A Study on Cache Conscious Optimization for Compressed FP-Tree

Faizal KURNIAWAN[†], Kazuo GODA[††], and Masaru KITSUREGAWA[†,††]

† Graduate School of Information Science and Technology, The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
†† Institute of Industrial Science, The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan
E-mail: {faizal,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract One very important application in the data mining domain is frequent pattern mining algorithm. Various researchers have worked on improving the efficiency of this computation, mostly focusing on algorithm-level improvement. More recent work has explored on architecture specific optimization to reduce the gap between processor and memory speed. This paper studies our attempt to reduce memory access overheads for Compressed FP-Tree (CFP-Tree) mining algorithm. In this paper we present optimization of cache conscious data structure over CFP-Tree and show our intensive experiments, then qualifying the potential benefits.

Key words data mining, optimization, cache, processor

1. Introduction

Frequent Itemset Mining (FIM) is one of fundamental problems in data mining which aims to discover groups of items or values that co-occur frequently in a dataset. Following the first work by Agrawal et al. [1], over the last decades, many FIM Implementations (FIMI) have been proposed in the literature. Furthermore it also has many applications in important business scenes such as market basket analysis [1], intrusion detection [3], inferring pattern from web access log [4] and software bug detection [5].

In the same time frame, processor speeds have increased almost double every year according to the Moore's law. However DRAM speeds have not kept up. Hence this widening gap between processor and DRAM becomes increasingly critical to the application performance. Despite the fact that processor technology has developed rapidly, it is very likely even the most efficient FIMI algorithms still grossly underutilize modern processor capabilities [6].

We evaluated the performance of data mining algorithm, Compressed Frequent Pattern Tree (CFP-Tree) mining algorithm in Pentium-4 processor. As shown in table 1, CFP-Tree experiences high L3 cache miss rate and the CPI is also far diverged from the optimum performance that the processor can provide. The experiments illustrate an important point. Advanced architectural designs, even those possessing intelligent mechanisms for hiding memory latency, do not necessarily translate to improved application performance. Architecture specific optimization is rather necessary to boost such data mining applications.

In this paper, we present an approach to improve the cache performance of CFP-Tree mining algorithm. First, we improve the cache performance of this algorithms through the design and implementation of CFP-Array data structure. Second, with this cache conscious data structure, we also demonstrate how to extends the use of hardware and software prefetching in reducing the memory access latency. Our experiments evaluation reveals that, cumulatively, these techniques result in a speedup up to 4.2 on a modern-day processor.

Table 1 Cache performance of CFP-Tree algorithm

Dataset	CPI	L3 Misses/1K instr.
Kosarak	7.09	5.04
Accidents	2.88	4.42
Smallwebdocs	6.99	9.78
Bigwebdocs	7.73	14.35
Webdocs	3.42	5.02

2. Background and Related Work

Agrawal et al. [1] introduced a problem of mining association rules among sets of items with given minimum specified confidence in large database. He also proposed the first efficient algorithm, called Apriori, to solve this problem which repeatedly generates the candidates for finding all the frequent itemsets.

Briefly, the problem description by Argrawal is as follows [1], [2]: Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n items, and let the database $D = \{T_1, T_2, \ldots, T_m\}$ be a set of m transactions, where each transaction T_j is a subset of I. An itemset $x \subseteq I$ of size k is known as k-itemset. The *support* of x is $\sum_{j=1}^{m}$ (1:$i \subseteq$ T$_j$), or informally speaking, the number of transactions in D that have x as a subset. Here frequent pattern mining can be defined as a task to find all $x \in D$ that have *support* greater than minimum support value, *minsupp*. An association rule is an implication of the form r:$x_1 \rightarrow x_2$, with $x_1, x_2 \subset I$ and $x_1 \cap x_2 = \emptyset$. Each association rule r has a support defined as *support*(r) = *support*($x_1 \cup x_2$)/*support*(x_1). Once all frequent itemsets and their support are known, the association rules generation is straightforward. Hence, the problem of mining association rules is reduced to the problem of determining frequent itemsets and their support.

