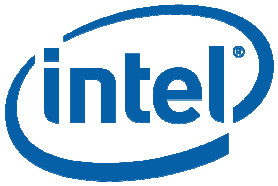


Optimization of Frequent Itemset Mining on Multiple-Core Processor

Li Liu, **Eric Li**, Yimin Zhang, Zhizhong Tang

Tsinghua University

Beijing 100084, China

Intel China Research Center
Intel Corporation

Beijing 100080, China

VLDB 2007

Sep. 27, 2007, University of Vienna, Austria

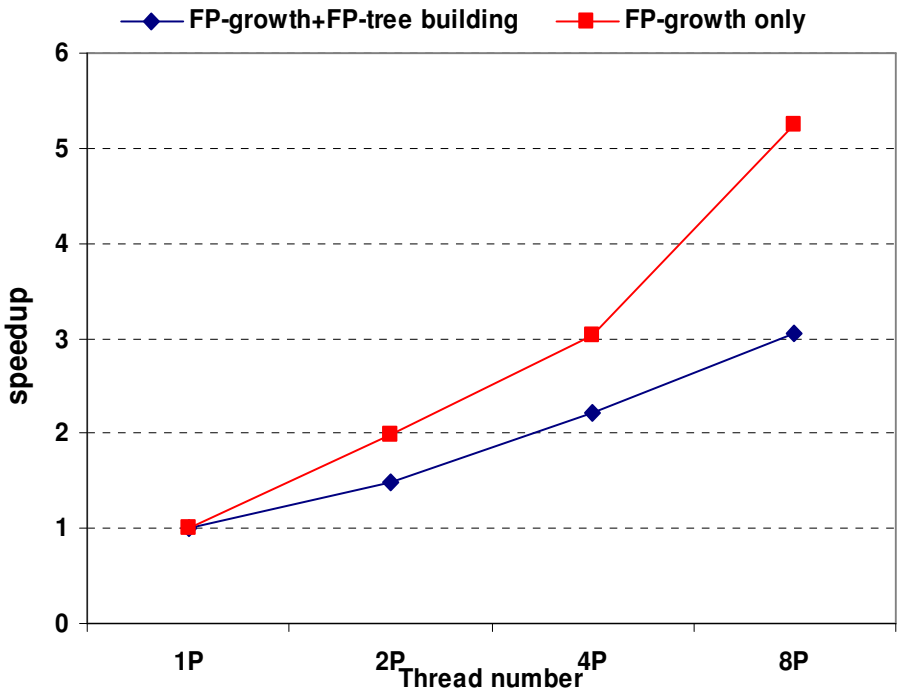
Outline

- **Motivation and contributions**
- Introduction
- Cache-conscious optimization
- Lock-free parallelization
- Performance evaluation
- Summary

Motivation

- Data mining applications
  - Fast growing segment
  - Compute and memory intensive
- Evolution of modern architecture
  - Cache performance becomes more critical
  - Entering the chip multiple processor era
- Frequent itemset mining
  - A typical example to show the significance of architecture oriented algorithm design

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



Contributions

- We improve the cache performance through design of a cache-conscious FP-array
  - Improve data locality performance
  - Hardware and software prefetching
  - Reduce off-chip memory access and improve scaling performance
- We present a lock-free method to efficiently parallelize the FPGrowth algorithm
  - Dataset tiling and the hot sub-tree provide a lock-free mechanism in FP-tree building
  - Better scaling performance to harness the multi-core processing capability

Outline

- Motivation and contributions
- **Introduction**
- Cache-conscious optimization
- Lock-free parallelization
- Performance evaluation
- Summary

Frequent Itemset Mining (FIM)

- Aims to discover groups of items or values that co-occur frequently in a dataset
- An example
  - Dataset:
    - T1: A, B, C, D,
    - T2: B, C, D
    - T3: A, B, E
  - When min-support = 2, The FIMs are:
    - {A} {B} {C} {D}
    - {A, B} {B, C} {C, D} {B, D}
    - {B, C, D}
  - Anti-monotone Apriori property: if any length *k* itemset is not frequent in the database, its length (*k* +1) super-itemset can never be frequent

