Cache-conscious Frequent Pattern Mining on a Modern Processor

Amol Ghoting[1], Gregory Buehrer[1], Srinivasan Parthasarathy[1*]
Daehyun Kim[2], Anthony Nguyen[2], Yen-Kuang Chen[2], and Pradeep Dubey[2]

Department of Computer Science and Engineering[1]
The Ohio State University, Columbus, OH 43210, USA.
Corporate Technology Group[2]
Intel Corporation, Santa Clara, CA 95052, USA.
Contact emails: {ghoting, srini}@cse.ohio-state.edu

Abstract

In this paper, we examine the performance of frequent pattern mining algorithms on a modern processor. A detailed performance study reveals that even the best frequent pattern mining implementations, with highly efficient memory managers, still grossly under-utilize a modern processor. The primary performance bottlenecks are poor data locality and low instruction level parallelism (ILP). We propose a cache-conscious prefix tree to address this problem. The resulting tree improves spatial locality and also enhances the benefits from hardware cache line prefetching. Furthermore, the design of this data structure allows the use of a novel tiling strategy to improve temporal locality. The result is an overall speedup of up to 3.2 when compared with state-of-the-art implementations. We then show how these algorithms can be improved further by realizing a non-naive thread-based decomposition that targets simultaneously multi-threaded processors. A key aspect of this decomposition is to ensure cache re-use between threads that are co-scheduled at a fine granularity. This optimization affords an additional speedup of 50%, resulting in an overall speedup of up to 4.8. To

the best of our knowledge, this effort is the first to target cache-conscious data mining.

1 Introduction

Frequent pattern mining [2] is an immensely popular data mining approach which aims to discover groups of items or values that co-occur frequently in a transactional data set. Following the seminal work by Agrawal and colleagues [2], over the last decade there has been a proliferation of efficient algorithms developed for frequent pattern mining [22, 35, 31, 18, 11, 6].

During this same time frame, processor speeds have increased *40-fold* according to Moore's law. However, DRAM speeds have not kept up. Given the memory intensive nature of such algorithms, and the widening gap between memory and processor performance, it is our conjecture that these algorithms are grossly inefficient in terms of CPU utilization. Furthermore, architectural innovations such as *prefetching* and *simultaneous multi-threading (SMT)*, designed to alleviate this gap, have largely been ignored by the data mining community. We believe that techniques leveraging these architectural innovations can significantly improve performance.

To motivate this study, we measured the scaling behavior of the fastest known implementation for the frequent pattern mining algorithm, *FPGrowth* [22]. We evaluate the performance of this algorithm while we scale CPU frequency from 1300MHz to 3100MHz[1]. Ideally, one would want execution to scale linearly with processor frequency. Figure 1 shows both the ideal speedup and the observed speedup in a real experimental setting with increasing CPU frequency. While the CPU frequency increases by a factor of 2.38, the speedup for *FPGrowth* saturates at 1.6, even though cache hit rates are held constant. This is simply a result of the fact that, even though processor speeds

---

[1]This evaluation was conducted on an experimental Intel Pentium 4 system on which CPU frequency can be varied.
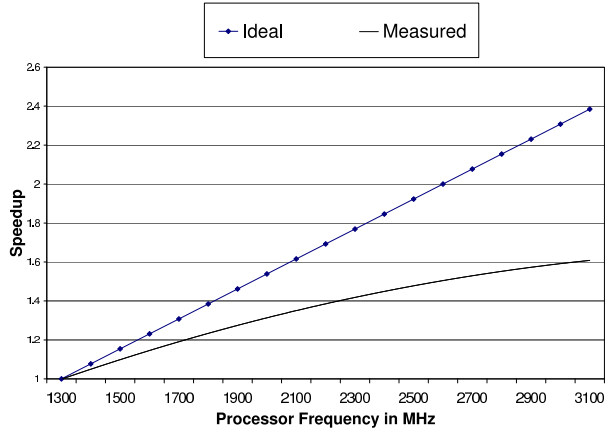
Figure 1: Scaling behavior of *FPGrowth* with increasing CPU frequency

have increased, memory stall times (measured in terms of CPU cycles) have also increased, thus limiting the performance of such algorithms.

The above experiment serves to illustrate an important point. Advanced architectural designs, even those possessing intelligent mechanisms for hiding memory latency, do not necessarily translate to improved application performance. Improving execution time will require rethinking by the program designer. In this paper, we further examine our hypothesis that even the most efficient frequent pattern mining algorithms are grossly under-utilizing a modern processor.

Specifically, we examine in depth the performance of three popular and efficient frequent pattern mining implementations from the *Frequent Itemset Mining Implementations (FIMI)* repository [6]. We evaluate implementations for the algorithms *FPGrowth* [22], *Genmax* [18], and *Apriori* [2], which apply pattern-growth, depth-first, and breadth-first search space traversal strategies, respectively. Our evaluation is performed on an Intel Pentium 4 processor using *Intel VTune Performance Analyzers*[2]. Our performance evaluation shows that these algorithms achieve a *CPU utilization of up to 8%*. Furthermore, our study reveals that this poor utilization is due to excessive number of cache misses (from the lack of data locality) and low instruction level parallelism (ILP). To the best of our knowledge, such a study is the first of its kind within the data mining community.

Faced with this performance bottleneck, we present three techniques to alleviate this problem. Our main contributions are:

- First, we improve the cache performance of these frequent pattern mining algorithms through the design and use of a *tile-able cache-conscious prefix tree.*

- Second, we demonstrate how, through the design

---

of this cache-conscious data structure, one can leverage *hardware cache line prefetching* [26], a processor technology that hides cache miss latency.

- Third, we present novel algorithms to capitalize on *simultaneous multi-threading (SMT)* [34]. Essentially, we instantiate a non-naive thread-based decomposition of the algorithms and co-schedule threads so as to maximize cache re-use and improve ILP.

Our empirical evaluation reveals that, cumulatively, these strategies result in a speedup of up to 4.8 on a modern-day uniprocessor.

2 Background and Related Work

Frequent pattern mining, also known as frequent itemset mining, plays an important role in a range of data mining tasks. Examples include mining associations [2], correlations [10], causality [33], sequential patterns [3], episodes [25], partial periodicity [21], and emerging patterns [15].

The frequent pattern mining problem was first formulated by Agrawal *et al.* [1] for association rule mining. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \cdots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \cdots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I. An itemset $i \subseteq I$ of size k is known as a k-itemset. The *support* of i is $\sum_{j=1}^{m} (1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern mining problem is to find all $i \in D$ that have *support* greater than a minimum support value, *minsupp*.