Following this first work, many algorithms have been proposed in order to improve the efficiency. And the most outstanding improvements over Apriori would be a method called FP-growth (frequent pattern growth) that succeeded in eliminating the candidate generation [10]. It scans the database only twice and uses compact data structure called FP-tree (frequent pattern tree) to summarize the original transactions.

Motivated by the FP-growth success, another variant of FP-Growth called Compressed FP-Tree(CFP-Tree) [9] has been proposed. This approach tries to compress the original FP-Tree by removing the identical subtrees. And in some extreme characteristic of dataset, the number of nodes in a CFP-Tree can be up to half less than the corresponding FP-Tree.

However, as pointed out by Goethals [11], the pointer-based nature of the FP-tree and CFP-Tree requires costly dereferences. Therefore as the gap between processor and main memory is getting larger, straight forward implementation of FP-Tree and CFP-Tree like data structures will suffer from high CPI and cache misses rate. In this paper we will describe an attempt to reduce the memory access overheads for a CFP-Tree based algorithm and show the experiments result.

3. Compressed FP-Tree

Compressed FP-tree is an extension of FP-Tree algorithm. There are several modifications from FP-Tree in constructing CFP-tree. The HeaderTable of CFP-tree includes four parts: index, item-id, count and pointer to the root of each item′s subtree. Items are in an descending order of their frequency in CFP-Tree. Unlike FP-Tree that stores the item-ids in the tree, in CFP-Tree the item-ids are mapped to an ascending sequence of integers that are the same as the array index in HeaderTable, which reduces the occupied space of item identifier.

In CFP-Tree constructions, all subtrees of the root of FP-Tree (except the leftmost branch) are collected together at the leftmost branch of CFP-Tree as much as possible. And each node consists of five fields: item-id, count array, parent pointer, child pointer and nodelink pointer. Item-id stores the converted original item label (index in the HeaderTable). Meanwhile parent, children and nodelink pointers store the pointers to the parent node, child node and next node with the same item-id respectively. The count array records counts of itemsets in the path form the root to that node and index of the cells in the count array corresponds to the level numbers of the nodes above. One major difference of CFP-Tree algorithm compared to FP-Tree is that CFP-Tree based algorithm avoids creating conditional CFP-Trees, hence for each frequent item, only one local CFP-Tree is created. The CFP-Tree algorithm is describe in Figure 1. Table 2 lists the sample dataset and Figure 2 shows the corresponding CFP-Tree(left).

Table 2 A dataset with *minsupp* = 2

Tid	Items
1	3 4 5 6 7 9
2	1 3 4 5 13
3	1 2 4 5 7 11
4	1 3 4 8
5	1 3 4 10

4. Cache Conscious Optimization on CFP-Tree

In order to improve the memory access performance, some works have been done on FP-growth algorithm. Gothing [8] proposed cache-conscious FP-Tree (CC-Tree). His idea is to reallocate the FP-Tree nodes in sequential memory space in depth-first order. Meanwhile Wei [7] proposed FP-Tree aggregation, lexicographic ordering and differential encoding to store the item id. And more recent work by Li-liu [6] proposed cache conscious FP-Array to improve the cache efficieny of FP-growth algorithm.

Among those proposed optimization techniques, to the best of our knowledge, cache conscious FP-Array is currently the best method to improve the cache access effieciency. It gains the largest performance improvement compared to the other proposed methods. In this paper we will design and implement cache conscious optimization on CFP-Tree algorithm using the similar techniques presented in the FP-Array.