FIMI algorithms

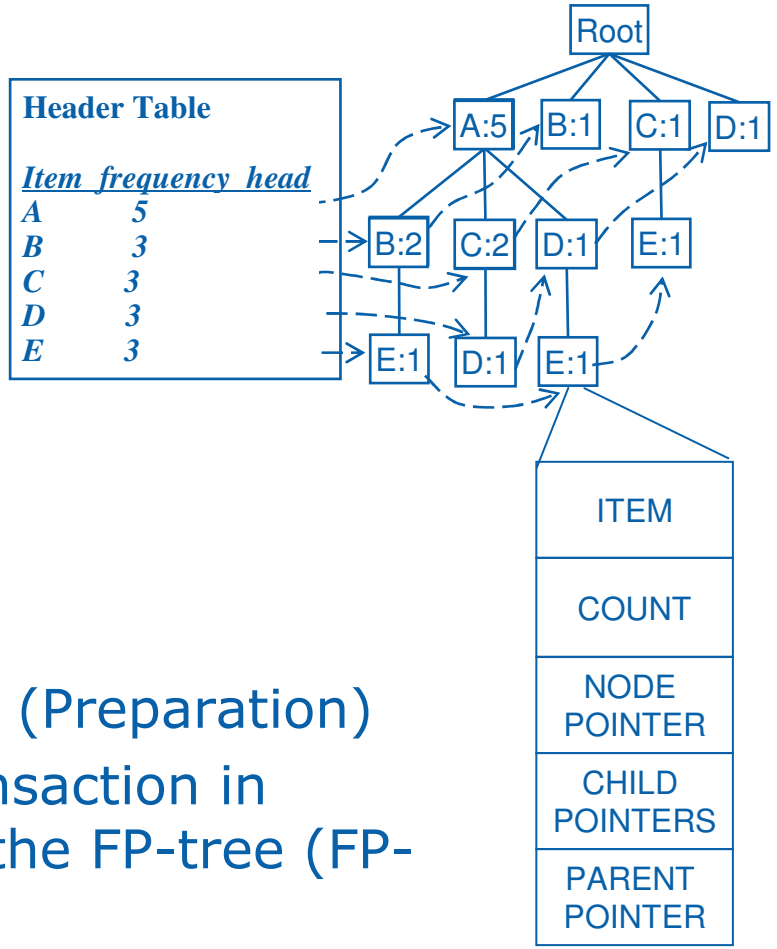
- Apriori (The first algorithm)
- DHP
- DIC
- Eclat
- Partition
- **FPGrowth** (The first **FP-tree** based algorithm)
- **Cache-Conscious tree** (The state-of-the-art algorithm)
- Nonordfp
- KDCI, parKDCI

FP-tree building

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

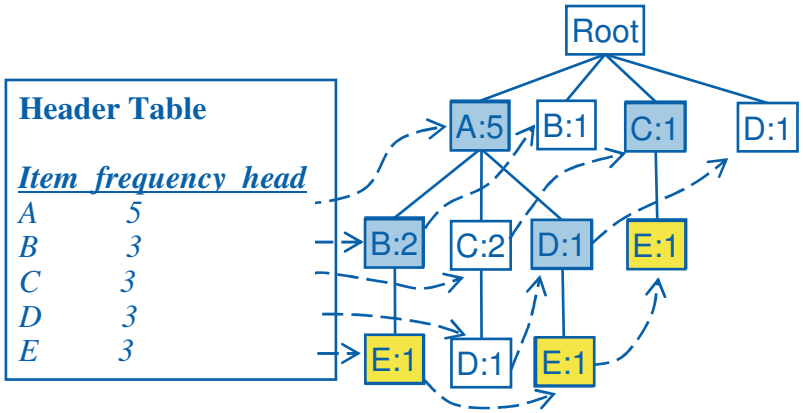
Min-support=3
Frequent items: A,B,C,D,E

Insert other transactions



- Build FP-tree
  - Count the frequency of each item (Preparation)
  - Insert frequent items of each transaction in frequency descending order into the FP-tree (FP-tree building)

