An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems[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 fundamental correlation-based data mining kernel used in a variety of fields. The innovation in this paper is a highly parallel custom architecture implemented on a reconfigurable computing system. Using this "bitmapped CAM," the time and area required for executing the subset operations fundamental to data mining can be significantly reduced.

The bitmapped CAM architecture implementation on an FPGA-accelerated high performance workstation provides a performance acceleration of orders of magnitude over software-based systems. The bitmapped CAM utilizes redundancy within the candidate data to efficiently store and process many subset operations simultaneously. The efficiency of this operation allows 140 units to process about 2,240 subset operations simultaneously.

Using industry-standard benchmarking databases, we have tested the bitmapped CAM architecture and shown the platform provides a minimum of 24x (and often much higher) time performance advantage over the fastest software Apriori implementations.

1 Introduction

Recent advances in state of the art high performance computing machines [8, 12] with large memories and large FPGA devices have caused a seismic shift in the way reconfigurable hardware is used. In these machines, the notion of an FPGA is not an afterthought. These machines are not a card or an add-on device that attempts to provide acceptable performance and usability, rather, they are a tightly integrated system.

The development of FPGA-integrated machines with large, fast SRAM banks and high bandwidth interconnect has set the stage for orders of magnitude higher efficiency for highly complex applications. The ability to stream data at high speeds from memory to an FPGA with integrated control and bitstream loading provided by a microprocessor has made systems such as the SRC MAP processor architecture [12] or the Cray XD-1 system [8] potentially very valuable.

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, 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. There are various forms of data mining. Much is for commercial purposes, but other work including clustering for bioinformatics [11] and correlation mining for evolutionary traits [4]. Correlation-based data mining is the field of algorithms to process this data into more useful forms, in particular, connections between sets of items. The Apriori algorithm [2] is a popular approach for progressively grouping together frequent itemsets in large databases given a particular cutoff of occurrence frequency.

Software implementations of the Apriori algorithm [6, 7, 10] utilize various tree structures, hashing methods, or approaches such as vertical mining [15] that radically change data structures to handle the support and candidate generation operations.

This paper demonstrates an efficient structure for computing the support of a set of candidates. The effectiveness of the architecture has been shown on the SRC FPGA machine, proving that the integration of FPGA acceleration and software control allows for significant advances in the design of data mining systems. While the architecture is designed for implementation on an SRC machine, other reconfigurable computing systems such as the Cray XD-1 [8] or a custom ASIC-based design would also provide high performance with few architectural changes.

In various performance studies, it was noted that the behavior of the Apriori algorithm has certain characteristics that allow for redundancy between candidates to be extracted for use in hardware. In general, while there may

[1]Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by equipment grants from the Xilinx and SRC Computers corporations.

be hundreds of thousands of candidates processed in a generation, the difference between any two adjacent candidates is fairly low. That is, the edit distance between candidate sets will be low. Through the combination of Content-Addressable Memories (CAM) implemented on FPGA and small internal memories, the performance of the algorithm in hardware can be improved. The results of tests on various correlation-mining benchmark databases show that the bitmapped CAM approach is highly efficient in both time and area compared to other hardware approaches.

In the following sections, the related work in the field will be introduced, and some background on FPGA architecture and the Apriori algorithm itself will be provided. A detailed description of the approach follows, as well as the rationale behind the approach. Finally, implementation details and results for the system will be presented.

2 Background on Reconfigurable Computing Systems

Field Programmable Gate Arrays (FPGA) provide a fabric upon which applications can be built. FPGAs, in particular, Static Random-Access Memory (SRAM) based FPGAs from Xilinx [14] or Altera [3] are based on a look-up tables, flip-flops, and multiplexers. In these devices, a SRAM bank serves as a configuration memory that controls all of the functionality of the device, from the logic implemented to the signaling standards of the IO pins. The values in the look-up tables can produce any combinational logic functionality necessary, the flip-flops provide integrated state elements, and the SRAM-controlled routing direct logic values into the appropriate paths to produce the desired architecture. The device is composed of extensive programmable routing and many thousands of basic *logic cells* that include the basic logic elements. Depending on on the device variety, FPGAs can include fast Application Specific Integrated Circuit (ASIC) multipliers, ethernet controllers, local RAMs, and clock managers. FPGAs started out as prototyping devices, allowing for convenient development of glue-logic-type applications for connecting ASIC components without high VLSI design costs or large numbers of discrete standard logic gates. As the gate density of FGPA devices increased and application specific ASIC blocks were added, the applications shifted from glue logic to a wide variety of solutions for signal processing and network problems. The devices have been deployed in the field as the final, but still flexible, solution. Because SRAM-based devices are controlled by the state of the memory bits, the functionality can be changed by changing the memory state. This can be useful, as logic can be customized for a particular set of input data.

