Optimization of Frequent Itemset Mining on Multiple-Core Processor

Li Liu[2, 1], Eric Li[1], Yimin Zhang[1], Zhizhong Tang[2*]
Intel China Research Center, Beijing 100080, China[1]
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China[2]
{li.a.liu, eric.q.li, yimin.zhang }@intel.com, tzz-dcs@tsinghua.edu.cn

ABSTRACT

Multi-core processors are proliferated across different domains in recent years. In this paper, we study the performance of frequent pattern mining on a modern multi-core machine. A detailed study shows that, even with the best implementation, current FP-tree based algorithms still under-utilize a multi-core system due to poor data locality and insufficient parallelism expression. We propose two techniques: a cache-conscious FP-array (frequent pattern array) and a lock-free dataset tiling parallelization mechanism to address this problem. The FP-array efficiently improves the data locality performance, and makes use of the benefits from hardware and software prefetching. The result yields an overall 4.0 speedup compared with the state-of-the-art implementation. Furthermore, to unlock the power of multi-core processor, a lock-free parallelization approach is proposed to restructure the FP-tree building algorithm. It not only eliminates the locks in building a single FP-tree with fine-grained threads, but also improves the temporal data locality performance. To summarize, with the proposed cache-conscious FP-array and lock-free parallelization enhancements, the overall FP-tree algorithm achieves a 24 fold speedup on an 8-core machine. Finally, we believe the presented techniques can be applied to other data mining tasks as well with the prevalence of multi-core processor.

1. INTRODUCTION

Frequent Itemset Mining (FIM) is one of the fundamental problems in data mining, which aims to discover groups of items or values that co-occur frequently in a dataset. It plays an increasingly important role in a series of data mining tasks, such as associations [2], correlations [4], causality [24], sequential patterns [3], episodes [17], partial periodicity [14], and emerging patterns [8]. Many FIM Implementation (FIMI) algorithms have been proposed in the literature [5,7,9,12,13,15,19,22,23,25,27,28], where the frequent pattern tree (FP-tree) [7,11,13,15,22,28] is most widely used and considered as the fastest known algorithm.

In the same time frame, the processor speed almost doubles every two years for a couple of decades. However, the DRAM speed has not kept up. The widening gap between processor and DRAM speed becomes increasingly critical to the application performance. On the other hand, since the power and thermal constraints increase with frequency, chip multi-processor (CMP) is now in the way of the microprocessor design as a means achieving higher performance with diminishing returns from increasing clock frequency. CMP has multiple cores integrated in the same chip, which significantly boosts the performance along with the increasing of thread number. However, given the memory intensive feature of FIMI algorithms and in-sufficient shared memory parallelism exploration, it is very likely even the most efficient FIMI algorithms grossly under-utilize a modern CMP processor in terms of CPU utilization.

For purpose of this study, we measure the cache and scalability performance of the fastest known FIMI algorithm, FPGrowth [15] and its parallel implementation [7] on an 8-core SMP system. The experimental setup and detailed datasets can be found in the experiment evaluation section. Otherwise specified, the following analysis presents the average performance data measured over various runs.

In FPGrowth algorithm, there are two successive phases: FP-tree building and FP-growth (mining on the existing FP-tree). FP-growth spends 80% to 95% of the total execution time. Table 1 shows the performance characteristics of FP-growth in terms of CPI (clock cycles per instruction) and cache misses. FP-growth experiences a very high L3 cache miss rate, over 1% of all data accesses missed in L3 cache are sent to main memory, which is primary performance bottleneck on a modern processor. The CPI is also far diverged from the optical performance the Pentium-4 processor can provide. The problem can be further exacerbated on a multi-core processor since frequent off-chip memory accesses will cause bus contention and limit the performance of thread level parallelization.

Conventionally FP-tree building is largely ignored due to its small execution time compared to FP-growth. However, Amdahl's law indicates that the overall scaling performance is limited by the application's serial time. Figure 1 depicts the scaling performance of FPGrowth. Though FP-growth itself achieves a 5.3 speedup with 8 threads, the whole application only obtains a less than 3 speedup when we consider FP-growth and FP-tree building together.

* The work was supported in part by National Natural Science Foundation of China under Grant No. 60573100

Table 1. Cache performance of FP-growth

Dataset	CPI	L3 miss rate (%)	L3 misses per 1000 instr.
Kosarak	5.74	15.9	6.5
Accidents	5.78	15.3	6.9
Smallwebdocs	4.80	14.4	6.0
Bigwebdocs	12.30	33.8	19.3
Webdocs	13.24	33.6	20.5

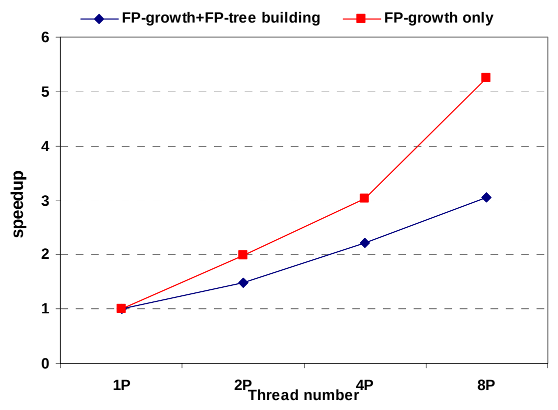

Figure 1. Scalability performance of FPGrowth

The experiments serve to illustrate an important point. The novel architectural designs cannot be directly translated into improved performance, which needs the program designer to understand and make use of these architecture features to improve the execution time. Furthermore, it becomes even demanding with the prevalence of CMP. The trend indicates that the number of cores in one processor will continue to grow according to Moore's law. In order to harness its power, the programmer should redesign the algorithm to exploit the thread level parallelism on top of multi-core processor.

In response to the performance bottlenecks of FPGrowth on the modern CMP machine, we present several techniques to alleviate these problems. Our main contributions are:

1) First, we improve the cache performance through the design of a cache-conscious FP-array. It not only reduces the cache misses for single-core processor, but also alleviates the off-chip memory accesses and improves the scalability on the multi-core processor. In addition, through the design of the cache-conscious FP-array, one can efficiently hide the cache miss latency by leveraging hardware features like hardware prefetching [10] and software prefetching [6].

2) Second, we propose a lock-free approach to parallelize the FP-tree building phase, where dataset tiling and hot sub-tree are used to improve the cache performance and provide a lock-free mechanism in the FP-tree building phase. Essentially, the thread level decomposition of the algorithm boosts the performance on the multi-core processor.

As a result of the cache-conscious and parallel-oriented optimizations, we achieve a cumulative 4.0 and 24 speedup with respect to FPGrowth for sequential and parallel program respectively.

The remainder of the paper is organized as follows. Section 2 introduces the related works. Section 3 develops a cache-conscious FP-array to take advantage of the advancements of modern architecture. Section 4 presents a lock-free mechanism to improve the scaling performance of FIMI on the multi-core processor. Detailed experimental evaluations are given in Section 5. Section 6 summarizes our study and points out some future research directions.