Agrawal *et al.* [2] presented *Apriori*, the first efficient algorithm to solve this problem. *Apriori* traverses the itemset search space in breadth-first order. Its efficiency stems from its use of the *anti-monotone* property: *If a size k-itemset is not frequent, then any size $(k+1)$-itemset containing it will not be frequent.* The algorithm first finds all frequent 1-items in the data set, and then iteratively finds all frequent l-itemsets using the frequent $(l-1)$-itemsets discovered previously.

This general level-wise algorithm has been extended in several different forms, leading to improvements such as *DHP* [27] and *DIC* [10]. We have proposed *Eclat* [35] and several other algorithms that use equivalence classes to partition the problem into independent subtasks. The use of the vertical data format allows for fast support counting by set intersection. The independent nature of subtasks, coupled with the use of the vertical data format, results in improved I/O efficiency, because each subtask is able to reuse data in main memory. Savasere *et al.* presented *Partition* [31], an approach that scans the data set twice; once for generating candidate frequent itemsets, and once for collecting their support. This approach processes the data set into partitions such that each partition fits in memory, improving I/O efficiency on large data sets. Han *et*

al. presented *FPGrowth* [22], an algorithm that effectively combats the above problems. *FPGrowth* summarizes the data set into a succinct prefix tree or *FP-tree*. This structure is often significantly smaller than the original data set, and thus, it can be stored in main memory in most practical scenarios. Furthermore, the algorithm does not have an explicit candidate generation phase. Rather, it generates frequent itemsets using *FP-tree* projections in main memory. The payoff is improved search space traversal and very high I/O efficiency. However, as pointed out by Goethals [16], the pointer-based nature of the *FP-tree* requires costly dereferences. We presented a hash tree-based parallel algorithm for frequent pattern mining on an SMP [28]. This article illustrates the benefits of improving memory locality in parallel algorithms. Another popular approach to frequent pattern mining is to directly find all maximal frequent itemsets, without generating all frequent itemsets in the data set. The benefit of this approach is that maximal frequent itemsets can be used to enumerate all frequent itemsets. This strategy is used in *Mafia* [11], *Maxminer* [5], and *Genmax* [18].

Several recent studies have revisited core database algorithms in an effort to improve cache performance [7, 32]. Rao and Ross [29, 30] proposed two new types of data structures: *Cache-Sensitive Search Trees* and *Cache-Sensitive B+ Trees*. This work builds upon the premise that the optimal tree node size is equal to the natural data transfer size. This corresponds to the disk page size for disk-resident databases, and cacheline size for main memory databases. Chen *et al.* [14] have further improved the index and range search performance of B+ trees using prefetching (a means of reducing materialized cache miss latency). More recently, Chen *et al.* [13] have improved the performance of Hash-Join operations using prefetching. Ailamaki *et al.* [4] examined DBMS performance on modern architectures, noting that poor cache utilization is the primary cause of extended query execution time. They conclude that database programmers must increase the attention given to data layout to improve cache performance. Lo *et al.* [24] analyzed the performance of database workloads on simultaneously multi-threaded (SMT) processors. They show that while database memory footprints tend to be large, working sets often can fit in cache (when properly organized). They determine that improved cache performance is required to leverage the abilities of multiple threads in an SMT environment. To the best of our knowledge, there has been no work in the area of cache-conscious data mining.

3 Performance Characterization

In the previous section, we presented a summary of several frequent pattern mining algorithms. In a broad sense, each algorithm is distinct in the data set representation it uses and the manner in which it traverses the itemset search space. The data set representation

that is used is horizontal [2], vertical [35], or based on a prefix tree [22]. The itemset search space traversal strategy that is used is depth-first [18], breadth-first [2], or based on the pattern-growth methodology [22]. Through several recent independent evaluations [17, 6], it is now well accepted that a prefix tree-based data set representation typically outperforms both the horizontal and the vertical data set representations for support counting. Thus, we design our frequent pattern mining workloads using a prefix tree-based data set representation, to span the three different itemset search space traversal strategies. Specifically, our workloads are prefix tree-based implementations of *FPGrowth*, *Genmax*, and *Apriori*, as representative algorithms for pattern-growth, depth-first using equivalence classes, and breadth-first-based search methodologies, respectively. For all three algorithms, we base our study on the fastest known public implementations from the FIMI repository [17, 6].

3.1 *FPGrowth*

FPGrowth [22] is a frequent pattern mining algorithm that uses an annotated prefix tree known as the *FP-tree* as a data set representation. Furthermore, it uses the pattern-growth based search methodology. In summary, the algorithm works as follows: Beginning with frequent 1-items in the data set, each k-itemset is extended with frequent items that occur in the projected data set for the k-itemset to create $(k + 1)$-itemsets. The projected data set for an itemset is the subset of the transactions in the data set that contains the itemset. This process is carried out recursively in depth-first order of the search space. Each level in the recursion uses the *FP-tree* as a data set representation. Several independent evaluations suggest that *FPGrowth* is the most efficient frequent pattern mining algorithm [17, 6] to date.

3.2 *Apriori*

Apriori [2] is a frequent pattern mining algorithm that traverses the itemset search space in a breadth-first order. Beginning at size 1, it finds frequent itemsets of size l using a data set scan, and then uses these to generate candidate frequent itemsets of size $l + 1$. In the next iteration, frequent $l + 1$ itemsets are discovered by reducing the candidate frequent $l + 1$ itemsets (using a data set scan), and the algorithm then generates candidate frequent itemsets of size $l + 2$. The process continues iteratively until all frequent itemsets are generated. The original implementation of *Apriori* uses the horizontal data layout. However, Borgelt [9], showed that the performance of *Apriori* can be significantly improved using a prefix tree. In each iteration, rather than traversing the entire data set, one can find the frequency count for the candidate itemsets by traversing the prefix tree. We use this version.

3.3 *Genmax*

Genmax [18] is a maximal frequent pattern mining algorithm that traverses the itemset search space in depth-first order. The algorithm directly enumerates all maximal frequent itemsets. The entire search task is broken down into independent subtasks using equivalence classes. Each subtask consists of a frequent itemset and a combine set, and the associated search space is traversed in depth-first order using a back-tracking search. The algorithm prunes the search space based on maximal frequent itemsets that are discovered at an earlier point in the search. Although the original *Genmax* algorithm uses the vertical data format, recently, a variant of *Genmax* that uses a prefix tree has been proposed by Grahne and Zhu [19]. The sizable improvement stems from their employment of a projected data set for frequency estimation.

