On Benchmarking Frequent Itemset Mining Algorithms

from Measurement to Analysis

Balázs Rácz
Computer and Automation Research Institute of the Hungarian Academy of Sciences
Lágymanyosi u. 11., H-1111 Budapest, Hungary
bracz+fp5@math.bme.hu

Ferenc Bodon
Department of Computer Science and Information Theory,
Budapest University of Technology and Economics
Magyar tudósok körútja 2. H-1117 Budapest, Hungary
bodon@cs.bme.hu

Lars Schmidt-Thieme
Computer Based New Media Group (CGNM)
Albert-Ludwigs-Universität Freiburg
Georges-Koehler-Allee, D-79110 Freiburg, Germany
lst@informatik.uni-freiburg.de

ABSTRACT

We point out problems of current practices in comparing Frequent Itemset Mining Implementations, and suggest techniques that can help to avoid the conclusions of measurements being tainted by these problems.

1. INTRODUCTION

Frequent Itemset Mining (FIM) is one of the initial, most basic problems of Data Mining. It has wide applications not only in its natural form, but also as a subroutine in various other problems, the most famous being association rule mining.

During the past decade, over 100 papers were published about Frequent Itemset Mining. Some of these brought novel approaches, while others tuned them with heuristics and data structure optimizations. There is one common feature in all the papers: an implementation was benchmarked and shown to be faster/better (in memory usage or disk access) than... some other (publicly available) implementation, on some mining tasks (datasets and support threshold levels). While based on this approach everybody claimed that their algorithm is the fastest/best on the field, this can obviously be not true.

This called for the 'Frequent Itemset Mining Implementations' (FIMI) workshops[4] held in conjunction with ICDM03 and ICDM04, where members of the community were invited and encouraged to send their implementation. A benchmark was run over all submitted implementations and a wide set of publicly available input data. This raised the evaluation of FIM algorithms to a new level, thus fortunately holding back the flood of proudly presented papers.

Detailed understanding of the problem Frequent Itemset Mining, neither of existing algorithms and approaches is not necessary to read this paper. Nevertheless, we give some pointers to the readers who are not familiar with the field [3, 4].

The rest of the paper is organized as follows: In Section 2 we discuss problems and benchmarking issues of the FIMI contests. Section 3 introduces programming techniques, including the proposal of a unified, highly optimized I/O framework. In Section 4 we present issues concerning the actual measurement, machine dependance, and selection of performance metric. We also show sample analysis diagrams for selected FIM implementations. In Appendix A a short introduction is given to modern computing hardware, which may be necessary to understand some details of Section 4.

2. ON FIMI CONTESTS

There is an important difference between traditional data mining contests and Frequent Itemset Mining evaluation. In tranditional contests, there are two important measures: the *quality* of the algorithm, i.e., how good the output of the algorithm is (like prediction quality), and the running time of the implementation. As in case of the FIM problem the task is mathematically defined and solvable, i.e., there is one correct output, the quality is measured only in terms of resource usage: mostly running time and also memory usage.

PROBLEM 1. *We are interested in the quality of algorithms, but we can only measure implementations.*

This is an issue of all fields of Computer Science where theoretical considerations, such as asymptotical running time analysis can yield only very loose bounds on actual performance.

PROBLEM 2. *If we gave our algorithms and ideas to a very talented and experienced low-level programmer, that could completely re-draw the current FIMI rankings.*

PROBLEM 3. *Seemingly unimportant implementation details can hide all algorithmic features when benchmarking. These details are often unnoticed even by the author and almost never published.*

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

Such differences can be explained by hardware reasons, for example some implementations of the same algorithms may be better served by the memory hierarchy than others. See Figure 1 for an illustration of 10-fold difference in running times.

The history of FIM algorithms is a very good example of people not knowing what to aim for. The initial hypothesis was that I/O size is the factor that primarily determines performance (this resulted in the level-wise apriori-like algorithms). In the meantime focus was moved to reducing the number of false candidates (resulted in different hashing techniques). This hypothesis was also obsoleted by the stunning performance of depth-first mining algorithms like eclat and FP-growth, known to generate more candidates than their predecessors. The question comes naturally: What determines the performance of current algorithms/implementations?

PROBLEM 4. *FIM implementations are complete 'suites' of a basic algorithm and several algorithmic/implementational optimizations. Comparing such complete 'suites' tells us* what *is fast, but does not tell us* why.