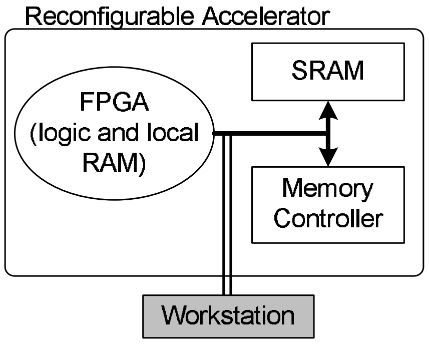

Figure 1: A general platform model is targeted, composed of a microprocessor-based host connected via fast interconnect with FPGA fabric and several banks of SRAM

FPGA devices provide a surprisingly powerful and convenient platform for implementing application accelerators. For some kernel domains, such as data mining, FPGAs provide many of the advantages of ASIC design, including potential for parallelism, very efficient bit-level operations, and high bandwidth. Unlike ASICs, though, reconfigurable hardware can be reprogrammed to suit a specific set of input data or operating situations, allowing the flexibility of software with the power of fixed hardware.

The trend for accelerating general computation is a generic platform shown in Figure 1. By providing a workstation with an FPGA accelerator combined with banks of fast memory, machines like the SRC MAPstation [12] or the Cray XD-1 system [8] can provide significant performance increases for scientific and data-intensive computing. The SRC system uses a reconfigurable device integrated into the DIMM memory slots, allowing the FPGA to take advantage of the high memory bandwidth as well as transparently behave as a memory.

3 Related Work in Hardware Data Mining

As far as we know, the Apriori algorithm has not been studied in any significant way for efficient hardware implementation by other researchers. Currently, our initial paper ([5]) on a systolic array architecture for data mining is the only recent work in this area. However, research in hardware implementations of related data mining algorithms has been published [9, 13, 16].

In [9] and [13] 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 each group are computed. The elements are then iteratively moved between clusters to move them closer to the center of a group. 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 [16] 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.

In [5], we developed a systolic array architecture for data mining that allows the time required for all phases of the apriori algorithm to be significantly reduced. The array architecture provides a performance improvement of orders of magnitude over 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. However, the support operation still requires an two orders of magnitude more time than the other stages. We focus on addressing the support problem in this paper. The architecture we develop in this paper is entirely distinct from the work in [5].

4 Apriori Algorithm

For the benefit of the reader, we present a brief introduction to the Apriori algorithm and the challenges of implementing it efficiently in hardware. We divide the Apriori [2] algorithm into three sections, as illustrated in Figure 2. Initial frequent itemsets are fed into the system, and the three phases of 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 the hardware implementation, multiple FPGA configurations are used throughout the various modes of operation to provide the highest possible performance. In this section, 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 frequent items that often occur together in the same basket. The cutoff of how often a set must occur to be included in the candidate set is the *support*.

Throughout this paper, we will use the following variables:
m, the number of items in a candidate list, equal to the generation number (one item is added to the list per generation)
$|C_m|$, the number of total candidates in generation m,
c_a, the number of candidate slots available (16 slots per hardware block)(c_a may be less than $|C_m|$),
$|T|$, the number of individual transaction baskets in the database
$t_t = \sum_{k=1}^{|T|} |T_k|$, the total number of items in the database,

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.

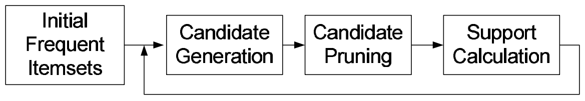

Figure 2: Process flow of the data mining system

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:

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

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:

```
when $m > 1$
    $\forall c \in C_m$ 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 time performance bottleneck for the Apriori algorithm is determining if each candidate is a subset of each transaction basket. Each candidate must be compared against every transaction set, a highly compute-intensive operation. The focus of this paper is on accelerating the support operation.

The analysis and results are based on two benchmark databases that are commonly used for testing the performance of implementation of the Apriori algorithm. The T40I10D100K (15 MB) and T10I4D100k (4 MB) datasets [1] with various support levels are tested.

5 Architectural Approach

In [5] we observed that the huge amount of data that must be streamed through the device causes the support operation to require two orders of magnitude more time than any other segment of the Apriori algorithm. This work addressing only the acceleration of the support calculation. The architecture presented in this paper is based on a several observations about the behavior of the algorithm. The most important is that the output of the candidate generation phase produces a series of very similar candidates.

For instance, A B C and A B D are both valid candidates, as well as A C D E and A B D E. If we can avoid replicating the storage resources for the replicated data, a higher level of efficiency can be achieved. If each hardware block handles a single candidate (as in the earlier architecture [5], then A C D E and A B D E will each require four unique elements, for a total of 8 memory locations across the two blocks. However, if the hardware block can be responsible for both candidates, the total number of unique elements is 5.

There is often a very little change between two successively generated candidates in the candidate set. A small extract from the 7th iteration of candidate generation follows:

```
249 316 395 482 743 787 819
236 249 395 482 743 787 819
249 316 395 482 743 787 804
236 249 395 482 743 787 804
236 249 316 395 482 743 787
249 319 482 620 743 787 819
249 482 620 743 787 804 819
249 316 482 620 743 787 819
236 249 482 620 743 787 819
249 482 529 620 743 787 819
249 319 482 620 743 787 804
```

Note that the total number of unique elements is very low. This is due to the regularity of the generation procedure.

$c := c_1 \cup c_2 = (i_1, ..., i_{m-1}, i_m, i^*_m)$

As $c_1(i_1, ...i_{m-1}) = c_2(i_1, ...i_{m-1})$ one can expect large groupings of very similar candidates.

Searching for subsets in a single unit (as in [5]) is quite simple. 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.

Handling many candidates in the same block is slightly more complicated. In the single candidate case, the candidate has a candidate pointer that keeps track of the next item of interest. Through the use of a memory for the candidate item, only a single comparator is required. As comparators are fairly expensive in terms of hardware resources, this is a good solution. However, in the multiple candidate situation, many items can require comparison with the input in any cycle. This either causes stalls in the pipeline while the various items are compared, or causes errors as candidate pointers are advanced or stalled incorrectly. The transaction must have all elements of a given candidate in order for the subset property to be satisfied.

The solution is to utilize a CAM linked with a bitmap memory in order to determine subset satisfaction for a large number of candidates simultaneously. Figure 3 illustrates the architecture. A CAM is a fully associative, programmable memory. In a CAM, a user requests the *content* of a memory instead of the location of a particular element. That is, instead of requesting location 4 of some memory and getting the output 35A3, we request the 35A3 and the memory returns the address 4. These sorts of memory are commonly used in network applications. We use the CAM to provide parallel comparisons for the union of the items present in the candidates sets.

5.1 Bitmapped CAM

In Figure 3, the CAM addresses a line from the bitmap memory. A set bit in the bitmap memory output corresponds to the presence of an item in the candidate set. Thus, the vertical line in the RAM corresponds exactly to the items in the candidate.

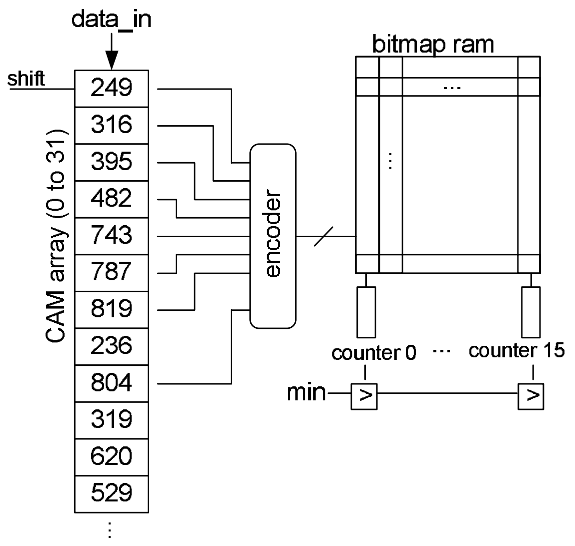

Figure 3: Architecture of the bitmapped CAM block. Each of the items included by the candidates of interest has its own CAM entry, indexing a RAM that specifies which of the candidates the item applies to.

The elements in the CAM S^b_{cam} is the set of items required to completely specify the candidate sets contained within block b, as follows:

