sections should $c_t > c_a$.

Next, the transactions are streamed into the units. Each transaction is also an ordered set to match the ordered candidate sets, so finding if the candidate is a subset of the transaction basket is very 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 (it may be possible to increase the bandwidth of the system by processing several sub-partitions of a set in parallel [17]). Given that the subset is a candidate, the support counter is incremented. If the counter is less than m at the end of the transaction data, the candidate is not present in the transaction. In either case, the unit's candidate pointer is reset and the process begins again for the next transaction basket, until all transactions pass through the all of the units. This requires $t_t + c_t$ cycles, and is thus very efficient. If $c_t > c_a$, there is no way to avoid passing the database stream through the units multiple times. This reduces inefficiency, but is still a far more efficient strategy than software-based sequential algorithms. The time given multiple passes is as follows: if $c_t > c_a$, then the total number of passes p is $\lceil \frac{c_t}{c_a} \rceil$, and thus the time for streaming transactions is $\lceil \frac{c_t}{c_a} \rceil (c_t + 1)$. 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.

The overall time for the support calculation is $\lceil \frac{c_t}{c_a} \rceil (t_t + 1) + c_a$. After the streams have moved through the candidates, The pipeline latency is only included once as after the pipeline is full it remains full until the beginning of the next phase.

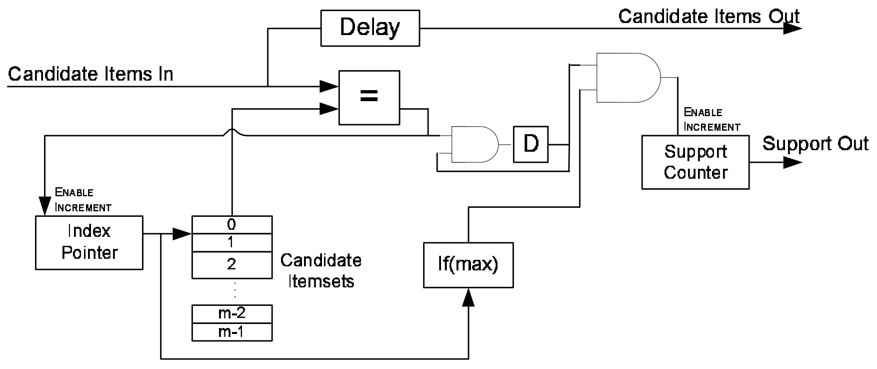

Figure 3: Architecture detail for the hardware implementation of the support calculation

4.2 Candidate Generation

We will now describe and analyze the candidate generation operation. Because each candidate is compared against every other candidate, the complexity of this operation is $m^2\, c_t$. 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 data a second or more times. In this way, the size of the candidate memories remains set to the maximum m required, and no other memory is required.

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 c_t, as earlier. The second pass through the array is more complicated, as the candidate data must be matched as it flows through the array, and data from those comparisons must be delivered to the controller within the confines of the systolic array structure.

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 only match until their penultimate item, at which point the two items are compared and, if the item in the candidate memory is greater, is injected into the data stream[2].

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 none, any, or all of the units could produce a new candidate item. It is unnecessary to send out the entire candidate as the controller already has the candidates 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 during before the next phase.

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. Thus, we must forward more information. As the datapath for streaming the candidate and transaction information is equal to the size of the candidates in memory, and 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[3] 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.

Figure 4 illustrates the hardware structures used in this phase. The running time

[2]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

[3]There is no expense at the cycle level, but the injection systems does add a multiplexer into the forwarding datapath, and the multiplexer has controlled by a signal with a long critical path.

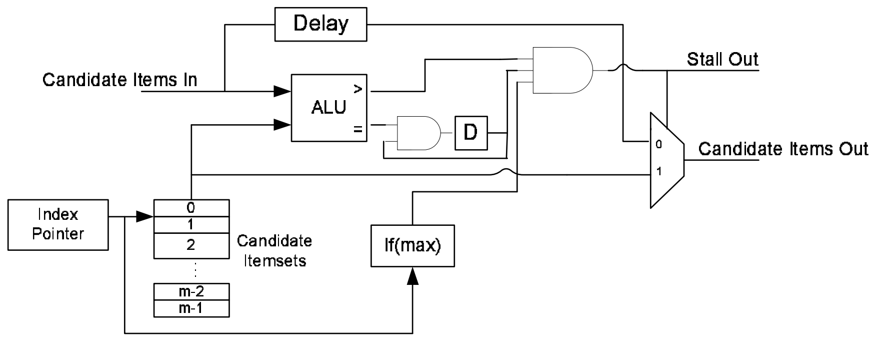

Figure 4: Architecture detail for the hardware implementation of the candidate generation