To improve the performance of CFP-Tree algorithm, first we transform the original CFP-Tree into an array based

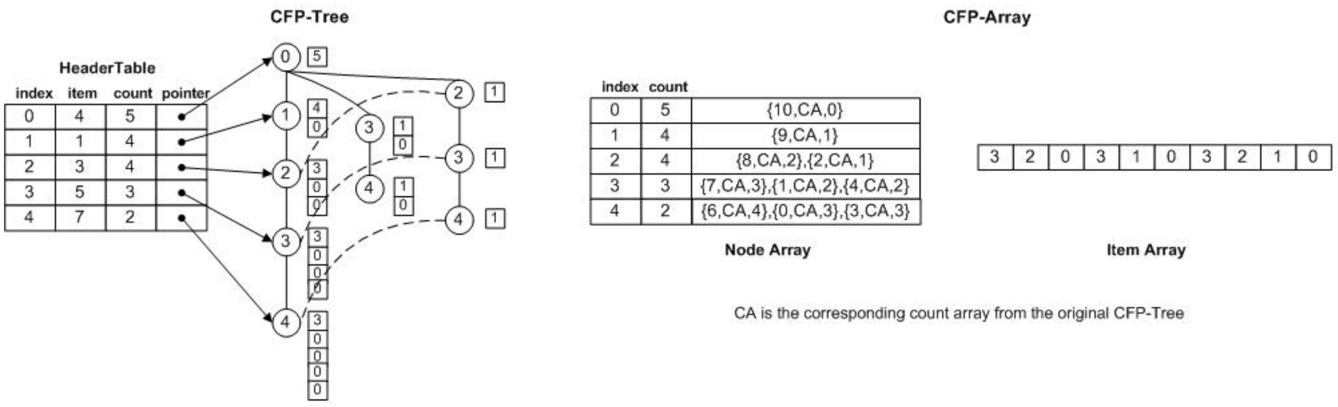

Fig. 2 CFP-Array(Right) After CFP-Tree(Left) Transformation

```
Input: Transaction Database $D$, minimum support $minsupp$
Output: Set of all frequent patterns
Phase 1: Construct CFP-Tree from a database
(1) Scan the database $D$ once, gather frequency of all items.
(2) Sort items based on their frequency in descending order.
(3) Construct the left most branch of the tree.
(4) Scan the database a second time: for each transaction, remove in-
frequent items and sort the transaction, insert the transaction starting
from the node pointed by HeaderTable with the same index as the first
item in the transaction. Increment the count array and each inserted
node is linked to its respective node link.
Phase 2: Mine the CFP-Tree by calling CFPMining()
CFPMining(CFP-Tree, suffix)
For each item $\alpha$ in the HeaderTable of CFP-Tree
(1) Output $\alpha \cup$ suffix as frequent
(2) Find all frequent items in the conditional pattern base C for $\alpha$
(3) If there is no frequent item in C, end this loop iteration
(4) Generate Local CFP-Tree $\tau$.
(5) If $\tau$ has only one path, output any sub set of items in this path,
end this loop, otherwise mine $\tau$ by calling CFPMiningLocal().
CFPMiningLocal(Local CFP-Tree, Local HeaderTable, suffix)
For each item $\beta$ in the HeaderTable of Local CFP-Tree $\tau$
(1) Output $\beta \cup$ suffix as frequent
(2) Find all frequent items in the conditional pattern base C for $\beta$
(3) If there is no frequent item in C, end this loop iteration
(4) Generate Local HeaderTable $\gamma$.
(5) CFPMiningLocal($\tau$, $\gamma$, $\beta \cup$ suffix)
```

Fig. 1 CFP-Tree Algorithm

data structure, which we named Compress FP-Array (CFP-Array). Figure 2 shows the CFP-Array(right) after transforming the original CFP-Tree(left). With this transformation we reallocate the original CFP-Tree in contiguous memory space. There are no pointers in the CFP-Array, and thus, the pointer based tree data structure is eliminated after the transformation. In CFP-Array there are two arrays, item array and node array. The item array works essentially as the replication of CFP-Tree. Each list in the node array is associated with one frequent item, while 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 of count array, and transaction size. This separation of node array and item array yields more compact data size because node size is much smaller than the original node size in CFP-Tree (we only store the item-ids in the item array). The other four members in the node of CFP-Tree, e.g. child pointer, nodelink pointer, parent pointer and count array are converted into the corresponding members in the node array, which is not along the critical path. Moreover, to further optimize the CFP-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. Figure 3 shows the algorithm of transforming the CFP-Tree into the CFP-Array in depth-first order.