2. RELATED WORKS

Frequent itemset mining plays an important role in a number of data mining tasks. Examples include data analysis of market data, protein sequences, web logs, text, music, stock market, etc. Among the FIMI algorithms, Apriori [1,2] is the first efficient algorithm to solve this problem. It is based on the anti-monotone Apriori heuristic: if any length k pattern is not frequent in the database, its length (k +1) super-pattern can never be frequent. The essential idea is to iteratively generate the set of candidate patterns of length (k + 1) from the set of frequent patterns of length k (for k > 1), and check their corresponding occurrence frequencies in the database.

Following Apriori, several other FIMI algorithms, such as DHP [21], DIC [4], Eclat [27] and Partition [23], were proposed, but these algorithms are I/O inefficient and suffer from multi-scan problem. Salvatore et al. proposed Direct Count & Intersect (kDCI) [19] and (parDCI[18]), but it requires at least 3 full dataset scans. Han et al. presented FPGrowth, which converts the dataset into FP-tree. This structure is significantly smaller than the original dataset. Since it does not have an explicit candidate generation phase and generates frequent itemsets using FP-tree projections recursively, FPGrowth is much faster than any of the apriori-like algorithms. However, the pointer-based nature of the FP-tree requires costly dereferences and is not cache friendly, which prevents it from achieving satisfying performance on a modern processor. In order to improve its cache performance, Ghoting et al [11] proposed a tile-able cache-conscious FP-tree (CC-tree) to accommodate fast bottom-up traversals in FP-tree. The original FP-tree, after constructed, is transformed into CC-tree by allocating the nodes in sequential memory space in depth-first order. CC-tree yields better cache performance than FPGrowth, however, due to the tree structure of the CC-tree, it still experiences cache misses when traversing the CC-tree in a bottom-up manner. Racz presented nonordfp [22] as yet another workaround to improve cache performance, which implements the FP-growth without rebuilding the projected FP-tree recursively. The memory consumption is unaffordable and it often fails with some small dataset, which limits its wide use in practice.

In spite of the significance of the frequent pattern mining, few advances have been made on parallelizing frequent pattern mining algorithms. Most of the existing work on parallelizing association rule mining was based on apriori-like algorithms. Osmar et al. proposed a multi-tree algorithm [26], where each thread builds its own FP-tree from a certain part of the dataset and calculates the candidate pattern base from its private FP-Tree and then merges the candidate pattern bases together. It can achieve good scaling performance, but at the expense of memory expansion and redundant node traversal with the increasing of thread number. Iko et al. proposed a remerging algorithm [20] which addresses the node expansion problem on a PC cluster. However, tree merging itself is much expensive due to additional overhead. In addition to multiple-tree approach, R.Jin, et al proved that [16], lock-based single-tree approach had poor scalability. Chen et al [7] presented a tree partition approach, where a unique tree is built. Though it partially removes lock contentions, it is not essentially a lock-free approach and does not exhibit good scaling performance on a multi-core processor.

As our optimizations are presented in the context of the FPGrowth algorithm, we will describe the FP-tree data structure and the FPGrowth algorithm in more details. In order to compare with the state-of-the-art, we also briefly illustrate CC-tree in this section.

2.1 FP-tree and FPGrowth

An FP-tree is a projected dataset, which provides a compact representation for the original dataset. Each node of the tree stores an item label and a count, with the count representing the number of transactions which contain all the items in the path from the root node to the current node. The design of FP-tree is based on the following observations:

- For a dataset, only frequent 1-items are necessary to be kept, while other items can be pruned away.
- If multiple transactions share an identical frequent item set, they can be merged into one with the number of occurrences registered as *count*.
- If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the *count* is registered properly.

With these observations, an FP-tree is constructed as follows. We first scan the dataset to count the frequency of 1-items. For each transaction, we insert its frequent items into an FP-tree in frequency descending order. A new node is generated when the node with the appropriate item label is not found; otherwise, we increase the count of the existing nodes.

Table 2. A dataset with min-support = 3

No.	Transaction	Sorted Transactions
1	F,E,B,A	A,B,E
2	F,C,D,A	A,C,D
3	A,C,E	A,C,E
4	A,D	A,D
5	B,A	A,B
6	B	B
7	E,C	C,E
9	D	D

Since an FP-tree is the projected dataset of a frequent k-itemset, the union of the k-itemset and any item in this FP-tree is a frequent (k+1)-itemset. Specifying α is an item in this FP-tree, the projected FP-tree for α is constructed from the conditional pattern base of α. Each transaction in the transaction pattern base is an item sequence in the bottom-up path starting from the node associated with item α in the FP-tree. Table 2 lists a sample dataset, and Figure 3 shows the corresponding FP-tree. Each node in the FP-tree consists of five members: item label, count, parent pointer, nodelink pointer and child pointers. The nodelink pointer points to the next item in the FP-tree with the same item-id, and child pointers records a list of pointers to all its children. A header table is used to store the pointers to the first occurrence of each item in the FP-tree. A path in the FP-tree represents a set of transactions that contain a particular frequent item pattern. For example, in Figure 3, the path "root->C->E" represents all the transactions that contain item "C" and "E".

```
Algorithm: FPGrowth
Input: A prefix tree D, min-support minsupp
Output: Set of all frequent patterns
Phase 1: Construct an FP-tree from a database
(1) Scan the transaction database D once, gathering frequency of all items.
(2) Sort the items based on their frequency in descending order.
(3) Create a root node, labeled null.
(4) Scan the database a second time: for each transaction, remove items
with frequency < minsupp, sort this transaction, and append it to the root of
the tree. Each inserted node is linked to a header list of the frequent one
item with that label.
Phase 2: Mine the FP-tree by calling FP-growth()
FP-growth(tree, suffix)
For each item α in the header table of FP-tree
  (1) Output α U suffix as frequent
  (2) Use the header list for α to find all frequent items in the conditional
pattern base C for α
  (3) If there is no frequent item in C, end this loop iteration
  (4) If there is only one frequent item in C, output this item U α U suffix as
frequent, and end this loop iteration
  (5) Generate an FP-tree τ according to C and header list of α
  (6) If τ has only one path, output any sub set of items in this path U α U
suffix as frequent, and end this loop iteration
  (7) FP-growth(τ, α U suffix).
```

Figure 2. FPGrowth algorithm

The FPGrowth algorithm is presented in Figure 2. As described earlier, FPGrowth is an FP-tree based approach to frequent pattern mining. In phase 1, it builds an FP-tree from a transaction database, removing all the infrequent items. Phase 2 iterates through each item in the FP-tree. It finds all the frequent items in the conditional pattern base for an item first, and then builds a new FP-tree for this conditional pattern base when it has at least two frequent items. Thus, the conditional pattern base is scanned twice in each iteration. In general, an FP-tree node will be accessed many times since the condition pattern bases of all items share the same FP-tree. For each new FP-tree, the algorithm proceeds recursively.