3.4 Memory Managers

It has been well documented that memory allocation requests in C/C++ implementations can be a performance obstruction [20]. Each allocation is a call to the function *malloc*(), which then requires a subsequent call to the function *free*(). Both these function calls often involve expensive system calls. Consequently, each of the frequent pattern mining implementations that we analyze use custom memory managers, designed to eliminate these costs. Oftentimes, these custom memory managers allocate large chunks of contiguous memory, and then distribute portions of the memory, as needed. This buffer of memory is generated with one *malloc*(), which consequently requires only one call to *free*(). These buffers can then serve a large number of memory requests (all in user space) with very little overhead. A potential downfall to this method is that memory allocation and deallocation should be done in near first-in-last-out order. This is not a problem with frequent pattern mining, due to the recursive nature of the algorithms. We inform the reader of these managers to quell reservations regarding the performance of the implementations we have chosen to profile – they are not naive straw man implementations.

3.5 Performance Benchmarking

To analyze the chosen data mining implementations, we use a system with an Intel Xeon processor and 4GB of physical memory. The processor runs at 2GHz and has a 4-way set associative 8KB L1 data cache, an 8-way set associative 512KB unified L2 cache on chip, and 2MB L3 cache. 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 100MHz and delivers a bandwidth of 3.2GB/s. We use *Intel VTune Performance Analyzers* to collect performance numbers. This tool profiles program execution at the level of source code and provides performance characteristics for each function in the implementation. We performed a workload characterization using a variety of data sets with different

support parameters. The following analysis and findings present average performance numbers measured over various runs. The variation is not significant.

3.6 Detailed Analysis

Table 1 presents the top kernel functions for *FPGrowth*, *Genmax*, and *Apriori*. In *FPGrowth*, 61% of the execution time is spent in the *Count()-FPGrowth* routine. This routine finds the set of all viable items in the *FP-tree* (projected data set) that will be used to extend the frequent itemset at that point in the search space. 31% of the execution time is spent in the *Project-FPGrowth()* routine, which scans the *FP-tree* to build a new projected *FP-tree* for the next step in the recursion. We point out that these two routines are very similar, the difference being that the *Project-FPGrowth()* routine is not called as often as the *Count-FPGrowth()* routine. For *Genmax*, the counting routine, *Count-GM()*, scans the prefix tree to find the support of an itemset. Furthermore, in order to specify a projected data set, it maintains pointers to locations in the tree at which the previous search terminated. This eases the processing burden on subsequent counting steps. This implementation does not have an explicit projection phase, and thus, the counting routine contributes to 91% of the execution time. Finally, for *Apriori*, the counting routine, *Count-Apriori()*, scans the prefix tree to compute the frequency count for each of the candidate itemsets. *CandidateGen()* generates candidates of size $l+1$ using frequent itemsets of size l. The counting phase accounts for 70% of the execution time and 25% of the time is spent in the candidate generation phase. We will now examine the operation mix and memory access behavior of these kernels.

3.6.1 Operation Mix

Table 2 presents the operation mix[3] for the top kernel functions. The operation mix for *Project-FPGrowth()* is very similar to that of *Count-FPGrowth()*, and therefore, has not been included. There were a negligible number of floating point operations and branch mispredictions per instruction, which is why these numbers have not been included. These kernels are memory intensive, with a large number of memory operations per instruction. Moreover, most of these operations are load operations that are associated with reads on the prefix tree. This is not surprising, because these kernel functions are associated with prefix tree traversals, which are read only operations. ALU operations are primarily increment and decrement integer operations (no floating point operations). These are associated with support counting. Please note that there are a negligible number of I/O operations per instruction. This is because, all I/O operations take place during the construction of the very first prefix tree, and subsequent operations on the prefix tree are handled in main

---

[3]Please note that the operation mix need not sum to 1, as a single x86 instruction can contribute to both ALU and memory operations.

FPGrowth	Genmax	Apriori
Count-FPGrowth() - 61%	Count-GM() - 91%	Count-Apriori() - 70%
Project-FPGrowth() - 31%	Other - 9%	CandidateGen() - 25%
Other - 8%		Other - 5 %

Table 1: Kernels

	Count-FPGrowth()	Count-GM()	Count-Apriori()
Integer ALU operations per instruction	0.65	0.64	0.34
Memory operations per instruction	0.72	0.69	0.66

Table 2: Operation Mix

memory. Construction of the first prefix tree takes a negligible amount of time compared to the task of generating frequent patterns.

3.6.2 Memory Access Behavior

Table 3 presents the memory access behavior of the three algorithms. All algorithms exhibit a poor L1 hit rate, and even worse L2 and L3 hit rates. As a consequence, over 3% of all instructions miss in L3 cache, and need to access main memory. On a modern processor, such a high fraction of cache misses per instruction is a primary performance bottleneck. The Intel Xeon processor is capable of executing 3 instructions per cycle, with an optimum CPI (clock cycles per instruction) = 0.33. Here, we see a CPI value that is greater than 4, a near 12-fold slowdown from the optimum, due to poor cache utilization. The CPU is grossly under-utilized for this very same reason, with utilization in the 8-9% range across all algorithms.

3.6.3 Key Insights

Based on the provided performance characterization, together with an understanding of the kernel functions, we can make the following observations.

First, frequent pattern mining algorithms are memory intensive, and their implementations have a large number of load operations per instruction. This is because, most of the time is spent on traversing the prefix tree in search of an item, which is a memory intensive operation. These tree accesses are bottom up accesses, along the paths connecting the leaf nodes to the root. Second, the prefix tree being a pointer-based structure, prefix tree traversals result in pointer-chasing. In other words, the address of the node to be accessed next in the prefix tree is only available through a pointer at the node that is currently being accessed. Third, the prefix tree is not accessed just once, but several times. These tree accesses are largely misses, as the prefix tree is typically several times larger that the L3 cache. Fourth, when the prefix tree is created, the memory address of each node in the tree is relatively independent of the memory address of its child and parent nodes. Looking back at how the prefix tree is constructed, the transactions in the data set can appear in any order. As a result, child node and parent node addresses are relatively independent.