To further utilize the ability of current modern processors we also extend the use of hardware and software prefetching capabilities to bring the data onto the cache before it is needed by the processor. Software prefetching initiates a data prefetch instruction to the processor, which specifies the address of a data word to be brought into the cache. In contrast, hardware prefetching employs special hardware which records memory access patterns of the application and prefetches data on a best effort basis. Figure 4 shows the algorithm of accessing the cache-conscious CFP-array that uses hardware and software prefetching.

5. Experiments

In this section we will briefly describe our experiments. In these experiments we used a machine with 2 cores Intel Xeon processors which are running at 3.2GHz with 2GB physical memory. Each core are equipped with 8KB L1 data cache, 512KB unified L2 cache and 1 MB L3 unified cache. The cache line sizes are 64 bytes for the L1, L2 and L3 caches. Furthermore, we used the Intel VTune performance analyzer to collect the cache performance numbers. Throughout this section, we will compare the execution time of a CFP-Array based algorithm with that of a CFP-Tree based algorithm.

Table 3 and 4 show the datasets used in our evaluation.

```
Algorithm: Transformation of CFP-Tree into CFP-array
Input: CFP-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 a in T, allocate memory space for NA[$\alpha$]
(3) For each child node C of the root node in CFP-tree T
      Visit(C, null, 0, IA, NA)
(4) Release memory space for T
Algorithm: Visit(N, S, 0, IA, NA)
Input: CFP-Tree node N, item stack S, depth D, Item array IA, and node arrays NA
Output: None
(1) If N is neighbor node and have no children, copy items in S to IA
(2) Allocate an element from NA[item label of N], set the count array, 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
```

Fig. 3 CFP-Tree Transformation Algorithm

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

Fig. 4 CFP-Array Traversal

Table 3 Experiment Datasets

Name	Num. Of Trans.	Size	Min-Sup	Avg. Eff. Trans. Len.
Kosarak	990000	31M	800	6
accidents	340000	34M	40000	28
smallwebdocs	230000	200M	20000	56
bigwebdocs	500000	460M	50000	49
webdocs	1690000	1,46G	300000	23

Table 4 Synthetic Experiment Datasets

Name	Trans. Num.	Avg. Item Per Trans.	Num. Of Different Items	Size
T60I200D100K	100000	60	200	17M
T60I30KD100K	100000	60	30000	35M

Accidents, Kosarak and Webdocs are the datasets from the Frequent Itemset Mining Implementations Repository [13]. 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 4^{th} and 5^{th} columns in table 3 lists the min-support for each dataset and average effective transaction length (infrequent 1-items are pruned away). Meanwhile to show the specific behavior of CFP-Array we also generated synthetic dataset. We used IBM Synthetic Data Generator [14] to generate the dataset. We generated two datasets, T60I200D100K and T60I30KD100K; the first represents dense dataset and the second represents sparse dataset.

We first evaluated the CFP-Array optimization. Figure 5 shows speedups of CFP-Array based algorithm over CFP-Tree based algorithm. From Figure 5, it is evident that we achieved significant performance improvement due to spatial locality optimization. It could obtain a speedup of 3.9 on average. When hardware prefetcher is enabled it provides an additional speedup up to 11%. And when hardware prefetcher and software prefetcher are enabled it provides an additional speedup up to 13%. Kosarak and Bigwebdocs had the largest speedup. Kosarak gained large speedup primarily due to its sparse dataset, hence the cache misses was reduced significantly by transforming CFP-Tree into CFP-Array. Meanwhile for Bigwebdocs it gained large speedup due to its large dataset with long average of effective transaction length. In contrast the performance of Accidents was not pronounced and even worse than that of the original CFP-Tree because it held a relatively small working set.

Figure 6 describes our experiment result measuring the speedup on Bigwebdocs dataset. It shows that CFP-Array consistently outperforms the CFP-Tree baseline with the variety of minimum support. We think lowering support values allows more opportunities for CFP-Array to reduce the cache misses and CPI and hence will increase the speedup.