2.2 CC-tree

The cache-conscious FP-tree is a modified prefix tree which accommodates fast bottom-up tree traversals. It still uses the FP-growth algorithm in FPGrowth. CC-tree allocates tree nodes in contiguous memory space following the depth-first order of FP-tree. Each CC-tree node has only 2 fields: item label and parent pointer. The new data structure is more compact and the node size is smaller than the original one. Hence, it has a better cache line utilization. In order to improve temporal data reuse, a tiling method is proposed to tile the CC-tree into several partitioned sub-trees, where the conditional pattern base for all the items

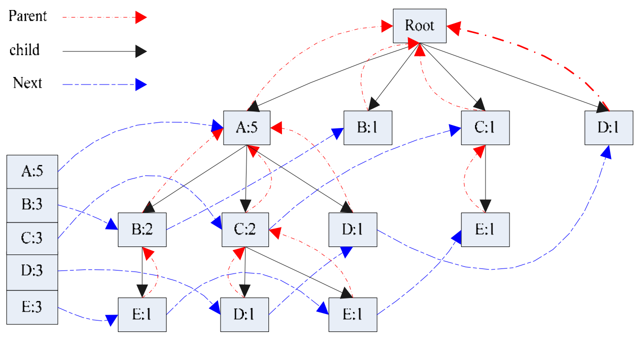

Figure 3. An example of FP-tree

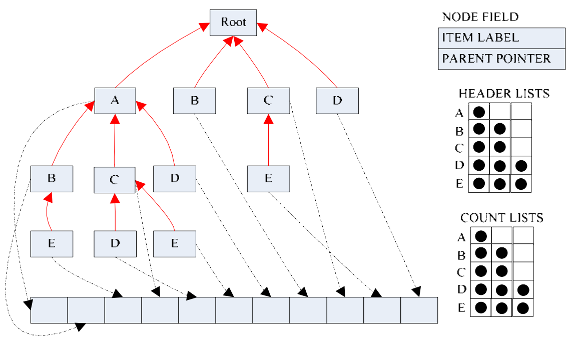

Figure 4. An example of CC-tree

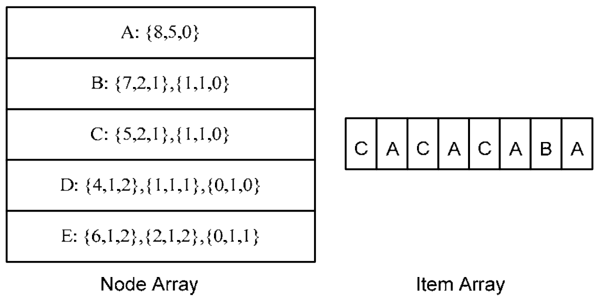

Figure 5. FP-array after FP-tree transformation

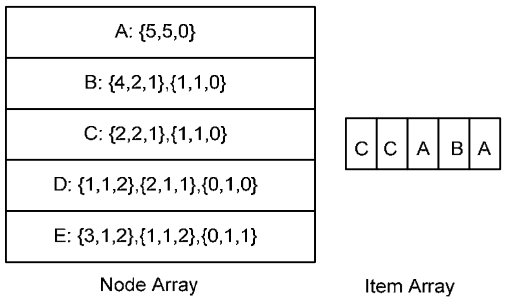

Figure 6. Compact FP-array after item elimination

within the current tile is updated when accessing a tile in the CC-tree. Figure 4 shows the CC-tree example corresponding to the FP-tree in Figure 3.

Besides FP-tree data structure reorganization, CC-tree also employs a new parallelization strategy to allow two threads work in the same tile simultaneously in order to improve data reuse performance.

3. CACHE-CONSCIOUS OPTIMIZATION

In this section, we present several novel techniques to improve the performance of frequent pattern mining using FP-tree. The details of our optimizations are presented in the context of the FPGrowth algorithm. The optimization techniques can also be applied to most frequent pattern mining algorithms that use FP-tree.

3.1 Cache-Conscious FP-array

Before we detail the optimization techniques, we first profile the FPGrowth with VTune[1]. This tool profiles program execution at source code level and provides performance characteristics to guide optimization. The FP-tree construction procedure spends 5%~20% of total execution time. In FP-growth, approximately 62% of the execution time is spent in the condition_pattern_base() routine, which finds the frequent items in the conditional pattern base for an item, and another 32% of the execution time is spent in the FPGrowth() routine, to use the results of this step to create a new projected FP-tree for the next step in the recursion. The work flow of these two procedures is very similar. They both have very poor cache utilization, mainly for the following reasons.

First, the routine that scans the conditional pattern base performs a bottom-up traversal of the FP-tree. Similarly, this access pattern also applies for the routine that builds the projected FP-tree for the next step in the recursion. In the FP-tree, each node has a total 5 elements: a list of child pointers, a parent pointer, a nodelink pointer, a count, and an item label. Except for item and parent pointer, all the other fields in the FP-tree node are not required for the two main routines in the FP-tree traversal. Consequently, once we fetch an FP-tree node, only two fields are actually used. This significantly degrades cache line utilization. Second, a node and its associated child nodes may not reside in the same cache line due to the way an FP-tree is constructed. The FP-tree is built as the dataset is scanned, and thus, successive accesses in the bottom-up traversal of the tree are not contiguous in memory. Third, the pointer based data structure in nodelink prevents two nodes with same item-id from presenting at an adjacent position. The next node with the same item-id is not likely to be present in any other cache line due to the lack of temporal locality.

We present the cache-conscious frequent pattern array (FP-array), a data structure designed to significantly improve cache performance. A cache-conscious FP-array is a data reorganization of FP-tree by transforming it into two arrays, i.e. item array and node array, which allocates in contiguous memory space. There are no pointers in the FP-array, and thus, the pointer based tree data structure is eliminated after the transformation. Given an FP-tree, we first allocate the item and node array in main memory. Then we traverse the FP-tree in depth-first order, and copy the item in each node to the item array sequentially. The item array works

[1] http://www3.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

essentially as a replication of the FP-tree. When encountering a joint node, we replicate the joint path in the item array. The node array, organized as an array list, records the occurrences of the frequent items in the item array. Each list in the node array is associated with one frequent item, and each element in the node array corresponds to an FP-tree node, which has three members: begin position of the item in item array, reference count, and transaction size. Therefore, the count, nodelink pointer, parent pointer and child pointers in the node of FP-tree are converted and stored back in the node array. Figure 5 shows the FP-array after transforming the FP-tree in Figure 3. It uses preorder tree traversal and starts from node A (the first child of the root node), records the path from A->E in the item array and updates the corresponding node array member. This procedure iterates in depth-first order until the FP-tree is traversed. Note that the item array is inserted in reverse order in order to facilitate in-order item traversal. Take {6,1,2} in node array E as an example, 6 represents the corresponding transaction starts from the 6th position in the item array; 2 is the size of this transaction, which has two contiguous items from 6th position of the item array; 1 is the reference count for this transaction. Therefore, according to the element {6, 1, 2} in the node array E, a transaction (B, A) is constructed with reference count 1. There is some redundancy in building the item array, for example, in Figure 3, when mining on item E, the bottom-up path B->A is traversed, when moving to item D, the same node A will also be accessed. Therefore, A is replicated in the item array since both bottom-up path B->A and C->A share the same parent node A.