FP-growth



Conditional Pattern Bases

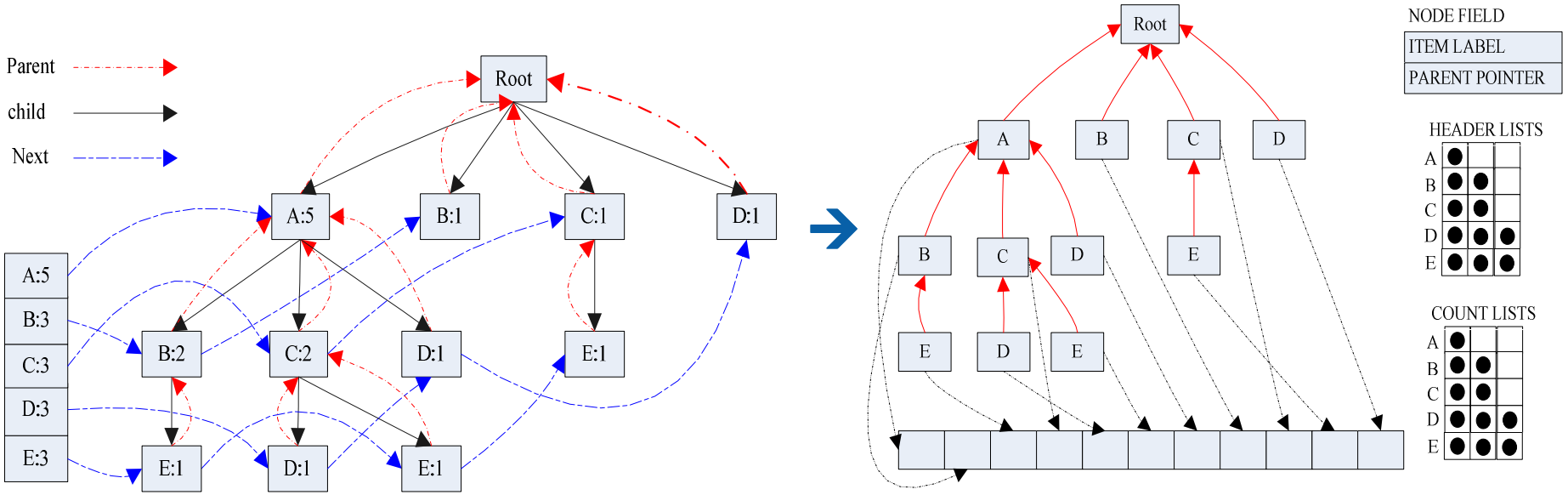
item	*cond. pattern base*
E	
D	
C	
B	

- FP-growth (mining an existing FP-tree)
  - For each item in the FP-tree
    - Construct the conditional pattern base

Poor spatial locality: only two node fields are used in the FP-tree traversal, pointer de-referencing

Poor temporal locality: large data structure

Cache-Conscious tree (CC-tree)



- CC-tree
  - FP-tree data structure reorganization
  - More cache friendly, but it still suffers from pointer de-referencing

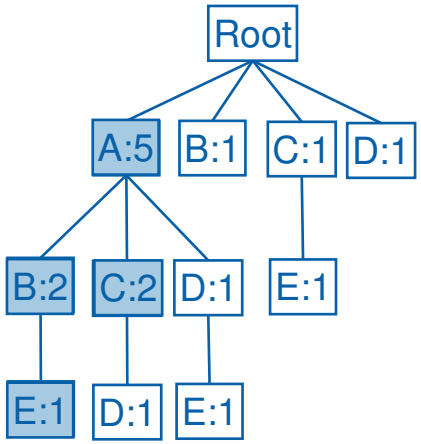
Outline

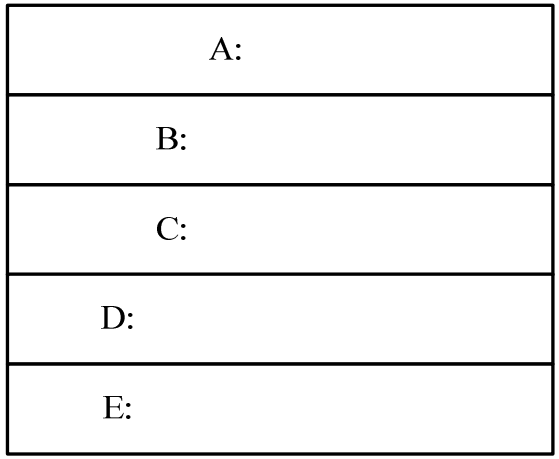
- Motivation and contributions
- Introduction
- **Cache-conscious optimization**
- Lock-free parallelization
- Performance evaluation
- Summary

Cache-Conscious FP-array

- Item array
  - Each element is an item label
  - Replicate items in the joint path in the FP-tree
- Node array
  - Organized as an array list. Each item records the occurrences of a frequent item in the item array
  - 3 elements of each item
    - begin position of the item in the item array
    - reference count
    - transaction size
- Traverse the FP-tree in a depth-first order to build FP-array