of these phase suffers from the same hardware limitation problems that trouble the support phase, but, as the candidate data is orders of magnitude smaller 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 $\lceil \frac{c_t}{c_a} \rceil$. If there are c_n new candidates generates while passing through, each of those new candidates only stalls the pipeline by one cycle, so end with the total number of cycles for the complete candidate generation phase is $\lceil \frac{c_t}{c_a} \rceil c_t + c_n$.

4.3 Candidate Pruning

The last highly comparison-intensive phase of the algorithm is determining the apriori existence of subsets of the new candidates within the current generation. Essentially our objective in this phase is to take all of the new candidates and feed them through the system again and find if subsets of the new candidates match the previous candidates. One could, of course, simply create all potential subsets and then stream them through the linear array and then remove any new candidates in which any of the subsets did not find a matching ancestor. However, this would require significantly more time, by a factor of $m - 1$.

Our approach sends all candidates through only once, but keeps track of two pieces of data, a) if only one item is missing from a given ancestor candidate, and b) which item. Because a factor of m fewer candidates are streamed through the array, we have significantly less work to do, and a total time of $(\lceil \frac{c_t}{c_a} \rceil c_n + 1) m$. We implement this by maintaing two bits in the unit, one defining a primary match failure and one defining a second failure. The key to this phase is that one failure must occur, implying that the subset is present (the entire new candidate cannot be in the old candidate, by definition), but two failures signifies that the old candidate is not, in fact, a subset of the new candidate. When the candidate pointer arrives at $m - 1$ (which are not required as they have already been proved to have subsets in the candidate generation phase) it injects the failing m matches into the stream. The controller collects the failing match 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 $n_a < n_t$, 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. This 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, and, without some method of controlling the data, could produce a tangle of data collisions. Consider the candidate generation phase; $(c_t)^2$ comparisons are made, and should every one of the c_a units produce a result (an impossibility) 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. Unlike an application with a predictable maximum-size buffer model such as string matching under the Knuth-Morris-Pratt algorithm [4], 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 where it is less attractive in mostly-real-time applications such as string matching for intrusion detection because we have a great deal of flexibility in when data is provided to the array.

The first intuition is to stall the entire pipeline, and allow the collisions to clear. This is necessary in that case 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 will try to limit global routes to it. Second, several hundred *stall* signals coming together into a priority queue that only needs to stall 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 delay termination. We choose to pursue a more intelligent stalling approach, instead only stalling the previous unit and letting the stall signal propagate toward the first unit in the array. 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 an 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 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.

Figure 5 illustrates the data movement through the pipeline and the data storage behavior within a unit.