This new array based data structure provides significant improvements because the FP-growth algorithm accesses the item array several times sequentially, where all the accessed items are allocated continuously and a large portion of them reside in the same cache line. Second, the separation of node array and item array from FP-tree yields a more compact data size, where the node size is much smaller than the original node size in FP-tree, because we only store the item name in the item array. The other four members in the node of FP-tree, e.g. child pointers, nodelink pointers, parent pointer and counts are converted into the corresponding members in the node array, which is not along the critical path. Though the replications of joint paths increase the number of items in the item array, the total allocated memory size decreases dramatically due to the reduction of node size. Comparing with FP-tree, each element of the item array is only less than 1/5 size of the FP-tree node, which accommodates the memory expansion from node replications. Once the FP-array is created, the original FP-tree can be purged, and thus memory usage does not increase significantly.

Figure 7 shows the algorithm of transforming the FP-tree into the FP-array in depth-first order. To elaborate how it works, following gives some details in FP-tree transformation.

- For the child nodes which share the same parent node, the first child node (head of child link-list) is named as the head node, and the rest child nodes as the neighbor nodes. We use a position iterator to mark the current position in the item array. The iterator decreases by 1 after a new node is inserted in the current position of the item array.
- Item stack *S* records the node path from the current node to the root node. Unless a neighbor node is detected, the items in *S* will be copied back to item array *IA*.
- When a node is visited, a new element in the node array corresponding to this node is allocated. The item label of this node is written back to both the item array and the item stack *S*.

There is data redundancy in the item array construction in Figure 5. For example, mining D and E share the same bottom-up path C->A, likewise, mining B, C and D have the same path A. Redundant item elimination in item array can decrease the memory consumption and improve the cache performance. Figure 6 shows the compact item array and node array. Furthermore, to further optimize the FP-array data structure, we dynamically choose the node size ranging from 4 bytes to 1 byte in the item array according to the total frequent items in use. Since each element in the item array corresponds to the item label of an FP-tree node, it is not necessary to use 4-byte node size all the time in the FP-growth process. For example, 1-byte node size can represent less than 256 frequent items, similarly, 2-byte node size is enough when the number of total item labels is smaller than 65536. Since typically the number of frequent items is in a several thousand scale, 2-byte node size is often used in item array for most of datasets. In the process of FP-growth, we can also dynamically change the node size, depending on the total number of frequent items in the current iteration. When the number is under the threshold, e.g. 256 or 65536, we can use smaller data size accordingly. Narrower data size often leads to better data locality performance and smaller memory consumption. In this case, the element size in the item array is further optimized to less than 1/5 of the node in FP-tree.

```
Algorithm: Transformation of FP-tree into FP-array
Input: FP-tree T, number of items in item array L
Output: Item array IA, and node array NA

(1) Allocate sequential memory space for IA and set the iterator position of
IA to L-1
(2) For each item α in T, allocate memory space for NA[α]
(3) For each child node C of the root node in FP-tree T
    Visit(C, null, 0, IA, NA)
(4) Release memory space for T

Algorithm: Visit(N, S, 0, IA, NA)
Input: FP-tree node N, item stack S, depth D, Item array IA, and node arrays
NA
Output: None

(1) If N is a neighbor node, copy the items in S to IA
(2) Allocate an element from NA[item label of N], set its reference count,
transaction size and begin position to  node N's count, depth D, and the
iterator position of IA respectively
(3) Write the node N's item label to the current iterator position in IA
(4) For each child C of node N
        Visit(C, S U item label of N, D+1, IA, NA)
(5) If N has no child, decrease the IA's iterator position by 1
```

Figure 7. FP-array transformation algorithm

3.2 Hardware/Software Prefetching in FP-array

Although large cache hierarchies have proven to be effective in reducing the latency for the most frequently used data, it is still common for memory intensive programs to spend a lot more run time stalled on memory requests. Data prefetching has been proposed as a technique for hiding the access latency of data referencing patterns that defeat caching strategies. Rather than waiting for a cache miss to initiate a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. It is an effective mechanism to significantly improve overall program execution time by overlapping computation with memory accesses. The frequent pattern mining is an essential memory intensive application, which does not have a significant amount of computation when accessing each node. To alleviate this problem, we can use data prefetching to mask the cache miss latencies.

There are two data prefetching mechanisms, i.e. hardware and software prefetching. Software prefetching initiates a data prefetch instruction issued by the processor, which specifies the address of a data word to be brought into the cache. When the fetch instruction is executed, this address is simply passed on to the memory system without forcing the processor to wait for a response. In contrast, hardware prefetching employs special hardware which monitors the processor in an attempt to infer prefetching opportunities, which records memory access patterns of the executing application and prefetches data addresses on a best-effort basis. The Intel Pentium-4 processor has a hardware prefetcher that operates without user intervention. Simple patterns such as sequential and stride memory accesses are easily recognized.

```
Algorithm: AccessItemNodeArray
Input: Node arrays A, N: number of elements in A, Item array I
Output: None
(1) For k = 0 to N-1, step 1
(2)     Prefetch(A[k+1]->begin);    // Software Prefetching
(3)     Node=A[k];
(4)     Begin=Node->begin;
(5)     Count=Node->count;
(6)     Length=Node->length;
(7)     For j = 0 to Length-1, step 1
(8)         Access I[j+Begin];      // Hardware Prefetching
```

Figure 8. FP-array traversal

Figure 8 shows the algorithm of accessing the cache-conscious FP-array. The inner loop (the 7th line) accesses the items in a transaction, and the outer loop (the 1st line) locates the transactions associated with the same item. Since item array is allocated contiguously in sequential memory space, traversing the item array and node array will yield better data locality performance. In Figure 8, a transaction in the frequent pattern base in the inner loop is accessed sequentially, which is preferable for hardware prefetching to capture the sequential data access pattern and improve its cache performance. Furthermore, since different transactions belonging to the same frequent item are not located in the adjacent position in the item array, we use manual software prefetching to fetch the next adjacent transaction based on its lookup index in the node array, as shown in the 2nd line of Figure 8.