FP-array construction from an FP-tree





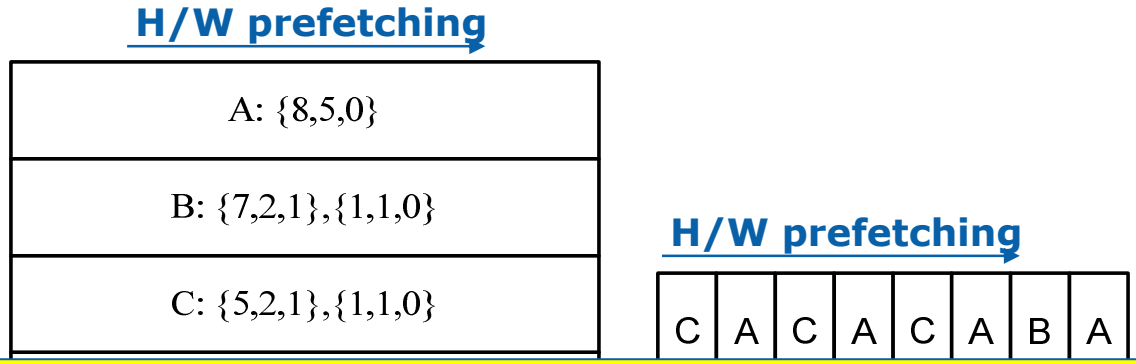
Node Array

Item Array

Visit other nodes

FP-array optimization

- Compact data size for the item array
- Hardware prefetching
  - Continuous accesses in both the node array and the item array
- Software prefetching
  - Prefetch data in the item array, located by the node array



FP-array provides a smaller node size than FP-tree and CC-tree: improve cache line utilization and spatial data locality

Hardware prefeteching: enable strided data access pattern

Software prefeteching: enable non-strided transaction data access

Outline

- Motivation and contributions
- Introduction
- Cache-conscious optimization
- **Lock-free parallelization**
- Performance evaluation
- Summary

The importance of FP-tree building parallelization

- FP-tree building
  - Largely ignored due to its small execution time compared to FP-growth
  - However, after FP-array optimization the FP-tree building time consists of 10~40% of the total run time
  - It cannot be ignored according to Amdahl's law
- Traditional approaches
  - Multi-tree method
  - Lock based single tree method

Dataset tiling

- General terms
  - Hot item: A, B; Cold item: C, D, E
  - Class id: the bitmap of hot items
  - Hot FP-node: the FP-tree node corresponding to the hot item
  - Hot sub-tree: the sub-tree consists of all the hot nodes
  - Hot node hashing table: each hot node can be hashed by a unique class id

No.	Sorted Transactions	New transactions
1	A,B,E	<11, 1, (E)>
2	A,C,D	<01, 2, (C,D)>
3	A,C,E	<01, 2, (C,E)>
4	A,D	<01, 1, (D)>
5	A,B	<11, 0, ()>
6	B	<10, 0, ()>
7	C,E	<00, 2, (C,E)>
8	D	<00, 1, (D)>

Lock-free FP-tree building

- Preparation: generate new transactions as <class id, cold item number, cold item list>
- Dataset tiling: merge new transactions with the same class id into a tile
- Dataset insertion: insert new transactions in each tile into the tree. For each tile, append each transaction to the hot node corresponding to the class id

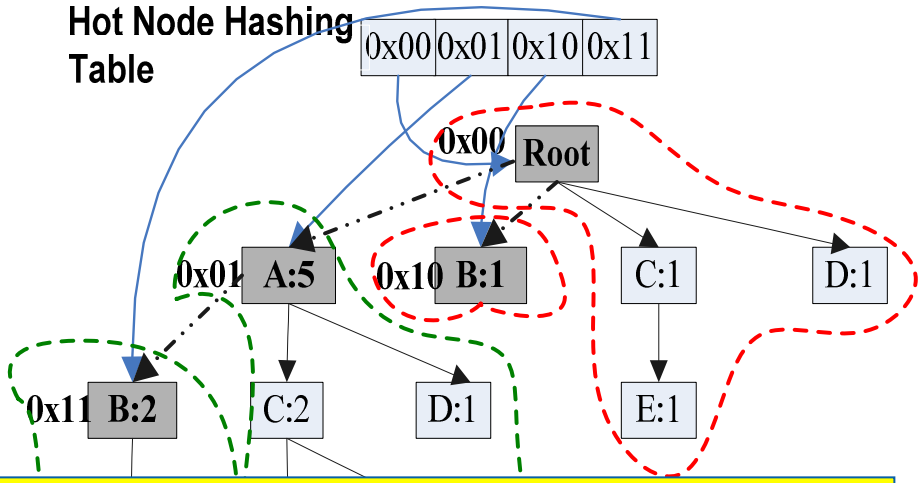
No.	Sorted Transactions	Class id
1	A,B,E	11
2	A,C,D	01
3	A,C,E	01
4	A,D	01
5	A,B	11
6	B	10
7	C,E	00
8	D	00

Tile Id.	New transactions
00	<2, (C,E)>, <1, (D)>
01	<2, (C,D)>, <2, (C,E)>,<1, (D)>
10	<0, ()>
11	<1, (E)>, <0, ()>