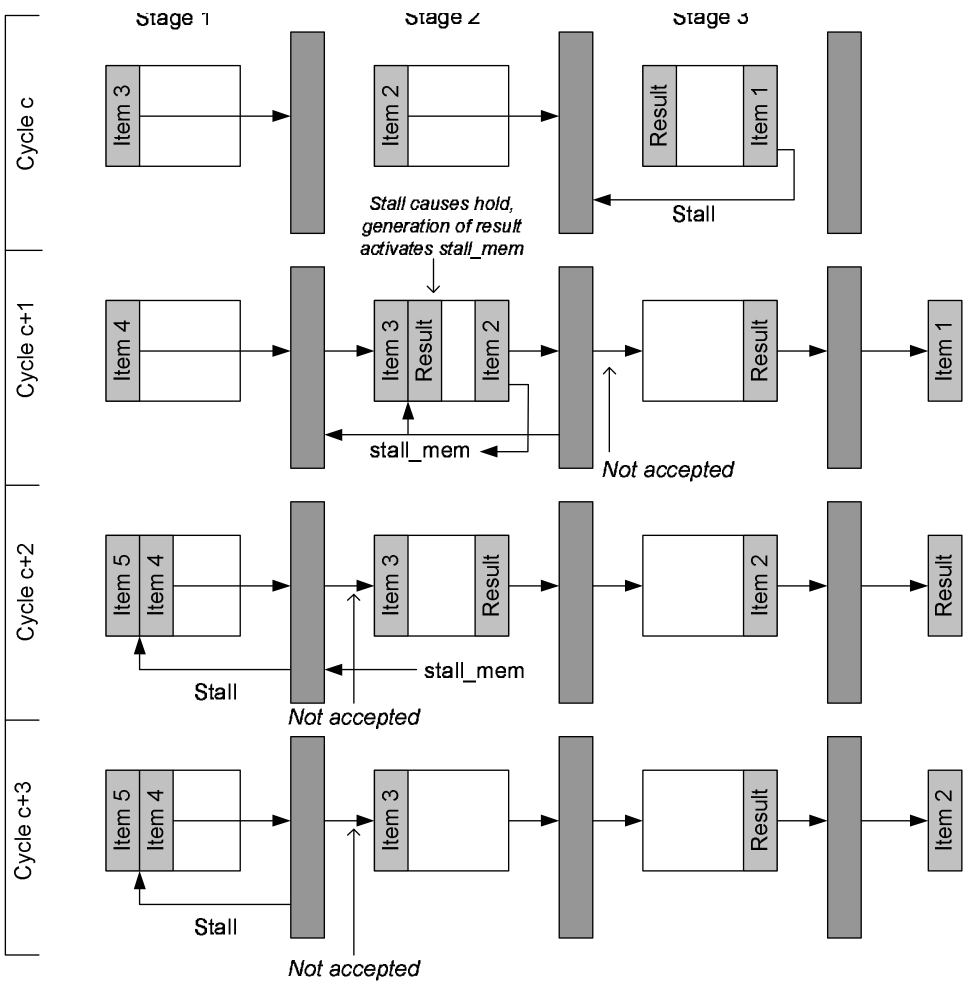

Figure 5: Systolic injection occurs in Stage 3(cycle 1), and Stage 2(cycle 2) but does not cause unrecoverable collisions

We illustrate in Tables 1 and 2 two potential situations in which stalls occur and propagate up the array without causing loss of data. The key element here is that when a unit is stalled, it can accept one more piece of data, but no more, and, moreover, it does not do computation on either of the pieces of data in its memory. Thus, it cannot produce a stall on its own.

	Stage Number																
Cycle Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
1	0	1	2	3	4	5	6	7	8								
2		0	1	2	3	4	5	6	7	8							
3			0	1	2	3	4	5	6	7	8						
4				0	1	2	3	4	5	6	**7s**	8					
5					0	1	2	3	4	**56s**	7	i	8				
6						0	1	2	**34s**	5	6	7	i	8			
7							0	**12s**	3	4	5	6	7	i	8		
8							**0s**	1	2	3	4	5	6	7	i	8	
9								0	1	2	3	4	5	6	7	i	8

Table 1: A single injection i, stalling the pipline by one cycle

	Stage Number																
Cycle Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
1	0	1	2	3	4	5	6	7									
2		0	1	2	3	4	5	6	7								
3			0	1	2	3	4	5	6	7							
4				0	1	2	3	4	5	6	**7s**						
5					0	1	2	3	4	**56s**	7	i					
6						0	1	2	**34s2**	5	6	7	i				
7							0	**12s2**	**i3s**	4	5	6	7	i			
8							**0s2**	**12s**	**is**	3	4	5	6	7	i		
9							**0s**	1	2	i	3	4	5	6	7	i	
10								0	1	2	i	3	4	5	6	7	i
11									0	1	2	i	3	4	5	6	7

Table 2: Two injections i colliding and causing two stalls to propagate in the pipeline. This situation is illustrated in more detail in Figure 5.

In Table 3 we illustrate all possible combinations for a given unit. *Need_stall_now* is the variable declaring that unit i needs to inject a result in the next cycle. *Stall_in* signifies that a stall has been requested from unit $i + 1$, and *my_stall_mem* is the memory bit that defines how many stalls have built up due to coincidental requests. The table has an added column, *need_stall_now_real* that is the adjusted value of *need_stall_now* to account for the definition of the stalling behavior, that is, a unit cannot request a stall if it is currently has a stall request outstanding in its stall memory.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 possibly situations a unit can encounter serves to demonstrate to the reader that there is no way for the unit to get backed up with more than one element, and moreover, the progressive stalling can function effectively in a system.