Figure 7, 8, 9, and 10 describe the detail information about cache utilization efficiency. The original CFP-Tree has very high CPI. It is ranging from 2.88 until 7.73 depends on the dataset. Meanwhile with CFP-Array transformation we could reduce the CPI to 1.38. And when prefetching techniques are applied it could reduce the CPI to 1.27. Hence with CFP-Array and prefetching techniques we could get much lower CPI than the original CFP-Tree.

Furthermore in terms of L1, L2 and L3 cache efficiency, CFP-Array shows better performance over the original CFP-Tree. At the first level cache, CFP-Array on average had more than 90% L1 hit rate, meanwhile L1 hit rate was around 50% for the original CFP-Tree. At the second level cache CFP-Array also showed some improvements. CFP-Array on average had more than 99% L2 hit rate, meanwhile for the original CFP-Tree L2 hit rate was only around 96%. And in the last level cache, CFP-Array reduces the number of L3 misses rate significantly. The original CFP-Tree algorithm had average of L3 misses rate around 7.72 MPKI, meanwhile

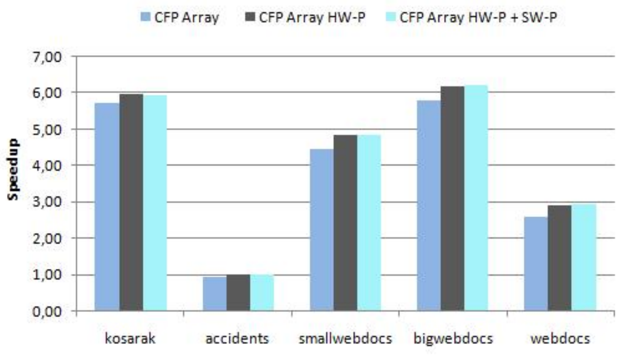

Fig. 5 Speedups of CFP-Array based algorithms over CFP-Tree algorithm

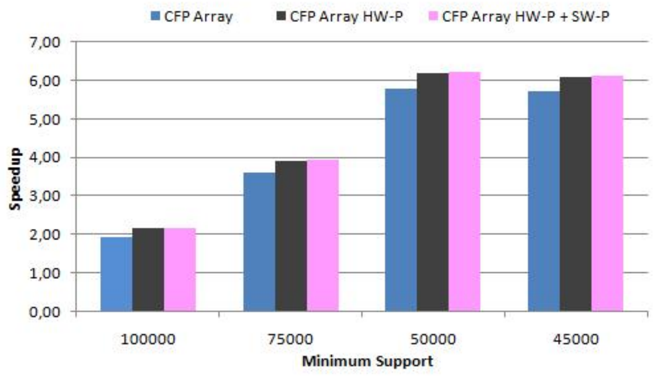

Fig. 6 Speedups on Bigwebdocs

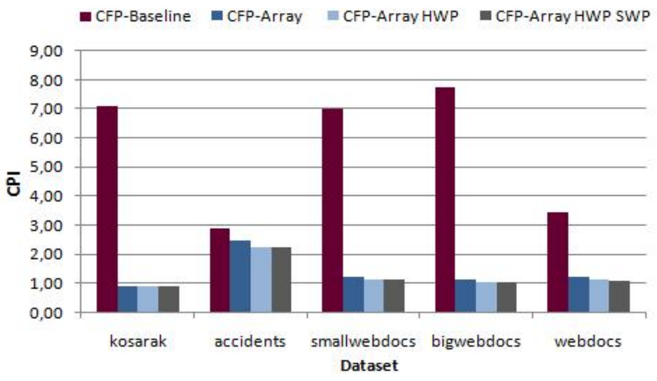

Fig. 7 CFP-Array CPI reduction

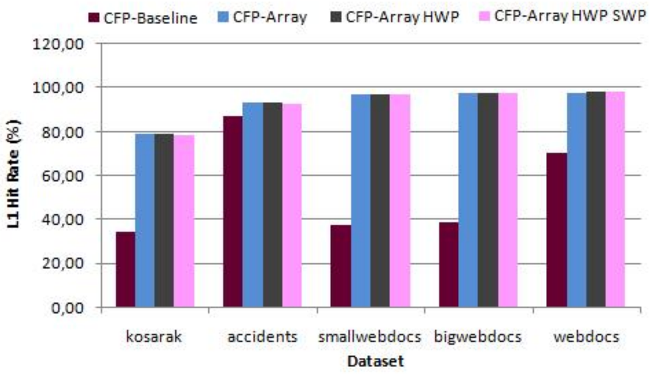

Fig. 8 CFP-Array L1 hit rates

L3 misses rate for CFP-Array was only around 1.5 MPKI. On average we could reduce the number of L3 misses rate by factor of 5.