Recommendation 1. The suites should be programmed in a way that the optimization features can be turned on/off, and several possible underlying data structures are pluggable. Running benchmarks over these options would give us an insight on what counts and what doesn't.

PROBLEM 5. *All 'dense' mining tasks' run time is dominated by I/O.*

The effect can be as tremendous that 75–90% of the total running time is spent in rendering the mined frequent itemsets to text. The total output size is often on the order of several hundred megabytes of even gigabytes.

PROBLEM 6. *On 'dense' datasets FIMI benchmarks are measuring the ability of submitters to code a fast integer-to-string conversion function.*

Still, a data mining program has no real relevance if it does not output its result. Furthermore, several optimizations could be employed if it is known that the output is discarded – this would harm the fairness of comparison and make our conclusions irrelevant to practical purposes.

Recommendation 2. When comparing benchmark result of different FIM implementations, as much of the code should be identical, as possible.

PROBLEM 7. *The run time differences between different contestants are often very small.*

This is due to the nature of the mining tasks: except of very low support threshold test cases, the run time is on the order of a second, or even a fraction of a second.

PROBLEM 8. *Run time of a particular executable on a particular input is not a number but rather a distribution, i.e., it varies from run to run.*

This is true even when we provide an environment as clean as (from a user's point) possible: disable hyperthreading, use single-user mode, disable all services (including network services, and all non-vital system services), disable the GUI of the operating system, and of course not run any other programs multitasking. It is strange to see, that though almost none of the above requirements can be satisfied under Microsoft Windows OS, still many authors use it as a benchmarking environment. Although the effects may be hidden by the low precision of the OS timer, anyway.

Recommendation 3. 'Winner takes all' evaluation of a mining task on a FIMI benchmark is unfair.

PROBLEM 9. *Traditional run-time (+memory need) benchmarks do not tell us whether an implementation is better than an other in algorithmic aspects, or implementational (hardware-friendliness) aspects.*

PROBLEM 10. *Traditional benchmarks do not show whether on a slightly different hardware architecture (like AMD vs. Intel) the conclusions would still hold or not.*

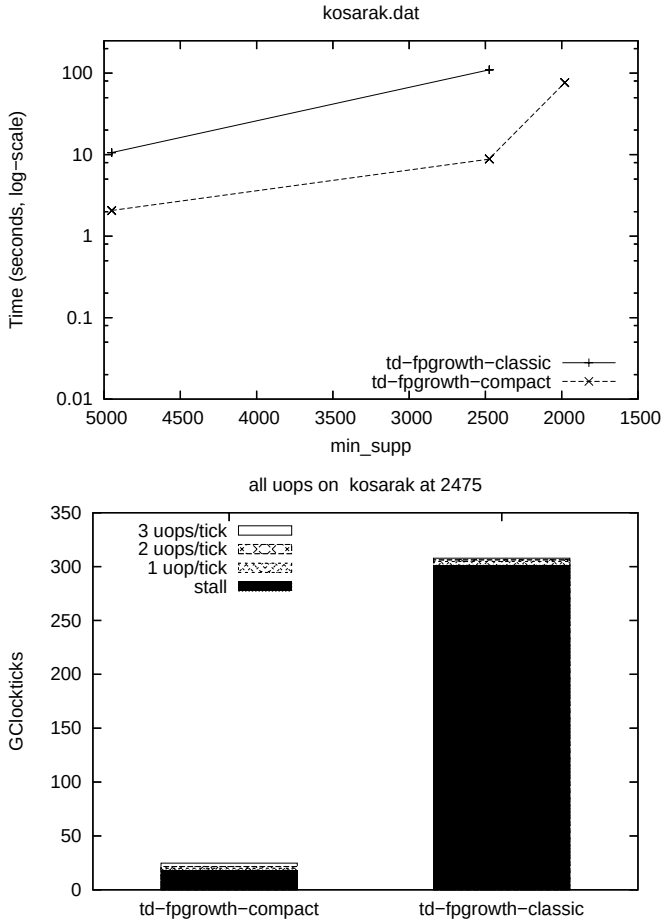
Recommendation 4. Traditional benchmarks should be executed either on virtual machines, or extended with some monitoring of hardware friendliness, most importantly the efficiency of memory hierarchy (caches), and branch prediction.