Based on the above observations, we conclude that prefix tree accesses exhibit poor locality. In other words, given we have accessed a certain node in the tree, its parent node, that will be accessed next, will most likely not be in the following location in the cache line. This results in very poor cache utilization, as *once a cache line for a certain node in the tree is fetched, the rest of the line is most likely going to be wasted. These algorithms do not benefit from hardware prefetching, because their memory access patterns lack structure.* For a large prefix tree that does not fit in cache, we will have a negligible amount of temporal locality. The primary bottleneck is that *prefix tree traversals leave the processor waiting on a data-cache stall for the majority of the time. This is exacerbated by the pointer-chasing problem, as such codes do not provide the processor with a large instruction pool to exploit ILP.*

4 Cache-conscious Optimizations

In this section, we present several novel techniques for improving the performance of frequent pattern mining using prefix trees. We then follow these descriptions with a performance evaluation for each technique. The details of our optimizations are presented in the context of the *FPGrowth* algorithm. We choose *FPGrowth* for our case study because it has been shown to be the most efficient frequent pattern mining algorithm to date [17, 6]. We take the time now to point out that our optimization techniques can be applied to most frequent pattern mining algorithms that use prefix trees. In section 4.7.4, we will summarize the results of using our strategies on *Genmax* and *Apriori*. Before we detail these techniques, we will introduce the reader to the prefix tree and the *FPGrowth* algorithm.

4.1 Prefix Trees

A prefix tree (or an FP-tree [22]) is a data structure that provides a compact representation of transaction data set. 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. By ordering items in a transaction, a high degree of overlap is established. The compressed nature of this representation allows in-memory frequent pattern mining, because in most practical scenarios, this structure fits in main memory. Its design is based on the following observations:

- A transaction data set representation only needs to consist of frequent 1-items in the data set; the remaining items can be pruned away. This is a direct consequence of the anti-monotone property used in *Apriori* [2].

	Count-FPGrowth()	Count-GM()	Count-Apriori()
L1 hit rate	89%	87%	86%
L2 hit rate	43%	42%	49%
L3 hit rate	39%	40%	27%
L3 misses per instruction	0.03	0.03	0.04
CPI	4	4	5
CPU utilization	9%	9%	8%

Table 3: Memory Access Behavior

No.	Transaction	Sorted Transaction with Frequent Items
1	f, a, c, d, g, i, m, p	a, c, f, m, p
2	a, b, c, f, l, m, o	a, c, f, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	a, c, f, m, p
6	a, k	a

Table 4: A Transaction Data Set with $minsupp = 3$

- If two transactions share a common prefix, as per some sorted order of the frequent items, they can be merged into one, provided a count value indicating this merge is registered. Furthermore, if frequent items in a transaction are sorted in descending order of their frequencies, there is a greater chance that more prefix strings will be shared[4].

With these observations in mind, a prefix tree is constructed as follows:

1. Scan the data set to produce a list of frequent 1-items.

2. Sort the items in frequency descending order.

3. Sort the transactions based on the order from (2).

4. Prune frequent 1-items.

5. For each transaction, insert each of its items into a tree, in sequential order, generating new nodes when a node with the appropriate label is not found, and incrementing the count of existing nodes otherwise.

Table 4 shows a sample transaction data set, and Figure 3(a) shows the corresponding prefix tree. Each node in the prefix tree consists of an *item, count, nodelink ptr,* (which points to the next item in the prefix tree with the same item-id) and *child ptrs* (a list of pointers to all its children). Pointers to the first occurrence of each item in the tree are stored in a header table.

To compute the frequency count for an itemset, say ca, using a prefix tree, we proceed as follows: First, we find each occurrence of item c in the tree using the node link pointers. Next, for each occurrence of c, we traverse the tree in a bottom up fashion in search of an occurrence of a. The count for itemset ca is then the sum of counts for each node c in the tree that has a as an ancestor.

4.2 FPGrowth Algorithm

The *FPGrowth* algorithm is presented in Figure 2. As described earlier, *FPGrowth* is a prefix tree-based approach to frequent pattern mining. In Step 1, it builds

[4]This may not be the minimal representation[22].

Algorithm: FPGrowth

Input: A prefix tree D, minimum support min_s
Output: Set of all frequent patterns

Step 1: Construct an FP-tree
(1) Scan the transaction database D once, gathering support of
(-) all items.
(2) Sort the items based on their frequency.
(3) Create a root node, labeled null.
(4) Scan the database a second time.
(5) For each transaction, remove elements with
(-) $frequency < min_s$.
(6) Sort the transaction, and append it to the root
(-) of the tree, maintaining the prefix property.
(-) Each inserted node is linked to a header list of the
(-) frequent one item with that label.

Step 2. Mine the FP-Tree by calling FP-Growth(FP-tree,null,min_s)
FP-Growth (tree, suffix, min_s)
{
(1) If tree has only one path
(2)   Output $2^{path} \cup$ suffix as frequent
(3) Else
(4)   For each frequent one item β in the header table
(5)   { Output the item $\cup$ suffix as frequent
(6)   Use the header list for β to find
(-)   all frequent items in conditional pattern
(-)   base C for β
(7)   If we find at least one frequent item in the conditional
(-)   pattern base, use the header list for β, and C
(-)   to generate conditional prefix tree τ
(8)   If $\tau \neq 0$ then
(9)   FP-Growth(τ, suffix $\cup\beta$)
(10)   }
}
Note: 2^{path} denotes the power set of the elements in path

Figure 2: FPGrowth Algorithm

a prefix tree from the transaction database, removing all infrequent items, using the procedure outlined in section 4.1. Step 2 iterates through each item β in the tree, and performs two sub-steps. First, it uses the prefix tree to find all frequent items in the conditional pattern base for the item β. This involves scanning the tree bottom up beginning at all node locations for item β. The header table provides a starting point for this search. The remaining locations for item β are derived using the node link pointers. Second, given we have discovered at least one frequent item in the conditional pattern base of item β in the tree, we build a projected database (represented as a prefix tree) for item β. This sub-step also involves scanning the conditional pattern base of item β, in search for items to be included in the projected database. For each projected database that we build, the algorithm proceeds recursively.