After understanding the benefits of transforming the FP-tree into the cache-friendly FP-array, we would like to further compare the FP-array with the CC-tree in Figure 4, which also orients at improving both local and temporal cache performance. CC-tree removes 3 members of the FP-tree node, and maintains a parent pointer to traverse the CC-tree in a bottom-up fashion. However, pointers based data structure prevents it from accessing all the nodes contiguously in memory space, especially when one node has multiple children nodes where only one child node can be allocated adjacently to the parent node. Furthermore, the hardware prefetching cannot mask the cache misses in pointer chasing in the nodelink pointer. In contrast, item array provides the most compact node size, 1/2 to 7/8 smaller than CC-tree, and software prefetching is employed to hide the cache misses across accessing different transactions. Although tiling CC-tree increases the temporal cache locality, the cost of tiling building and maintaining is non-trivial. Consequently, we achieve a much better performance in terms of both cache and overall execution time.

In summary, our cache-conscious FP-array transformation makes the following benefits:

- FP-array provides a much smaller node size than FP-tree and CC-tree, which improves the cache line utilization to allow for a larger fraction of the working set fit in cache.
- By converting the FP-tree into FP-array which allocates in contiguous memory space, once an item node is fetched into a cache line, the next consecutive element in the item array will likely reside in the same cache line. It reduces the cache miss rate in FP-array traversal.
- Hardware prefetching and software prefetching can be used in FP-array optimization to enable both strided and non-strided data accesses. They reduce the cache misses in both within and across transaction accesses in the item array.

4. LOCK-FREE PARALLELIZATON

Instead of focusing on faster clock speeds and more powerful single core CPUs, the trend of processor design has made a dramatic shift towards multi-core system, which also results in a paradigm shift for the development of computationally expensive data mining algorithms. Conventionally, most of the existing work on parallelizing association rule mining on shared-memory multi-processor architecture was based on apriori-like algorithms. With the prevalence of multi-core processors, it is important to exploit thread-level parallelism within applications to fully take advantage of multi core/processor processing capabilities.

In this section, we present several novel techniques to improve the parallel performance of frequent pattern mining with the proposed FP-array. According to detailed characterization, we find that the FP-tree building module has very poor cache performance, which approximately spends 10~40% of the total execution time after FP-array optimization. Since most of computations occur in the transaction appending routine, which scans the FP-tree in top-down manner and inserts the node from the root of tree. The pointer chasing data access pattern in FP-tree construction accounts for the poor cache utilization. Before we detail the lock-free parallelization mechanism, we first present some techniques to improve the serial execution time which serves as the baseline for parallel implementation.

4.1 Dataset Tiling

Typically an FP-tree does not fit in cache. It is very likely the nodes which are associated with a new transaction are already evicted out from the cache when this transaction is inserted into the FP-tree. Since two adjacent transactions probably access different portions of FP-tree, data in one portion of the FP-tree which are already loaded into the cache can not be used for other transactions. Therefore, the cache is grossly under-utilized in the FP-tree building phase, which exhibits a poor temporal data locality performance in the FP-tree. Temporal locality states that recently accessed memory locations are likely to be accessed again in the near future. Therefore, in order to make use of the FP-tree in an efficient way, it is imperative to exploit the existing temporal locality in the FP-tree building algorithm.

We accomplish this by proposing a new approach called *dataset tiling*. It reorganizes datasets so as to make them more likely access the same portion of the FP-tree, therefore, reuse the FP-tree in a temporal fashion once it is fetched in the cache. Before we detail the proposed approach, we define some terms in the dataset tiling algorithm. The frequent items are classified into two categories: *hot items* and *cold items*. Typically, we choose the top 16 frequent items as hot items, and the remaining frequent items as code items. The FP-tree node corresponding to a hot item is named as a hot node. *Class-id* of a transaction represents the bit sequence of the hot items in this transaction. The lowest bit in *class id* stands for the hot item with largest count. For example, in Table 2, after selecting 2 hot items A and B, the *class id* of transaction {A, B, E, F} and {F, C, D, A} are "11" and "01", respectively. A hot sub-tree is a subset of FP-tree which only consists of the hot nodes, and each node path of the hot sub-tree represents one *class id*.

The procedure of dataset tiling works as follows:

1) Preparation: we scan the dataset and bypass the infrequent items. For each transaction which contains the frequent items, the *class id* is computed and the new transaction is recorded as <class id, number of items, transaction item list>, where the transaction item list stores the sorted cold items in this transaction. Meanwhile, we choose 16 hot items and build a *hot sub-tree*.

2) Dataset tiling: according to the *class id* information, the new transactions with the same *class id* are sorted and merged into a tile. The tile is stored contiguously in memory space. Once the full set of tiles is created for the new transactions, the original dataset is purged from memory. Dataset tiling executes in a recursive manner until the tile size is smaller than the cache size. After dataset tiling, the transactions which can be processed in one portion of FP-tree are assembled in one tile. They have a large overlap in sharing the same portion of sub-tree in the transaction insertion stage.

3) Dataset insertion: the transactions within one tile have the same *class id*, indicating they share the same hot node path in the hot sub-tree. When inserting these transactions to the hot sub-tree, they will be appended to the tail nodes in the hot node path. Thus, all the data insertion in one tile only performs in one subset of the FP-tree. The procedure iterates until all the tiles complete transaction insertion. In the end, the hot sub-tree is transformed into the final FP-tree. In order to accelerate finding the tail hot node, we use hash index to index all the hot nodes according to their *class id* information.

Figure 9 illustrates the FP-tree building process using transactions in Table 2. We empirically choose A and B as two hot items, according to the *class id* information, the 9 transactions are partitioned into 4 tiles, e.g. the tile with *class id* "11" has two transaction {A, B, E, F} and {A, B}. The gray nodes represent the hot nodes. The hot sub-tree is connected using dashed lines. When we insert a transaction {A, C, E} into the hot sub-tree, we first finds its *class id*, use its hash index to locate the tail hot node A, and then append the item C and E to this hot node. The building procedure iterates until all the transactions are inserted into the hot sub-tree.

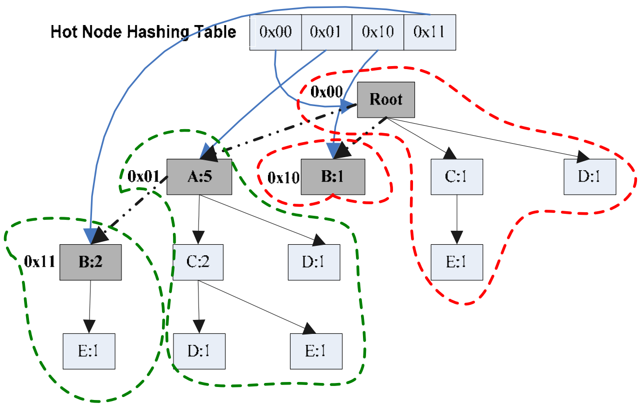

Figure 9. Lock-free FP-tree building

There are several advantages of the dataset tiling. First, it significantly improves the temporal cache locality performance by partitioning the transactions into smaller tiles, where each tile only pertains to a subset of an FP-tree, thus largely reuse the FP-tree in a temporal fashion. Second, with the infrequent and hot items pruned in the new transactions, the size of new transaction is reduced to less than 20% of original size. Coupled with their contiguous sequential data access pattern, the spatial locality cache performance is improved as well. Third, the classification of hot nodes and hot sub-tree provides a good foundation for the thread level parallelization, which will be elaborated in the next section.