```
b = 0
$\forall c \in C_m$ do
        if( $|S^b_{cam}| + |(c \setminus S^b_{cam})| > 32$)
            b++
        else
            $S^b_{cam} = S^b_{cam} \cup c$
```

Bitmap$_b$ for block b is filled as follows:

```
$\forall i \in S^b_{cam}$ do
        $\forall c \in C_m$ do
                if $i \in C_i$
                    bitmap$_b$(i,c) = 1
                else
                    bitmap$_b$(i,c) = 0
```

As a transaction is streamed through the system, each item in the transaction set is presented to the CAM input. If the incoming item matches any of the items in the CAM, the CAM produces an address that corresponds to that item in the bitmap, as in Figure 3. If a bit of the RAM output is set, the corresponding candidate counter increments. At the end of the transaction, the counters are inspected. If a counter's value is equal to the size of the candidate, all of the candidate's items were present in the transaction. This is guaranteed as an item occurs only once in any transaction, and the counter is only incremented when the bitmap indicates that the candidate requires the item.

5.2 Implementation of the Bitmapped CAM

The performance of the system is highly dependent on how quickly the support operation can be executed, and this is entirely dependent on the speed at which data can be streamed and the subset operation completed for each candidate. As the total number of candidates is in the tens of thousands, and the system can only support a limited number of candidates at any given time, multiple iterations may be necessary. Thus, it is important to maximize the total number of candidates that can be processed in parallel in a given iteration. This is directly related to the area required per candidate in hardware, and the frequency of the hardware clock. An appropriate performance metric, then, is (time * area).

The elements are connected end-to-end in a one-dimensional array. Data flows in a single direction, and there are no global connections except for the clock signal. This allows for a faster clock rate than designs requiring long wires. The support operation is broken into three steps;

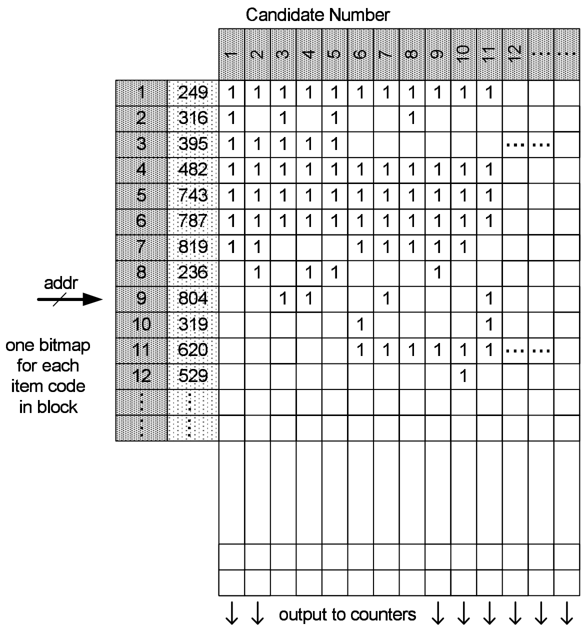

Figure 4: Architectural detail for the bitmap operation. Set bits in a row represent the candidates that contain a given CAM entry. Set bits in a column represent the CAM entries in a candidate.

namely, programming, streaming of transaction data, and collection of support results.

The first step is to load the units with candidates. Programming information 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 b CAM blocks and bitmaps for each block in the array is full. This requires $b\,|S_{cam}| + m\,|C_m|$ cycles. The time, however, may be split into multiple sections if $|C_m| > c_a$.

The transactions are streamed through the array. All transaction are sent through, one element per cycle. 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 counter 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 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 $\lceil \frac{|C_m|}{c_a} \rceil\,(t_t + 1)$. 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 programming information for the systolic array is generated by the host. As the generation of the CAM table and bitmap is a simple operation, it can be executed in minimal time.

5.3 Analysis of the Approach

The efficiency of the bitmapped CAM is based entirely on the similarity of the candidates processed. Determining the appropriate ratio number of CAM elements to candidates in a unit is somewhat of a challenge.

Utilizing a sample run from the T40I10D100K database, we approach the problem as simply as possible. Each candidate is processed directly from a file containing all of the candidates for a given generation. The unique elements from each candidate are added to the set S_{cam} until there is either no more CAM elements available or the maximum number of candidates has been reached. For $|S_{cam}|$=32 and $|C|$ = 16, the behavior of the candidate set is as follows:

Generation Number	Max CAM elements	Num blocks above 32 max	Max cand. in 31 CAM entries
1	16	0	–
2	18	0	–
3	22	0	–
4	31	0	–
5	41	1	12
6	28	0	–
7	41	2	9
8	29	0	–
9	31	0	–
10	41	1	15
11	25	0	–

Table 1: Detailed CAM usage information for each generation for $|S_{cam}|$=32 and $|c|$=16 for the T40I10D100K dataset at 0.01 support

Table 1 demonstrates a few interesting characteristics. In this database, the maximum number of unique elements in the set of 16 candidates is generally below the hard limit of 32. However, the number of times a the number required CAM elements is above the hard limit is very rare. In the 11 generations computed, it only occurs in generations 5, 7, and 10. And in these generations, the number of occurrences is 1, 2, and 1, respectively. For these situations, the solutions is simple: stop adding candidates to the block when the maximum number of elements has been reached. The right-most column gives the maximum number of candidates possible when the CAM has been exceeded. Again, this is a rare occurrence, handled by a simple reduction in the number of candidates contained within a block.

This empirical analysis of the behavior of the bitmapped CAM architecture is extended in Tables 5 and 6. Overall, the vast preponderance of candidate sets fit easily into the 31 CAM entries per 16 candidate template.

6 Results

The architecture as implemented in VHDL is highly pipelined, with the CAM, encoders, and datapath requiring roughly 10 cycles per CAM stage. The units are connected end-to-end in a linear array for higher frequency performance, and are parameterized for item code size, block dimensions, and number of blocks. We found that the maximum clock rate was roughly 120 MHz, so the SRC restriction of 100 MHz did not constrain performance by a large factor.

The bitmapped CAM results are compared against the fastest software results available running benchmark databases in standardized formats. The results are based on a run of the T40I10D100K (15 MB) and T10I4D100k (4 MB) datasets [1] given various support levels. The hardware implementation is compared against the APRIORI-BRAVE, Borgelt, and Goethals implementation results provided in [6]. These results are based on program execution on a 2.8 GHz dual Xeon processor machine with 3GByte RAM.

The synthesis tool for the VHDL designs within Carte version 2.1 is Synplify Pro version 8.1, and ISE tools version 7.1.2 is Synplicity Synplify Pro 8.1 and the place-and-route tool is Xilinx ISE 7.1.2. The target device is the Virtex II XC2V6000 with -4 speed grade. The results are based on the placed-and-routed design. For the VHDL results, only the systolic array is implemented as the controller requires very little bandwidth and only moderate interaction with the array and should not significantly affect the performance results.

The linear array of bitmapped CAM units described in the Section 5 with 70 individual units of 16 candidates and 31 CAM entries per unit maps to 29,875 of 33,792 slices on the master XC2V6000 device. For symmetry, 70 are placed on the second FPGA device. The design place-and-routes to 8.3 ns, or 120 MHz, ideal for implementation on the SRC platform [12]. In terms of unit size, the architecture requires about one-half of the logic cells per candidate compared to the earlier systolic array architecture [5].

Figures 7 and 8 give comparisons to three state-of-the-art microprocessor-based implementations. The T40I10D100K (15 MB) and T10I4D100k (4 MB) datasets have been utilized. 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 be seen in the differences in overall timings between Figure 7 and Figure 8. The results in Figure 8 demonstrate that the T40I10D100K database has longer average timings due to the increased number of correlations between items to process.