4.3 Spatial Locality Related Enhancements

Through the detailed characterization presented in the previous section, we concluded that approximately 60% of the execution time is spent on finding frequent items in the conditional pattern base for an item, and an additional 30% of the execution time is spent on using the results of this step to create a new projected prefix tree. Both these procedures 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 prefix tree. This access pattern also holds true for the routine that builds the projected prefix tree for the next step in the recursion. While we scan the prefix tree, we are only concerned with the *item* and *parentpointer* associated with the node. In the prefix tree proposed by Han *et al.* [22], each node has a list of child pointers, a parent pointer, a nodelink pointer, a count, and an item. Except for *item* and *parentpointer*, all other fields in the prefix tree node are not required for the two main routines. Consequently, once we fetch a prefix tree node, only two fields are actually used. This significantly degrades cache line utilization. Second, due to the way a prefix tree is constructed, the chances are that a node will not be present at an adjacent location in the cache line of its child node. This prefix tree is constructed as the data set is scanned, and thus, successive accesses in the bottom-up traversal of the tree are not contiguous in memory. Due to the lack of temporal locality, this node is not likely to be present in any other cache line. The result is commonly a cache miss.

We present the *cache-conscious prefix tree* (Figure 3b), a data structure designed to significantly improve cache performance through spatial locality. A cache-conscious prefix tree is a modified prefix tree which accommodates fast bottom up traversals and improves cache line usage. First, given a prefix tree, our solution to improve spatial locality is to reallocate the tree in main memory, such that the new tree allocation is in depth-first order of the original tree. We *malloc*() one contiguous block of memory equal to the total size of the prefix tree. Next, we traverse the tree in depth-first order, and (in one pass) copy each node to the next block of memory (in sequential order). This simple reallocation strategy provides significant improvements, because all algorithms access the prefix tree several times in a bottom up fashion, which is largely aligned with a depth first order of the tree. Second, our node size is much smaller than the original node size, because we do not include child pointers, node pointers, and counts. These data members are eliminated, because in FPGrowth, child pointers, node link pointers, and counts are not used during a bottom up traversal of the tree. This new node size is less than half of the original node size, which allows at least twice as many nodes to reside on one cache line. However, the node fields *node link pointer* and *count* are required at the start of each bottom up traversal. Therefore, these fields are stored in a separate structure, as *node link pointer* and *count* accesses are not along the critical path. We would like the reader to note that once the cache-conscious prefix tree is created, the original tree can be purged, and thus memory usage does not increase significantly. The tile-able aspect of cache-conscious prefix trees will be presented in section 4.5.1.

In summary, our allocation strategy has the following benefits:

- Prefix tree accesses being bottom up accesses, once a node is fetched into a cache line, the next consecutive node in depth-first order of the tree will likely be located in the next consecutive location in the cache line. This should reduce the cache miss rate associated with prefix tree traversals.

- The fact that we are improving cache line utilization by using a smaller prefix tree node allows for a larger fraction of the working set to fit in cache.

4.4 Prefetching

Cache line prefetching [12] is a popular technique for reducing the effect of cache line misses, particularly when applications do not perform a significant amount of computation per cache line. Frequent pattern mining is one such application. Although we improve the spatial locality of data access in the algorithms, we still spend a significant portion of the execution time waiting on cache misses. These misses are particularly difficult to mask, because the algorithm only performs simple ALU operations upon accessing each node. This can be alleviated by prefetching nodes of the tree into cache with hardware [8]. The Intel Pentium 4 has a hardware prefetcher that operates without user intervention. It records memory access patterns of the executing application and prefetches data addresses on a best-effort basis. Simple patterns such as sequential and strided memory accesses are easily recognized. Since the accesses to the cache-conscious prefix tree are largely sequential, the use of the hardware prefetcher has great benefit. Hardware prefetching outperforms software prefetching for this access pattern. Therefore, we do not consider software prefetching in the analysis to follow.

4.5 Temporal Locality Related Enhancements

Temporal locality states that recently accessed memory locations are likely to be accessed again in the near future. Cache designers assume that programs will exhibit good temporal locality, and store recently accessed data in the cache accordingly. Therefore, it is imperative that we find any existing temporal locality in the algorithm and restructure computation to exploit it.

To simplify further discussion, we present the core loop for *FPGrowth* in Figure 4. For each loop iteration, we do the following: First, we scan the conditional pattern base of each item i in the header table, for its projected frequent items. Second, if there are any frequent items in the conditional pattern base of i, we build a conditional prefix tree for item i, again, accessing the conditional pattern base of i. The prefix tree does not typically fit in cache. Even the conditional pattern base does not always fit in cache. As a result, both scans of the conditional pattern base do not reuse portions of the tree in cache. This holds between: (a) a call to the conditional pattern base routine followed by the build
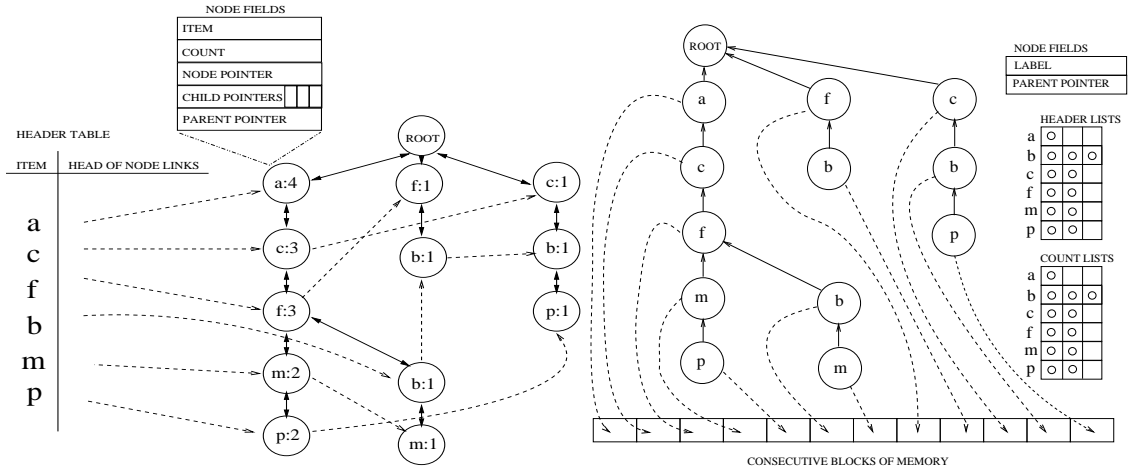
Figure 3: An FP-tree/prefix tree (a) and a Cache-conscious Prefix Tree (b)

Core loop: FPGrowth

Input: A prefix tree D, minimum support min_s

```
(1)   For each frequent item i in D
(2)   {
(3)        Find number of items j in conditional
(4)        pattern base of i with support min_s
(5)
(6)        if j > 0
(7)        {
(8)            Build conditional prefix tree P for
(9)              item i
(10)           FPGrowth (P, min_s)
(11)       }
(12)  }
```