Figure 11 and 12 show CFP-Array speedups on synthetic dataset. On dense dataset (T60I200D100K), CFP-Array speedup ratio is decreasing when we lower the minimum support. In the other hand, on sparse dataset (T60I30KD100K) CFP-Array speedup ratio is increasing when we lower the minimum support. In the dense dataset, we think that CFP-Tree could compact the tree data structure enough. When we lower the minimum support the dataset get more denser, transforming CFP-Tree into array structure does not improve the cache efficiency so much. Meanwhile in the sparse dataset CFP-Tree could not so much compact the tree structure, so the tree is now become larger and sparser when we lower the minimum support. Hence on sparse dataset transforming CFP-Tree into compact array structure could give more benefits.

Our finding of these experiments is that, cache-conscious CFP-Array transformation makes the following benefits:

- By converting the CFP-Tree into CFP-Array 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. Hence it will reduces the cache miss rate in the CFP-Array traversal.
- We extends the use of hardware prefetching and software prefetching in CFP-array data structure. This could be utilized to reduce the latency time between processor and main memory when traversing CFP-Array.

6. Conclusion

In this paper we have presented cache-conscious optimization techniques on CFP-Tree based algorithms to alleviate the performance gap between the processor and main memory. We implemented an approach to make the data structure more contiguous and also presented the design of CFP-Array to take the advantage of hardware and software prefetcher. Our idea is to transform the original CFP-Tree into CFP-Array data structure to improve the spatial locality as well as reducing the memory access overheads and improving the cache utilization efficiency.

We also have presented some experiment results on a CFP-Array based algorithm, and showed that it outperforms the original straight forward implementation of CFP-Tree algorithm. The experiments showed that it could reduce the CPI and cache misses significantly, and thus also reduce the execution time. We also showed that CFP-Array could