4.2 Lock-free FP-tree Building

The parallelization of FP-growth is straightforward. After separating the first FP-growth loop from its recursive format, the frequent items can be assigned to the worker threads in a dynamic manner. However, parallelizing the FP-tree building is more difficult. Traditionally, the main challenge in parallelizing the building FP-tree on a shared memory system arises because of the possible race conditions when multiple processors update the same node of the FP-tree. To avoid the race conditions, generally lock based exclusive access mechanism is used to protect it from mutual access. Experiments show that this approach has poor scaling performance, e.g. the parallel program running on an 8 processor system was even slower than its sequential version [16].

Motivated by the disadvantages of existing parallel algorithms, we propose a novel lock-free approach to efficiently accelerate the FP-tree building procedure on multi-core system. In the original algorithm, the common pattern for each transaction is ignored when inserting transactions into the FP-tree, i.e. in multithreaded context, transactions in distinct threads may access the same node in the FP-tree. To resolve the conflictions among different transactions, we use the aforementioned *class id* and hot sub-tree to restructure the FP-tree building procedure into the lock-free fashion. Following shows the details of parallelized FP-tree building.

1) First, after the first scan, we get support of all items and sort the items based on their frequency. At the same time, we choose 16 hot items and build a hot sub-tree. Then we tile the dataset into a number of segments, whose amount is much larger than the total thread number. We use dynamic thread scheduling policy to assign the segments to each thread to minimize the load imbalance. Then each thread computes the *class id* for each transaction and generates a new dataset.

2) Next, the new transactions attached with the same *class id* are reordered and merged into the same tile. Each thread is responsible for one portion of transaction data. Since the start position and the size of the new transactions are determined beforehand, there is no data access conflict among the working threads.

3) Finally, the transactions are inserted to the hot sub-tree in the context of tile format. Since each tile has its unique *class id*, which does not overlap with other tiles, the transactions in each tile will be appended to one particular tail hot node in the hot sub-tree, while other tiles will not touch the same hot node pertaining to this tile. Therefore, all the tiles are essentially independent from each other. They can be inserted into the hot sub-tree non-exclusively in contrast to using locks to protect the sharing nodes in the conventional method. Since we choose 16 hot items, which leads to more than 60K tiles, in order to minimize the load imbalance, we merge some tiles together according to their computational load. Consequently, these merged tiles are assigned to each processor and inserted into the FP-tree in a lock-free manner.

Since most of the execution time is spent in the transaction insertion stage, other modules like hot sub-tree building are trivial for parallelization. Figure 9 shows the example of lock-free FP-tree building. In case of two threads, the tiles are grouped into two partitions (circled in red and green) and assigned to each thread respectively.

To summarize, we make the following contributions in the FP-tree building phase:

- Dataset tiling uses *class id* to classify the transactions into different tiles which contain the same set of frequent items. It improves the temporal data locality by reusing the same portion of FP-tree in the cache and reduces the memory consumption.
- A lock-free parallelization mechanism is proposed in the FP-tree building phase. It uses a hot sub-tree and transaction tiling to eliminate the locks in transaction insertion, which significantly improves the scaling performance on the multi-core processor.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the benefits of our optimizations and compare them with prior works. We use an 8-way Intel Xeon machine with 4GB physical memory. It has 4 dual-core processors running at 3.2GHz. Each core is equipped with a 16K L1 data cache and a 1MB unified L2 cache. A 4MB L3 unified cache is shared by two cores in one physical processor. The cache line sizes are 64 bytes for the L1 and L2 caches, and 128 bytes for the L3 cache. The system bus runs at 166MHz and delivers a bandwidth of 5.3GB/s. We use OpenMP programming model to parallelize this application. Furthermore, we use the Intel VTune performance analyzer to collect the cache performance numbers.

Table 3. FIMI Datasets

Name	Num. of trans.	Size	Min-support	Num. frequent 1-items	Aver. Effective Trans. Len.
Kosarak	990000	31M	800	1530	5.1
Accidents	340000	34M	40000	66	26.1
Smallweb-docs	230000	200M	12000	662	54.1
Bigwebdocs	500000	460M	50000	280	25.2
Webdocs	1690000	1.46G	120000	428	35.8

Table 3 shows the datasets in our evaluation. Accidents, Kosarak and Webdocs are the datasets from the Frequent Itemset Mining Implementations Repository. Webdocs is the largest dataset in the FIMI Repository containing about 1.7 million transactions. Smallwebdocs and Bigwebdocs are artificial datasets which are cut from Webdocs to represent different sizes of FIM dataset. The 4th~6th columns in Table 3 list the min-support for each dataset, the corresponding frequent 1-items number and average effective transaction length (infrequent 1-items are pruned array). Throughout this section, we compare the execution time with respect to FPGrowth from FIMI repository. In addition, to demonstrate the effectiveness of FP-array, we also include CC-tree for performance comparison. Note that hardware prefetching is enabled for both FPGrowth and CC-tree in our evaluation.

5.1 Impact of FP-array optimization

We first evaluate the FP-array optimization. From Figure 10, it is evident that we achieve a significant performance improvement due to spatial locality optimization. It obtains a speedup of 2.76 on average. Kosarak has the largest performance gain primarily due to its sparsest dataset, the cache misses of which is reduced significantly when transforming the FP-tree to the FP-array. When the hardware prefetching is enabled, it provides an additional 5%~30% speedup. Since the FPGrowth algorithm employs a tree structure where the nodes are linked through pointers, when traversing the tree in a bottom-up fashion, pointer based irregular data references will break the data spatial locality. In contrast, FP-array removes the pointer based data structure, and all the data in the FP-array are stored sequentially in memory, which directly improves the cache locality performance and facilitates hardware prefetching. Furthermore, because the item array only contains one data element, when we traverse the item array, it can fit up 16 to 64 nodes (dynamic item size from 1 to 4 bytes) in one cache line (64 bytes in Pentium-4 architecture). In the FP-tree implementation, each node spans at least 20 bytes, and at most 3 nodes can fit in a cache line. Even in CC-tree, one cache line can only keep 8 nodes. Consequently, the performance of CC-tree is better than FPGrowth, but still inferior to FP-array. Figure 12 plots the L3 cache miss reduction in terms of MPKI (misses per thousand instructions). FP-array reduces the cache misses by a factor of 17.6 on average compared to the baseline FPGrowth.