Core loop after path tiling: FPGrowth
Input: A prefix tree D, minimum support min_s

```
(1)   For each path tile t in the cache-conscious prefix tree
(2)   {
(3)        For each frequent item i in D
(4)        {
(5)          Find counts c_i for each item in the
(-)             conditional pattern base of i
(-)             with node locations in t
(6)        }
(7)   }
(-)
(8)   Aggregate conditional pattern base counts c_i collected
(-)   across all tiles, for all items in D
(-)
(9)  For each path tile t in the cache-conscious prefix tree
(10){
(11)       For each frequent item i in D
(12)       {
(13)           j = c_i = number of items
(-)               in conditional pattern base of i
(14)           If j > 0
(15)           {
(16)               Build conditional prefix tree
(-)                 P_i for item i with node
(-)                 locations in t
(17)           }
(18)       }
(19)}
(20) For each frequent item i in D
(21) {
(22)       If conditional prefix tree P_i for item i exists
(23)           FPGrowth(P_i, min_s)
(24) }
```

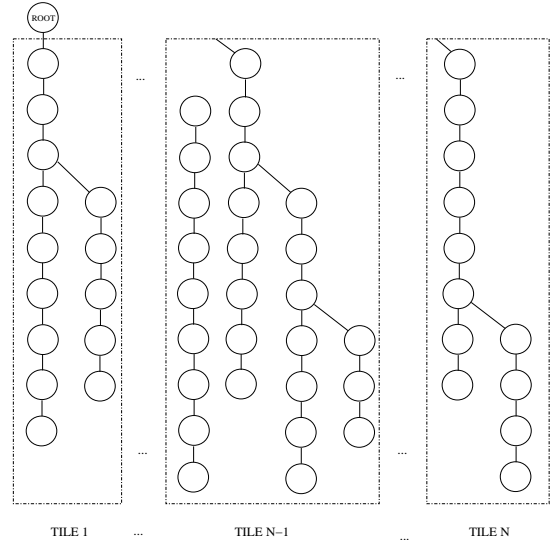Figure 4: Core Loops for FPGrowth and Path Tiled FPGrowth using Cache-conscious Prefix Trees

Figure 5: Path Tiling

conditional prefix tree routine, and (b) consecutive iterations of the loop.

4.5.1 Path Tiling

We can restructure computation within the algorithm so as to improve temporal locality. The goal of restructuring the algorithm is to maximize reuse of the prefix tree once it is fetched into cache. We accomplish this by reorganizing computation, and thus, accesses to the prefix tree, in the algorithm. Our approach, called *path tiling* works as follows.

First, we break down the tree into relatively fixed sized blocks of memory (tiles) along paths of the tree from leaf nodes to the root, as illustrated in Figure 5. This is possible because our tree is allocated in depth first order. The tiles are identified using their start and end memory addresses.

Next, we iteratively fetch each tile into cache. Then for each frequent item i, we traverse the part of its conditional pattern base that has leaf nodes located within the tile's start and end address. There is a large over-

lap between the conditional pattern bases of different items. Thus, once a tile is brought into the cache, conditional pattern base accesses for all items that hit the tile are managed in cache. This dramatically improves temporal locality for the algorithm.

Once we determine the number of frequent items in the conditional pattern bases of all items i in D, we must build the projected prefix trees for each i. As this step also accesses the conditional pattern bases as above, it can also make use of path tiling. This improves temporal locality through the conditional tree building phase, but results in increased memory usage, as we need to maintain a larger number of projected trees in main memory. There is a workaround however. We introduce an additional loop that builds a conditional prefix tree for the first k items, and not all items. These k trees are processed recursively, after which they are purged. We then continue to the next k items, and proceed similarly. We have not included this additional loop in Figure 5 for simplicity of explanation.

4.6 Improving ILP via Simultaneous Multithreading

Simultaneous Multithreading [34] (SMT) is a processor design that combines hardware multithreading with superscalar processor technology to allow multiple threads to issue instructions each cycle. SMT permits all thread contexts to simultaneously compete for and share processor resources by maintaining several thread contexts on chip. Unlike conventional superscalar processors, which suffer from a lack of per-thread instruction-level parallelism, simultaneous multithreading enables multiple threads to compensate for low single-thread ILP. The performance consequence can be significantly higher instruction throughput and program speedups for database workloads, web and scientific applications. SMT has been incorporated into the Intel Pentium 4 processor in the form of Hyper-Threading technology [23] which supports two thread contexts on chip. For a multithreaded implementation, SMT can provide several benefits.

- First, it can absorb an appreciable portion of the cache miss latency seen by a single threaded implementation by overlapping computation in one thread with a cache miss in another thread.

- Second, data fetched into the cache by one thread can be reused by the second thread. This reuse of data can take place across all cache levels, and serves as another way of reducing materialized cache miss latency.

A natural candidate for a two-thread decomposition of a frequent pattern mining algorithm is to use an extant strategy like that proposed in [28]. Such a strategy would involve decomposing execution into two independent threads of computation. However, when we evaluate this strategy for *FPGrowth* on an SMT, we

Name	Number of transactions
DS1 - T40I15D300K	300000
DS2 - T60I15D300K	300000
DS3 - T70I15D300K	300000
DS4 - T100I15D300K	300000
DS5 - Webdocs.dat	500000

Table 6: Data Sets

were not able to gain any benefit from simultaneous multi-threading. A detailed study revealed that the first benefit from above is not materialized in frequent pattern mining implementations when using this extant strategy. This is because there is insufficient computation to overlap with the memory stalls. In the future, as more hardware contexts are put on chip, it is likely that this first benefit will materialize, because we will have a larger instruction pool. For now, however, we leverage the second benefit of SMT for improving ILP in frequent pattern mining.

We devise a novel parallelization strategy in which the two threads follow each other through the same FP-Growth() calls. These threads are not independent, but rather, they operate on the same tile simultaneously. This is accomplished through fine grained parallel execution of the tiled loops that were shown in Figure 4. The workload for each tile is partitioned across the two threads. By co-scheduling the two threads, when one thread fetches a portion of the tile into the cache, it will be reused by the second thread. This results in significant cache reuse between the two threads.