Lock-free FP-tree building

- Preparation: generate new transactions as <class id, cold item number, cold item list>
- Dataset tiling: merge new transactions with the same class id into a tile
- Dataset insertion: insert new transactions in each tile into the tree. For each tile, append each transaction to the hot node corresponding to the

Tile Id.	New transactions
00	<2, (C,E)>, <1, (D)>
01	<2, (C,D)>, <2, (C,E)>,<1, (D)>
10	<0, ()>
11	<1, (E)>, <0, ()>



Tiles are independent with each other: a natural lock-free parallel mechanism in FP-tree building

Tile-by-tile insertion leads to better temporal data locality performance

Lock-free FIMI implementation

- FP-tree building
  - Parallel preparation: select 16 hot items to build a hot sub-tree, and each thread generates new transactions from the original dataset
  - Parallel dataset tiling: each thread merges the new transactions into tiles according to the class id information
  - Parallel dataset insertion: each thread inserts a set of tiles into the FP-tree in a lock-free manner
- FP-growth
  - The frequent-1 items can be simply parallelized after building the FP-tree

Outline

- Motivation and contributions
- Introduction
- Cache-conscious optimization
- Lock-free parallelization
- **Performance evaluation**
- Summary

Experimental Setup

Hardware:

	Xeon MP 7130M
CPU type	Dual-core
Core speed	3.20GHz
L1 data cache	16KB
L2 cache	2x1MB
L3 cache	4MB(unified)

- 4-U Xeon system: 4 Xeon MP 7130M CPU, total 8 cores
- Physical memory: 4GB

Software:

- OpenMP programming model
- Intel C/C++ 9.1 compiler
- Intel Vtune performance analyzer

Experimental Setup

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
Smallwebdocs	230000	200M	12000	662	54.1
Bigwebdocs	500000	460M	50000	280	25.2
Webdocs	1690000	1.46G	120000	428	35.8

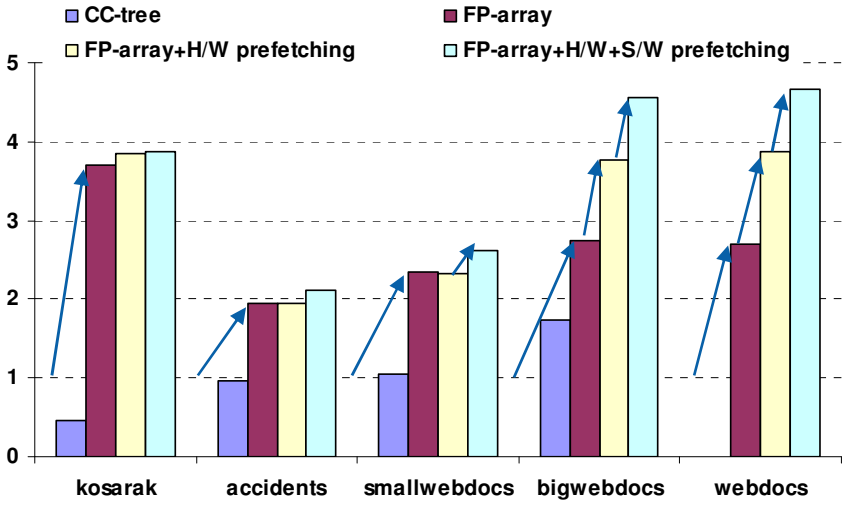
Dataset

- *Accidents*, *Kosarak* and *Webdocs* are the datasets from the Frequent Itemset Mining Implementations Repository
- *Smallwebdocs* and *Bigwebdocs* are artificial datasets which are cut from Webdocs

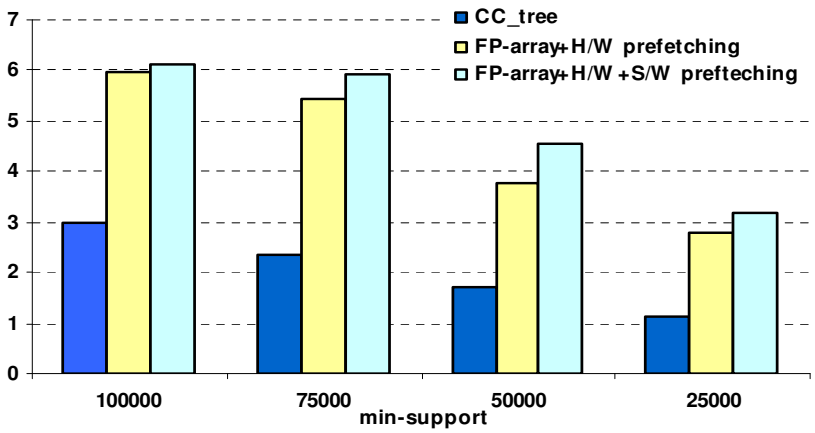
Impact of FP-array optimization

FP-growth sequential performance analysis

- FPGrowth - base
- CC-tree
- FP-array
  - 2.7 fold speedup when H/W prefetch is off
  - Hardware prefetching provides an additional 5%~30% speedup
  - Software prefetching provides 20% speedup for the large datasets
  - Consistently outperforms CC-tree and FP-tree with the decreasing of min-support