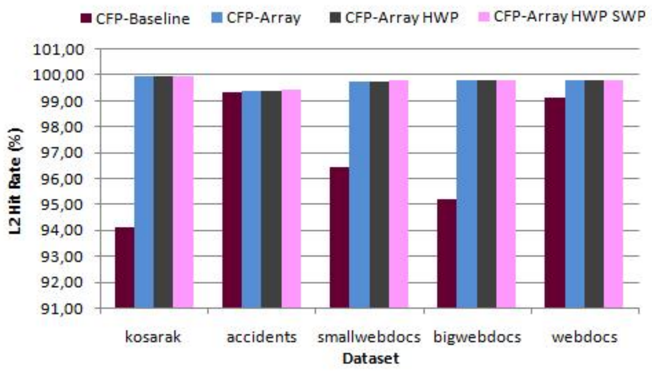

Fig. 9 CFP-Array L2 hit rates

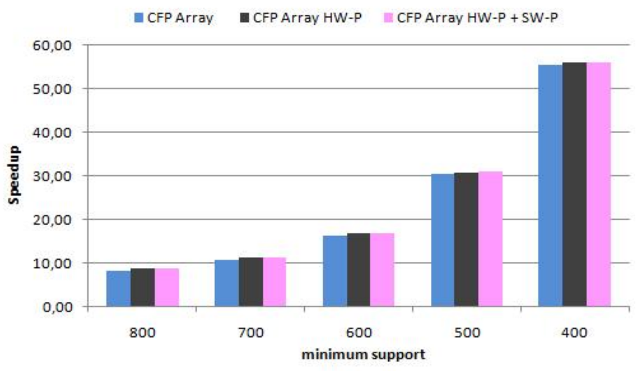

Fig. 12 Speedups on T60I30KD100K

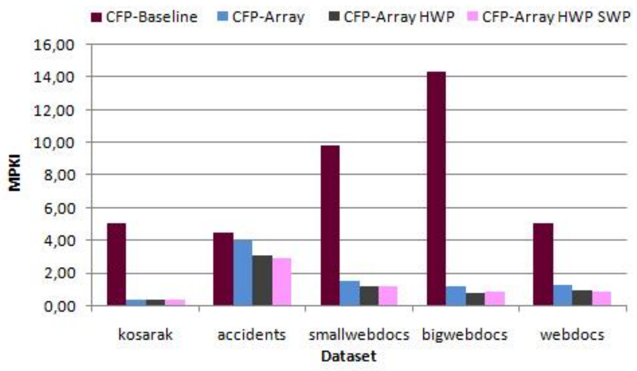

Fig. 10 CFP-Array L3 cache misses

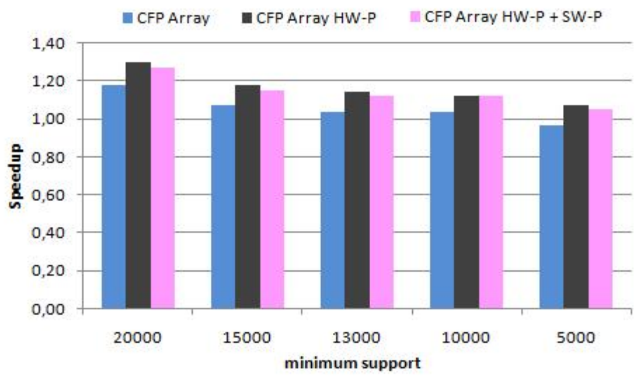

Fig. 11 Speedups on T60I200D100K

give more benefits when the dataset is sparse, in other hand when dataset is dense the benefit is not necessarily well pronounced. Finally as the gap performance between CPU and memory continues to grow, it should be clear that the importance of cache-aware algorithm design will grow equally.

For the future work, we would like to explore more deeply the behavior of cache conscious frequent pattern mining algorithms in large datasets. We would like to verify the effectiveness of the current cache conscious approach in a vast variety of configurations. Another future work is to study cache-conscious approach in other frequent itemsets mining algorithms, such as LCM [12], and in the different processor architectures, such as a GPU (Graphic Processor Unit).

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 Databases*, 1994.

[3] Wenke Lee and Salvatore J. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium* , 1998.

[4] B. Mobasher, N. Jain, E-H. Han, and J. Srivastava. Web mining: pattern discovery from world wide web transactions. In *Technical Report TR96-050, Department of Computer Science University of Minnesota*, 1996.

[5] Shan Luy, Soyeon Parky, Chongfeng Huy, Xiao May, Weihang Jiangy, Zhenmin Liyz, Raluca A. Popax, Yuanyuan Zhouyz. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of ACM SIGOPS symposium on Operating systems principles*, 2007.

[6] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of the International Conference on Very Large Databases*, 2007.

[7] Mingliang Wei, Changhao Jiang, and Marc Snir. Programming Patterns for Architecture-Level Software Optimization on Frequent Pattern Mining. In *Proceedings of the International Conference on Data Engineering*, 2007.

[8] Amol Gothing, Gregory Buehrer, Sinivasan Partharasary, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradep Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Databases*, 2005.

[9] Yudho Giri Sucahyo and Raj P. Gopalan. CT-PRO: A Bottom-Up Non Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structure. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, 2004.

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

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

[12] T. Uno, T. Asai, Y. Uchida, H. Arimura. LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In *Proceedings IEEE ICDM'03 FIMI'03*, 2003.

[13] Frequent Itemset Mining Implementations Repository. *http://fimi.cs.helsinki.fi/*.

[14] Synthetic Data Generation Code for Associations and Sequential Patterns. *http://www.almaden.ibm.com/software/quest/Resources/index.shtml*. Intelligent Information Systems, IBM Almaden Research Center