4.7 Performance Evaluation Revisited

We now empirically evaluate the benefits of our optimizations. We use four synthetic data sets generated by the IBM Quest Dataset Generator and a real data set called Webdocs, as presented in Table 6. For the synthetic data sets, the naming parameters are the average transaction length T, the average maximal pattern length I, and the number of transactions D. Webdocs [17] was chosen, because most other FIMI data sets are too small. We do realize the limitations of using this synthetic data set generator [36], but truly large real data sets are not readily available. Although the synthetic data sets only have 300,000 transactions, these data sets are very dense, and the FIMI implementations we use are unable to handle a larger number of transactions. The experimental setup is identical to that provided in the previous section. Throughout this section, we compare execution time with respect to the fastest known implementation of *FPGrowth* from the FIMI repository [19]. Execution times for this implementation are summarized in Table 5 and we present speedup numbers with respect to these times. Also note that speedup is based on overall execution time, including the time required to create the first tree.

4.7.1 Benefits of Improving Spatial Locality

From Figures 6 through 8, it is evident that we achieve a significant performance improvement due to improved spatial locality. Most trials provided between 30
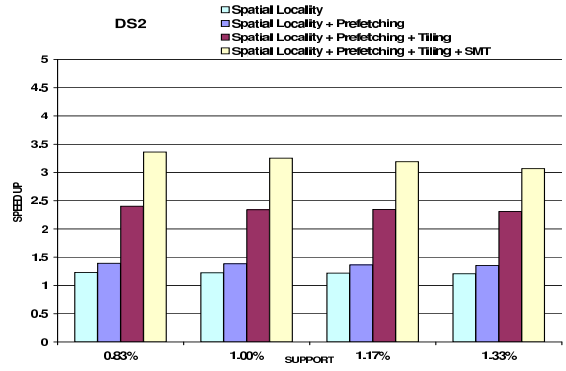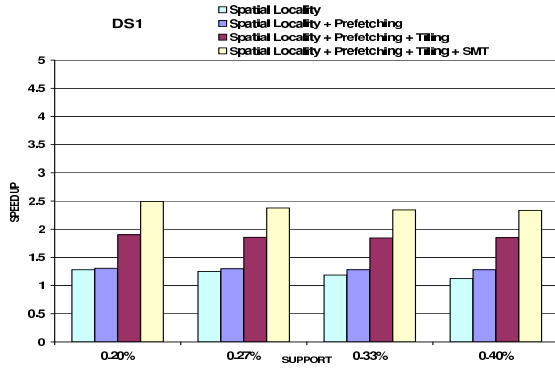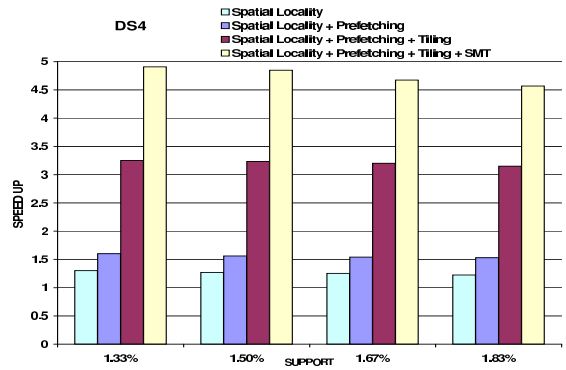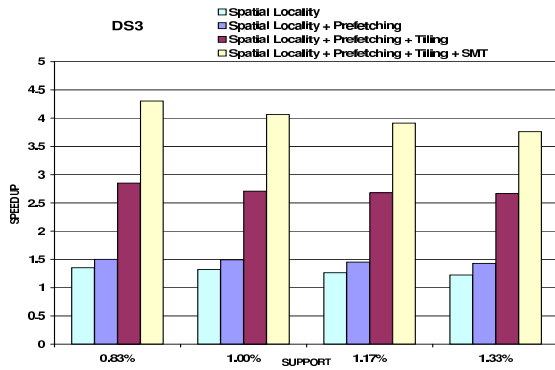
Figure 6: Speedup on DS1 and DS2
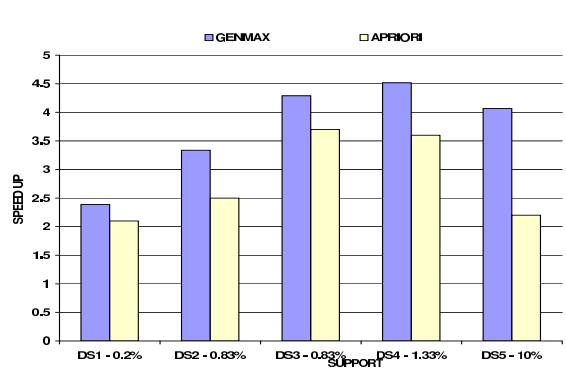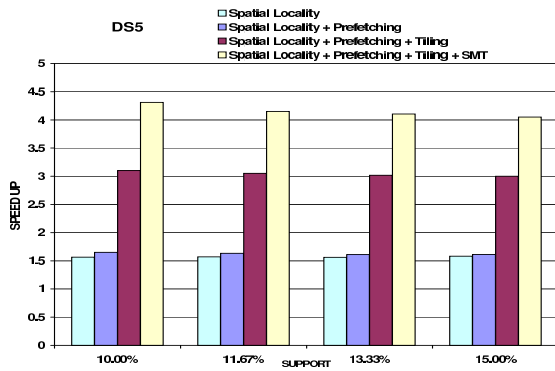

Figure 7: Speedup on DS3 and DS4

Figure 8: Speedup on DS5 and Speedup for Apriori and Genmax

	DS1 (0.20%)	DS2 (0.83%)	DS3 (0.83%)	DS4 (1.33%)	DS5 (10%)
Baseline	192 sec	269 sec	627 sec	3798 sec	949 sec
Cache-conscious	77 sec	80 sec	145 sec	773 sec	220 sec

Table 5: Execution Time Comparison

and 60% improvement. When the hardware prefetcher is enabled, there is an additional 10 to 25% speedup. The Pentium 4 processor has a 64 byte cache line size. Therefore, when we traverse the cache-conscious prefix tree, we can fit up to 8 tree nodes in one cache line. In the baseline implementation, each node spans at least 20 bytes, and at most 3 nodes would fit on a cache line. The cache-conscious tree directly improves cache utilization and also facilitates hardware prefetching, because the prefetcher can easily predict simple serial access patterns. We also note that speedup improves with increasing transaction length and decreasing support. Increasing transaction length improves the benefits of spatial locality, because the path lengths are greater. In addition, lowering supports increase the size of prefix trees, providing ample opportunity to leverage the sequential nature of the path traversals.