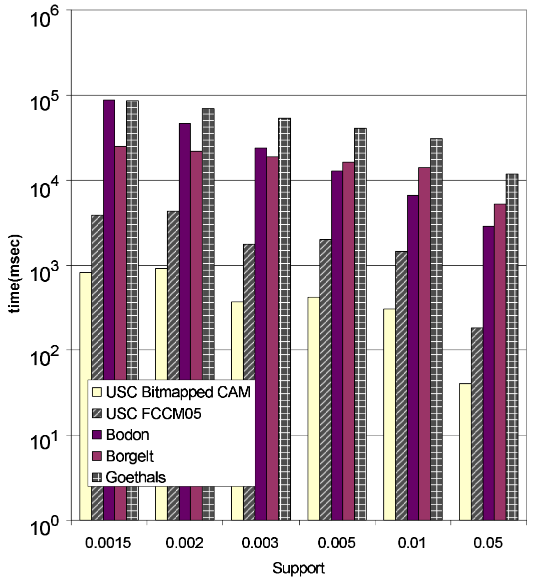

Figure 7: Performance comparison against various microprocessor-based implementations and our FCCM05 architecture [5] for the T10I4D100K dataset. Note that the time is in log(msec).

7 Conclusion

In conclusion, a novel hardware architecture for data mining operations was developed. The work is based on "bitmapped CAM" architecture that allows the similarities between candidates to be utilized. While the hardware was developed for use in the data mining field, it is useful in any fields where large numbers of subset operations are required and where the elements of the subset under search are similar. Overall, an FPGA implementations of the Apriori algorithm can provide significant performance improvement over software-based approaches.

The ideal architecture, which we plan to implement in the future, would take advantage of the fast candidate generation and pruning in our systolic array architecture developed in [5] and reconfigure the FPGA for the support phase of operation. Because the architectures are memory based and do not require changes to the hardware con-

Support	0.0015	0.002	0.003	0.005	0.01	0.05
Mean CAM Entries Needed	16.89	16.99	17.1	17	17.01	15.8
Num Exceeding Available CAMs	75	37	12	1	0	0
Mean Num . Candidates When Exceeded	12.4	12.5	12.5	15	0	0
Total CAM fills	18281	16885	14639	10106	4436	56

Figure 5: Measurement of the mean CAM entries needed for a 16 candidate block, the number of occurrences of an overflowed block, the mean number of candidates when a block overflowed, and the total number of block required for the T10I4D100K dataset

Support	0.002	0.003	0.005	0.01	0.05
Mean CAM Entries Needed	18.02	18.01	15.98	16.6	17.06
Num Exceeding Available CAMs	0	0	223	8	0
Mean Num. Candidates When Exceeded	0	0	11	12.25	0
Total CAM fills	372396	314517	216382	36658	2879

Figure 6: Measurement of the mean CAM entries needed for a 16 candidate block, the number of occurrences of an overflowed block, the mean number of candidates when a block overflowed, and the total number of blocks required for the T40I10D100K dataset

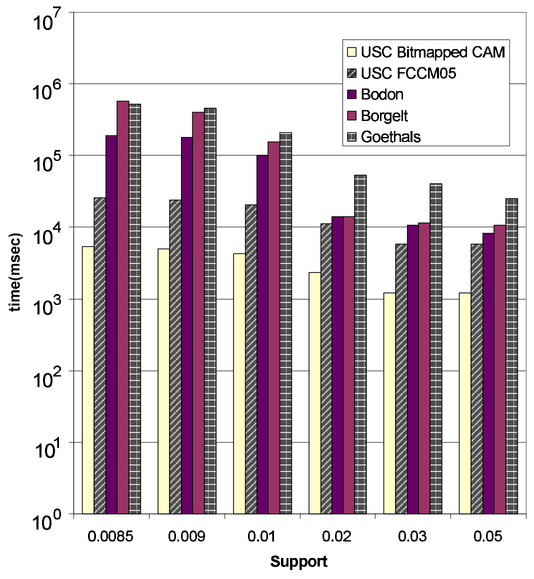

Figure 8: Performance comparison against various microprocessor-based implementations and our FCCM05 architecture [5] for the T40I10D100 dataset. Note that the time is in log(msec).

figuration bitstream, they can be swapped in and out of the FPGA with only the cost of swapping configurations. The overall execution time of the data mining algorithms is sufficiently lengthy that the cost of configuration is insignificant.

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 International Conference on Very Large Databases*, 1994.

[3] Altera, Inc. `http://www.altera.com`.

[4] D.A. Bader, B.M.E. Moret, T. Warnow, S.K. Wyman, and M. Yan. High-performance algorithm engineering for gene-order phylogenies. In *Proceedings of the DIMACS Workshop on Whole Genome Comparison*, 2001.

[5] Z. K. Baker and V. K. Prasanna. Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs. In *Proceedings of the Thirteenth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2005 (FCCM '05)*, 2005.

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

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

[8] Cray Inc. XD-1. `http://www.cray.com/products/xd1/`.

[9] 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.

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

[11] A. Kalyanaraman, S. Kothari, V. Brendel, and S. Aluru. Efficient clustering of large est data sets on parallel computers. *Nucleic Acids Research*, 31(11), 2003.

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

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

[14] Xilinx, Inc. `http://www.xilinx.com`.

[15] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335, New York, NY, USA, 2003. ACM Press.

[16] 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.