FP-growth sequential speedup

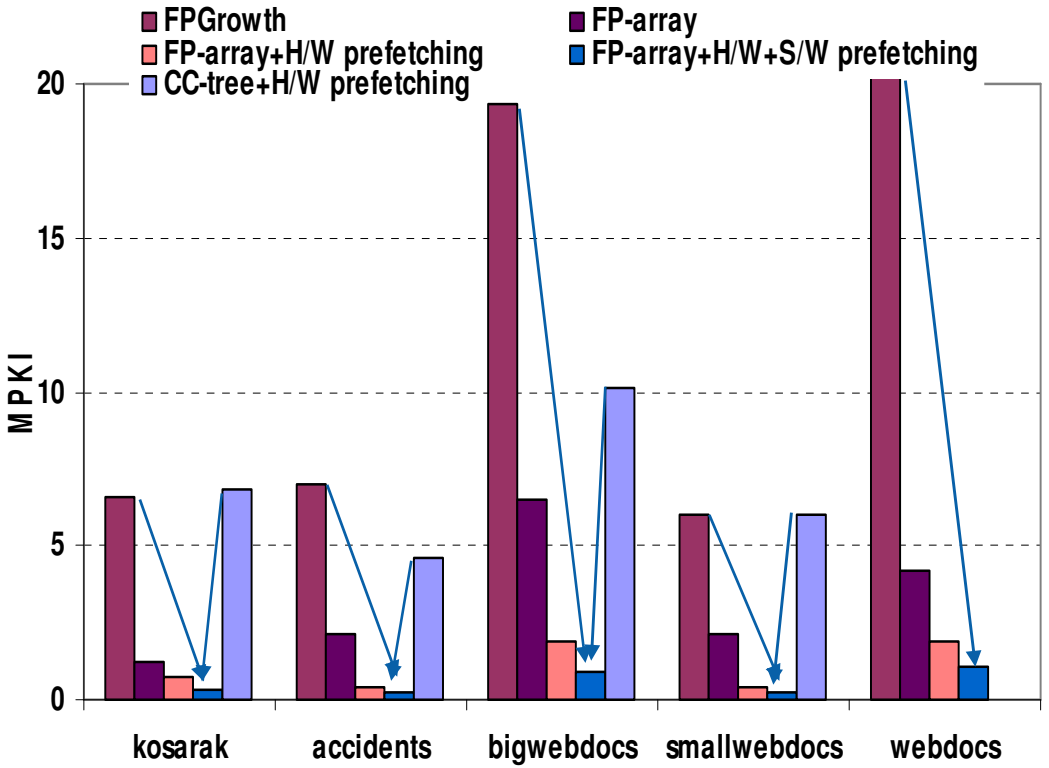


FP-growth sequential speedup of Bigwebdocs

Impact of FP-array optimization

FP-growth cache performance analysis

- FP-array reduces the cache misses by a factor of 17 on average compared to the baseline FPGrowth
- Much better than CC-tree in terms of cache performance