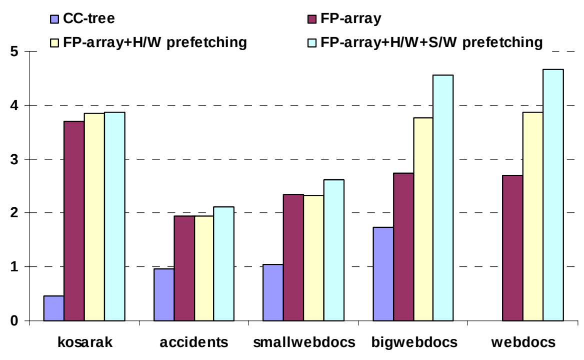

Figure 10. FP-growth sequential speedup

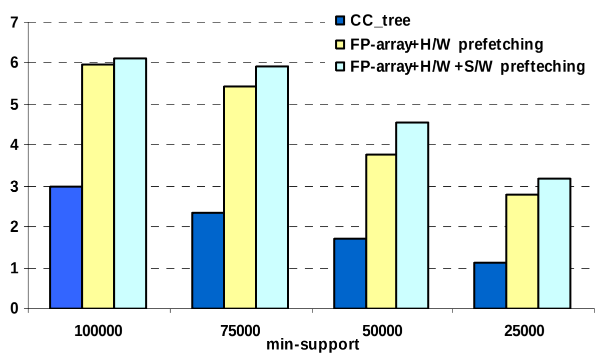

Figure 11. FP-growth sequential speedup of Bigwebdocs

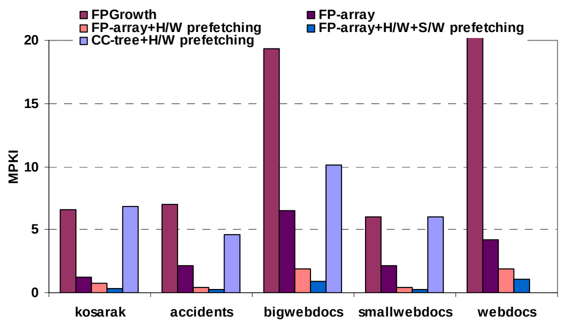

Figure 12. L3 cache miss reduction in FP-growth

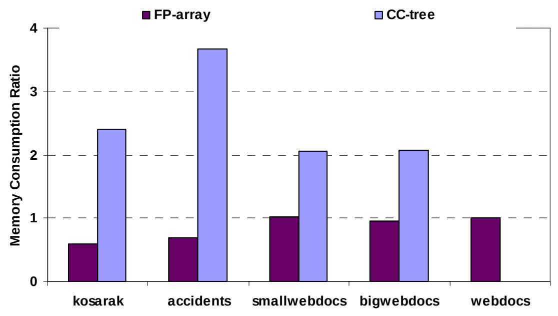

Figure 13. Memory consumption ratio in FP-growth

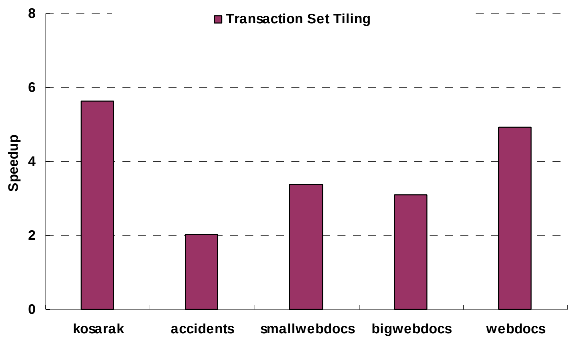

Figure 14. Sequential speedup of FP-tree building

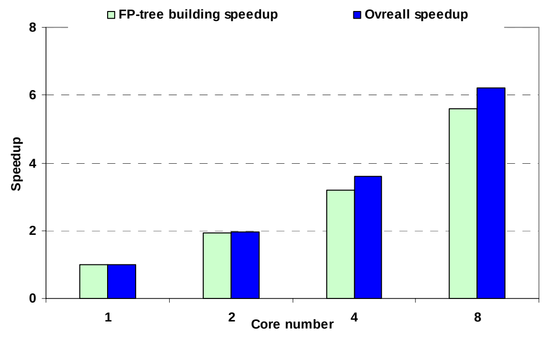

Figure 15. FP-tree building and overall scaling performance

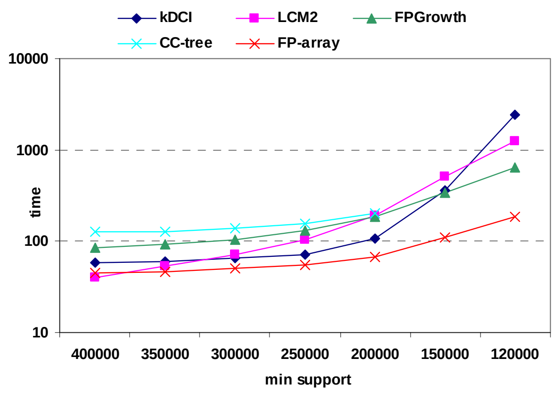

Figure 16. Total execution time of "Webdocs"

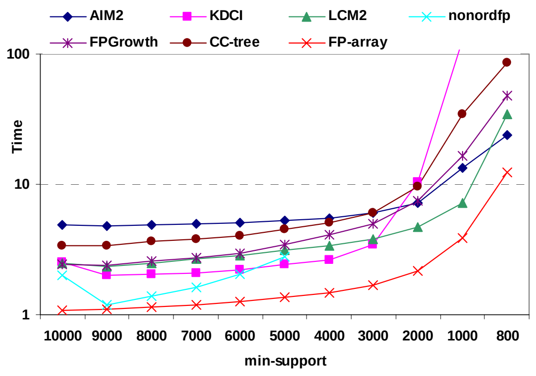

Figure 17. Total execution time of "Kosarak"

Besides hardware prefetching, software prefetching also provides 20% performance improvement for the large datasets. It pre-loads the next several transactions in the item array beforehand according to their positions in the node array. Software prefetching helps cause data at a specified memory address to be brought into cache. Therefore, it makes up for the sequential access pattern in hardware prefetching. The performance impact of software prefetching is more evident for large dataset with longer transaction length, because the longer length of a transaction makes fewer transactions installed in the cache, thus provides more opportunities for software prefetching. For instance, Bigwebdocs and Webdocs benefit most from data prefetching in Figure 10. In contrast, the speedup of Accidents is not pronounced because it holds a relative small working set. Figure 11 shows that FP-array consistently outperforms CC-tree and FP-tree with the decreasing of min-support. Lowering supports allows more infrequent items to become frequent, and thus increase the computation of FP-growth dramatically.

In addition to the performance evaluation, we also compare their memory consumption as indicated in Figure 13. FP-array roughly consumes the same memory as FPGrowth. However, CC-tree almost doubles the memory size, which practically limits it from running large dataset, e.g. Webdocs.