Need_stall_now	Stall_in	My_stall_mem	Need_stall_real	stall_out	My_ stall_mem*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	0	1	1

Table 3: Behavior of the stalling/injection system. Note that a modified need_stall_now is required based on the current state of my_stall_mem, as active high my_stall_mem implies that the last cycle

5 Results

Due the lack of comparable hardware implementations, we compare against the fastest software results available [5] running available databases in standardized formats, namely, [1]. We will base our results on a run of the T40I10D100K dataset given a support of 0.01. The Bodon implementation [5], running on a 2.8 GHz dual Xeon processor machine with 3GByte RAM requires roughly 100 seconds. We base our comparisons on this number.

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. We will do performance verification on the Xilinx ML-300 platform, but currently the results are based on the place and route report. 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 4 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, it is a good approximations of the overall system performance. With only one device we can beat the much faster clocked dual device Xeon machine by 4x. More importantly, the system has essentially perfect system scaling, as each additional FPGA reduces the number of passes of the database through the system. All that is required is 20 bits of data communication at 112 MHz between the devices for passing the 2-byte item datapath, the stall signal, and a flag signal. Figure 6 illustrates the scaling performance of the design. Scaling is linear with number of devices until 64 devices. Granted, this would be expensive, but from a performance point of view it is worthwhile. 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 [16].

m	num_candidates	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 transaction: 26 seconds		

Table 4: Performance analysis for nt = 4000000 at 112 MHz, na= 560. Number of candidates based on T40I10D100K dataset.

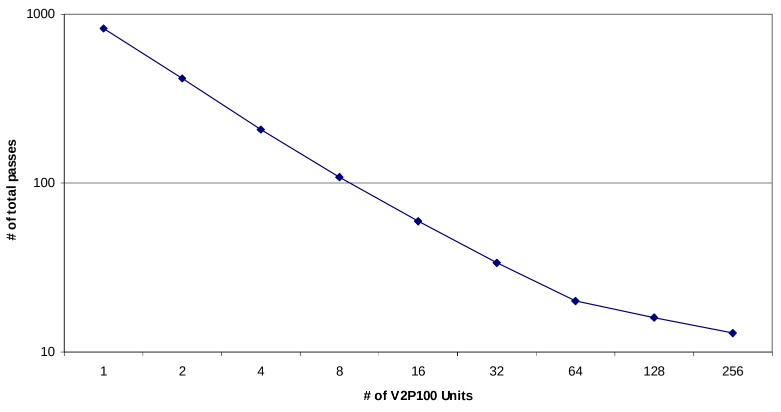

Figure 6: Scaling behavior for datamining system. Performance scales linearly until 64 units

6 Conclusion

We have shown that FPGA implementations of the Apriori algorithm can provide significant performance improvement over software-based approaches. We have observed strategies involving the interchange of the nested loops that provide performance in a way that is complimentary to the design in this paper. We are interested in utilizing the reconfigurable aspect of the hardware to switch operational modes at a 'sweet spot' during execution. 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 [14] and trie-based approaches [5], as well as leveraging some of the recent work in reconfigurable hardware-based neural networks [7]. We are also interested in leveraging our experience with high-performance string matching [3] along with a non-discrete itemset modification [8] for autonomous pattern generation for network security [15].

References

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

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM SIGMOD Conference*, 1993.

[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] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *The Twelfth Annual ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, 2004.

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

[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 Symposium on Field Programmable Custom Computing Machines 2001 (FCCM '01)*, 2001.

[7] K. Gaber, M.J. Bahi, and T. El-Ghazawi. Parallel Mining of Association Rules with a Hopfield-type Neural Network. In *Proceedings of Tools with Artificial Intelligence (ICTAI 2000)*, 2000.

[8] E. (Sam) Han, G. Karypis, and V. Kumar. Min-Apriori: An Algorithm for Finding Association Rules in Data with Continuous Attributes, 1997.

[9] 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.

[10] 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.

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

[12] K. Leung, M. Ercegovac, and R. Muntz. Exploiting Reconfigurable FPGA for Parallel Query Processing in Computation Intensive Data Mining Applications. In *UC MICRO Technical Report Feb. 1999*, 1999.

[13] 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 SuperCcomputing (SC) '99*, 1999.

[14] 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.

[15] 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.

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

[17] 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.

[18] The Xilinx Corporation. ML-300 Development Board. `http://www.xilinx.com/ml300`, 2004.

[19] The Xilinx Corporation. Virtex II Pro Series FPGA Devices. `http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+%Pro+FPGAs`, 2004.

[20] C. Wolinski, M. Gokhale, and K. McCabe. A Reconfigurable Computing Fabric. In *Proceedings of the* , 2004.

[21] 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.