4.7.2   Benefits of Improving Temporal Locality

Path tiling provides a significant improvement over that provided by spatial locality and prefetching. Returning to Figures 6 through 8, we see cumulative speedups ranging from 1.9 to 3.2. Some of the benefit of increased spatial locality is tempered due to tiling, but overall, we see significant speedup. By reusing cache content, a large fraction of the misses are eliminated. It can be seen that as we lower support, the impact of tiling greatens. We attribute this to larger prefix tree sizes and greater benefits from temporal locality. [5]

4.7.3   Benefits of SMT

An extant parallelization strategy did not provide more than 3% improvement on an SMT. Therefore, the benefits we see in Figures 6 through 8 are due to the reuse of cached data between threads, and thus, improved ILP. *The use of SMT gives us an overall speedup of up to 4.8. Cumulatively, our optimizations increase L1 hit rate to 94% (from 89%) and L2 hit rate to 98% (from 43%).*

4.7.4   Benefits on Apriori and Genmax

We also injected all our proposed techniques into publicly available implementations of Genmax and Apriori [17]. Due to space constraints, we are not able to detail the optimizations for these algorithms, other than to express that the methodology was similar. Execution time improvements were comparable with that of our case study, as depicted in Figure 8. Genmax improves up to 4.5 -fold, and Apriori improves up to 3.7-fold. Apriori shows slightly less improvement due to its candidate generation phase, which does not use the prefix tree.

---

[5]Note that we cannot evaluate path tiling without the improved spatial locality; it is the depth-first ordering of the cache-conscious prefix tree that provides the possibility of path tiling.

4.8   Discussion

It is our contention that a large percentage of data mining algorithms will not glean the benefits of state-of-the-art architectures. We tested this notion with three publicly available frequent pattern mining algorithms, and showed that they greatly under-utilize the cache and are affected by poor ILP. It is thus natural to believe that many other data mining algorithms, particularly those which share a common algorithmic structure with frequent pattern mining, suffer from similar bottlenecks.

Algorithms in the areas of tree mining, sequence mining, and graph mining are particularly susceptible to such bottlenecks. Solutions to these problems spend a considerable amount of time estimating support for patterns, often rereading the same blocks of data in a streaming fashion. In all likelihood, the working set will not fit in cache due to the size of the input data. These algorithms do not necessarily use prefix trees, however, as such the community could benefit from an investigation into how their data structures can be made cache-conscious. It is imperative that this investigation include an evaluation on the use of tile-able data structures. Furthermore, the investigation must look at mechanisms to decompose the algorithm into threads of execution that have significant overlap in access. This would help improve ILP on an SMT. These are required to derive high performance from today's architectures. Technology trends indicate that future architectures will possess more thread contexts on chip, as well as more execution cores per processor. This would motivate a study that targets both inter-thread ILP for SMT and inter-thread data reuse between cores.

5   Conclusion

In this paper, by way of an extensive performance characterization, we show that frequent pattern mining algorithms grossly under-utilize a modern-day CPU. This poor utilization is attributed to poor data locality and low instruction level parallelism. We improve the performance of said algorithms through the design of a tile-able cache-conscious prefix tree.

This data structure improves spatial locality and facilitates hardware cache line prefetching. Furthermore, it allows the use of path tiling, a novel tiling strategy, to improve temporal locality. All these optimizations significantly reduce the number of cache misses. We also present a multi-threaded decomposition for frequent pattern mining algorithms, which, coupled with a thread co-scheduling strategy, significantly improves ILP as well as cache performance on simultaneously-multithreaded processors.

Our results can be summarized as follows:

- Cache-conscious prefix trees improve spatial locality in data access, and coupled with prefetching and path tiling, result in up to 3.2-fold speedup.

- An intelligent thread-based decomposition on an SMT provides a cumulative speedup up to 4.8-fold

- Overall, CPU utilization improves nearly 5-fold

From our results, we conclude that the cache performance and ILP of frequent pattern mining algorithms can be greatly improved through the use of a cache-conscious prefix tree coupled with processor technologies such as hardware cache line prefetching and SMT. We believe that this work makes an important contribution towards applying cache-conscious techniques to various data mining algorithms.

Our work assumes that the prefix tree fits in main memory. For future work, we are extending our strategies to disk-resident prefix trees. In addition, we are exploring the fruitfulness of cache-conscious strategies in other pattern mining domains, such as graph mining and sequence mining.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. 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 Databases*, 1994.

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

[4] A. Ailamaki, D. J. DeWitt, M. Hill, and D. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of 25th International Conference on Very Large Data Bases*, 1999.

[5] R. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the International Conference on Management of Data*, 1998.

[6] R. Bayardo, B. Goethals, and M. Zaki. FIMI 2004 workshop. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2004.

[7] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proceedings of the International Symposium on Foundations of Computer Science*, 2000.

[8] D. Boggs, A. Baktha, J. Hawkins, T. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 2004.

[9] C. Borgelt. Efficient implementations of apriori and eclat. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[10] 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.

[11] 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.

[12] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[13] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. In *Proceedings of the International Conference on Data Engineering*, 2004.

[14] S. Chen, P. Gibbons, and T. Mowry. Improving index performance through prefetching. In *Proceedings of the International Conference on Management of Data*, 2001.

[15] 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.

[16] B. Goethals. Memory issues in frequent itemset mining. In *Proceedings of the ACM Symposium on Applied Computing*, 2004.

[17] B. Goethals and M. Zaki. FIMI 2003 workshop. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

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

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

[20] D. Grunwald and B. Zorn. Customalloc: Efficient synthesized memory allocators. *Software Practice and Experience*, 1993.

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

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

[23] IntelCorp. *Intel Hyper-Threading Technology*, 2004.

[24] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the International Symposium on Computer Architecture*, 1998.

[25] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1997.

[26] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[27] 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.

[28] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 2001.

[29] J. Rao and K. Ross. Cache conscious indexing for decision support in main memory. In *Proceedings of the International Conference on Very Large Databases*, 1999.

[30] J. Rao and K. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of the International Conference on Management of Data*, 2000.

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

[32] A. Shatdal, C. Kant, and J. Naughton. Cache-conscious algorithms for relational query processing. In *Proceedings of the International Conference on Very Large Databases*, 1994.

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

[34] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[35] 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.

[36] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2001.