5.2 Impact of Dataset Tiling

Figure 14 shows the sequential execution time improvement with the dataset tiling in FP-tree building. We achieve a 4.2 speedup on average relative to the baseline - FP-tree building in FPGrowth algorithm. Dataset tiling partitions the transactions into several segments based on its *class id*. Each transaction within a tile can be inserted into a subset of the FP-tree, which in turn improves the temporal data locality performance and reduces the cache misses by a factor of 32.1 on average. In addition, dataset tiling also reduces memory consumption in building the FP-tree.

5.3 Impact of Lock Free Parallelization

The FP-tree building phase is particularly important to the multi-threading implementation, which constitutes 10~40% of sequential running time depending on the dataset and support in use. Figure 15 depicts the scaling performance of the optimized implementation, where the left bar is the lock-free FP-tree building and the right bar is the overall execution time consisting of both FP-tree building and FP-growth. With the proposed lock-free mechanism, we obtain a 5.6 speedup for the five datasets on average. The synchronization cost is trivial for the lock free parallelization. There are no locks to protect the nodes since all the tiles are explicitly partitioned and assigned to the different threads. The scalability limiting factors are largely from load imbalance and hardware resource contention.

We also compare the lock-free approach with prior single-tree based methods, e.g. tree-partition [7]. It only achieves a less than 2.0 speedup on the 8-core machine due to lock overhead and load imbalance. In contrast, even after improving the sequential time by a factor of 4, we can further achieve an additional 5.6 speedup over the optimized serial program on the 8-core system.

The overall speedup is also pretty good due to the cache-conscious optimization of FP-array. It reduces off-chip memory accesses and alleviates memory bandwidth requirement. Furthermore, FP-array is shared among multiple threads, which makes it attractive in CMP where the last level cache is shared amongst the cores to minimize data replications.

5.4 Overall Execution Time Evaluation

Finally, we compare the overall sequential execution time between our proposed FP-array (including Dataset tiling in FP-tree building) and a set of FIMIs, i.e. FPGrowth[15], CC-tree[11], AIM2[9], kDCI[19], LCM2[25], nonordfp[22], which are considered as the state-of-the-art in literature. Figure 16 and 17 present the results for Webdocs and Kosarak. Note in Figure 16, some algorithms fail to run Webdocs due to their huge memory requirement. Other datasets also show the similar trends. It is obvious that our approach achieves the lowest execution time consistently compared with the other approaches. The performance gap is more pronounced with the increasing size of input datasets, since it directly increases the working set size and incurs more cache misses when the algorithms do not hold a cache-friendly data structure.

In addition to delivering the optimal sequential execution time, we also provide an efficient parallel approach to harness the multi-threading capability provided by the latest multi-core system. The optimized algorithm achieves a 6.2 speedup on an 8-core machine. Consequently, the total execution time is reduced by a factor of 24 in combination with cache-conscious FP-array optimization compared to the sequential FPGrowth.

6. CONCLUSION

In this paper, we show that the existing frequent itemset mining implementations like FPGrowth, largely under-utilize a modern multi-core processor in terms of poor data locality performance and low thread level parallelism. We propose a cache conscious FP-array to improve the spatial data locality performance. Furthermore, it allows the algorithm to leverage hardware and software prefetching. All these optimizations reduce the cache misses greatly and deliver a cumulative 4.0 speedup compared with FPGrowth. In order to make use of the multithreading capability equipped in the multi-core processor, a dataset tiling approach is proposed to enable lock-free FP-tree building. It allows the working threads to insert transactions to the unique FP-tree without data confliction. At the same time, the temporal data locality performance is improved as well due to the reuse of FP-tree. As a result, the parallel optimization achieves a 6.2 speedup on an 8-core system. Overall, we improve the performance by a factor of 24 compared with the sequential FPGrowth. From our results, we conclude that the frequent itemset mining algorithms can be significantly optimized in two aspects:

- The design of cache-conscious FP-array coupled with hardware techniques like hardware and software prefetching to improve the cache locality performance
- Algorithm redesign to enable lock-free tree-building through dataset tiling, which improves the temporal cache performance and makes the algorithm amenable to the thread level parallelization.

We believe the proposed methodology can also be applied to other pattern mining categories as well. Further, we would like explore large scale frequent mining problems on commodity multi-core computers. They will belong to our future work.

7. ACKNOWLEDGMENTS

We would like to thank Amol Ghoting and Yen-Kuang Chen for allowing us to use their CC-tree codes.

8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large database. In *Proceedings of the International Conference on Management of Data*, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases*, 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering*, 1995.

[4] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proceedings of the International Conference on Management of Data*, 1997.

[5] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset mining algorithm for transactional databases. In *Proceedings of the International Conference on Data Engineering*, 2001.

[6] D. Callahan, K. Kennedy, A. Porterfield. Software prefetching. In *Proceedings of international conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[7] DH. Chen, CR. Lai, W Hu, WG. Chen, YM. Zhang, WM. Zheng. Tree partition based parallel frequent pattern mining on shared memory systems. In *Proceedings of IPDPS Workshop on Parallel and Distributed Scientific and Engineering*, 2006.

[8] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*,1999.

[9] A. Fiat, S. Shporer. AIM2: Improved implementation of AIM. *In Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations*, 2004.

[10] John W. C. Fu, Janak H. Patel, Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of international symposium on Microarchitecture*, 1992.

[11] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious Frequent Pattern Mining on a Modern Processor. In *Proceedings of the International Conference on Very Large Data Bases*, 2005.

[12] K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the International Conference on Data Mining*, 2001.

[13] G. Grahne, J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[14] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series dataset. In *Proceedings of the International Conference on Data Engineering*, 1999.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generations. In *Proceedings of the International Conference on Management of Data*, 2000.

[16] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE Trans. Knowl. Data Eng*, *17*,1 (JAN. 2005), 71–89.

[17] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, *1*, 3 (SEP. 1997), 259-289.

[18] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. An efficient parallel and distributed algorithm for counting frequent sets. In *VECPAR*, 2002.

[19] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, F. Silvestri. kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets. In *Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[20] I. Pramudiono and M. Kitsuregawa. Tree structure based parallel frequent pattern mining on PC cluster. In *Proceedings of the International Conference on Database and Expert System Applications*, 2003.

[21] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the International Conference on Management of Data*, 1995.

[22] B. Racz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In *Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations*, 2004.

[23] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large database. In *Proceedings of the International Conference on Very Large Data Bases*, 1995.

[24] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[25] T. Uno, M. Kiyomi, H. Arimura. LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. In *Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations*, 2004

[26] O. R. Za¨ıane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *Proceedings of the International Conference on Data Mining*, 2001.

[27] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1997.

[28] J. Zhou, J. Cieslewicz, K. Ross, M. Shah. Improving dataset performance on simultaneous multithreading processors. In *Proceedings of the International Conference on Very Large Data Bases*, 2005