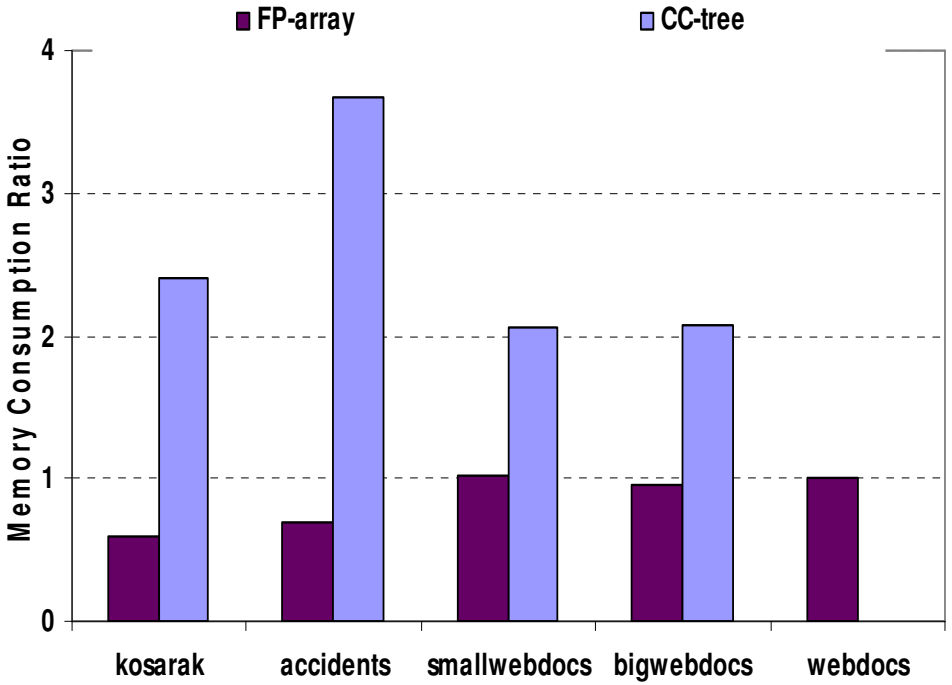


L3 cache miss reduction in FP-growth

Impact of FP-array optimization

FP-growth memory requirement analysis

- Base
  - FPGrowth
- CC-tree
  - Increases memory requirement significantly and sometimes fails for large size data input
- FP-array
  - Does not increase memory requirement and saves memory for some cases

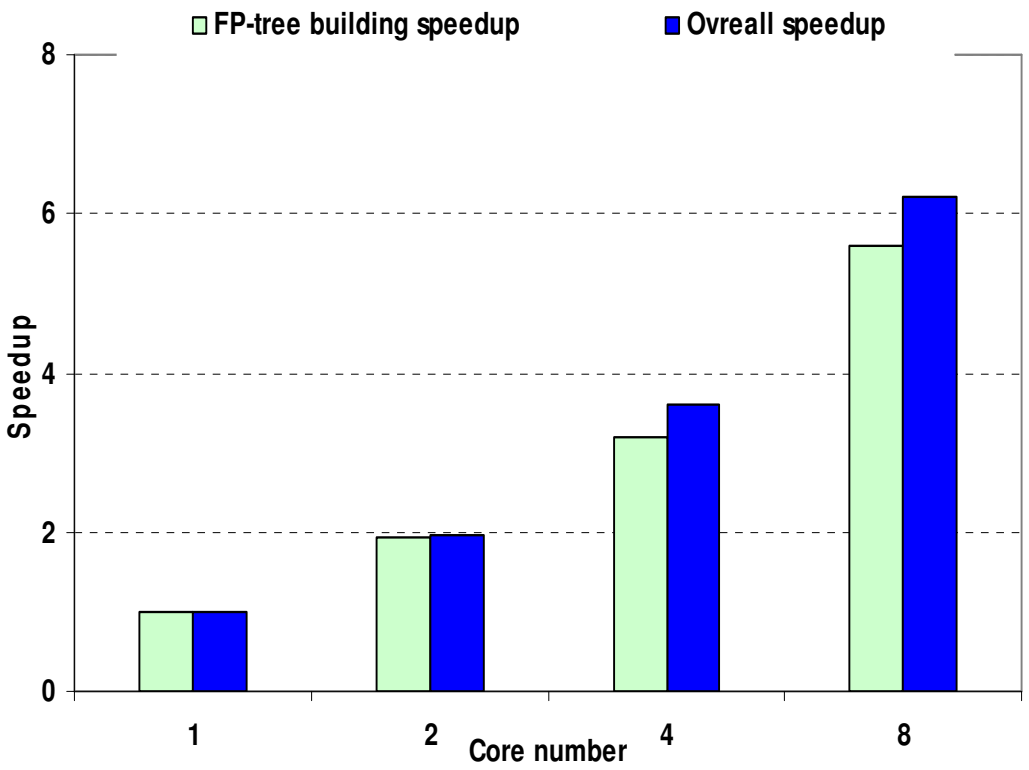


Memory consumption ratio in FP-growth

Impact of Lock-free parallelization

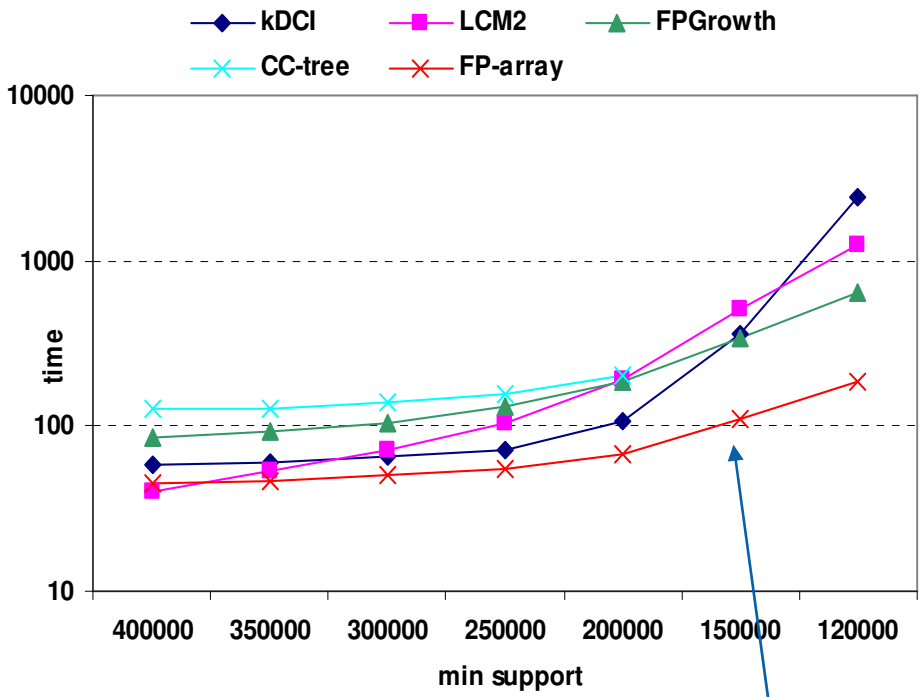
Scaling performance analysis

- Lock-free FP-tree building obtains an average 5.6 fold speedup on the 8-core system
- The whole application gets a 6.1 fold speedup

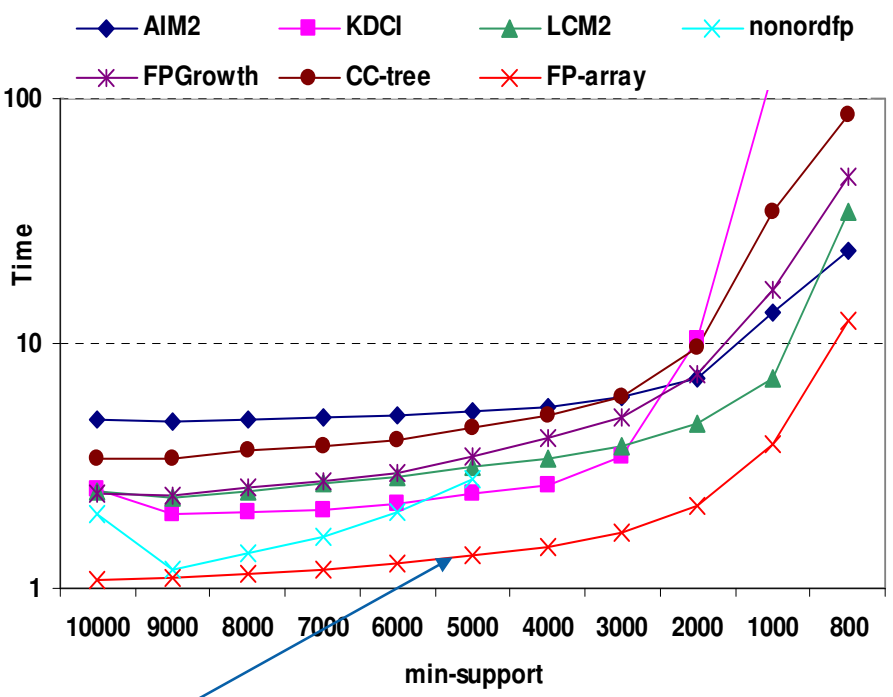


FP-tree building and overall scaling performance

Overall execution time evaluation



Total execution time of "Webdocs"



Total execution time of "Kosarak"

Serial FP-array algorithm. The parallelized version is 6.1x faster

Outline

- Motivation and contributions
- Introduction
- Cache-conscious optimization
- Lock-free parallelization
- Performance evaluation
- **Summary**

Summary

- Proposed a cache-conscious FP-array for FP-tree based algorithm
  - Improves spatial data locality
  - Allows for hardware and software prefetching
  - 4.0 fold speedup on a single core
- Proposed a new parallel mechanism to enable lock-free tree-building
  - Improves the temporal cache performance
  - Makes the algorithm amenable to the thread level parallelization
  - 6.1 fold speedup on an 8-core system and a final 24 fold speedup
- Effective algorithm design in data mining needs to take into account modern architectural designs

Questions?