In the following sections we take the recommendations one-by-one and show our analysis system that employs them without sacrificing performance and applicability.

3. PROGRAMMING TECHNIQUES

Due to personal attitudes or in order to reach marginal performance improvement some competitive FIM algorithms are coded in plain C. The C language only partially supports modularization, hence these codes are very difficult to read, but more importantly, the modification by other researchers is error-prone and laborious. This is not a problem when a code is embedded into a system as a black-box, where FIM is regarded to be an elementary function. Nevertheless, if we want to analyze, understand the performance gains and losses, and reuse a code, then non-flexibility and the procedural approach is a serious obstacle.

In a perfect FPM world anybody who figures out a new solution to a certain subtask of an algorithm, should be able to embed it without duplication or considerable modification into the best code of the algorithm. In other words, the codes should be modular, i.e., they should fulfill object-oriented requirements. It may sound surprising but even a single algorithm can be programmed in an OO manner, where data structures and/or even functions that work on data structures are represented by different classes. For example, in our testbed the APRIORI class has separate subclasses (1.) for representing a data structure, (2.) for determining the frequent items and pairs, (3.) generating candidates, (4.) removing infrequent candidates from the data structure, (5.) caching the transactions. The subclasses further use other classes, for example the trie data structure uses a class that stores the edges, which can be a simple linked list, an indexed array (with or without offset reduction), a hybrid/double representation and so on. Each class conforms to an interface, and if anybody has a good idea that only applies to a certain part of the algorithm, then it is possible to exchange only the respective part of the code, given that the functional description and the interface is followed.



The two implementations of TD-FP-Growth[10] share 90% of the code. The variant titled 'classic' uses separately allocated nodes and pointers as links. The 'compact' variant uses an array of nodes and integers as links, ensured that nodes of the same item occupy an interval of the array.
The bottom diagram shows the total number of clockticks used by the programs on a single mining task, and (with different colors/patterns) the distribution of how many instructions were executed per clocktick. It is clearly seen that the two implementations use roughly the same number of instructions, but the classic node-based one stalls the processor 10 times more than the array-based. These stalls are caused by the memory access pattern: as memory accesses are scattered, a huge amount of cache misses delay execution. In the first case, memory access is contiguous, and the hardware prefetch mechanism loads the required data into the cache before its needed, thus no delay occurs on execution.

Figure 1: Example of difference caused by seemingly unimportant implementational details.

Object oriented approach greatly supports Recommendation 2 and 1 because every part of the code can be reused or separately exchanged and analyzed. If interfaces are strictly kept, code segments could be easily be changed, and we would also better understand why a certain optimization, that is described on algorithmic level, performs different on different implementations of the same algorithm. Although function substitution can be solved by using macros in plain C as well, this solution has many disadvantages (referred to as messy, error-prone, inelegant, etc by the programmer's community) and should be avoided in building a library for FPM.

The advantages of object-oriented approach are beyond dispute, and are required by the FPM community. Opening the black-boxes would make it possible to acquire a better understanding about the algorithms and the performance improvement techniques. Doing this, however, is not trivial, and most believe that the answer is *no* to the following question: *Can we preserve efficiency while rewriting codes so that they fulfill the object-oriented requirements?*

The answer being *no* can easily be justified, if we want to rewrite our code in the classic object-oriented way, i.e by declaring *virtual* those functions that we want to be exchangeable. Figure 2 illustrates this with a small example.

```
class Alg{
  virtual void f(args){ ... }

  void g(){
    ...
    f(values);
    ...
     }
};

class SpecAlg : public Alg{
  void f(args){ ... }
};

int main(){
  Alg* alg;
  if(cond)
    alg = new Alg(...);
  else
    alg = new SpecAlg(...);
  ...
}
```

Figure 2: OO programming with virtual function

There are two reasons for the running-time increase of this solution over the classic C implementation. First, the function call and argument passing to function f requires operations on the stack. Second, calling a virtual function means doing an indirect call, which does not only require a pointer dereference, but is also inefficient to be executed on modern processors. Furthermore, this approach greatly reduces the compiler optimization possibilities. Consequently, this kind of rewriting does not result in a code that is able to compete with the classic C counterparts. Fortunately, there exists another solution.

All drawbacks of the virtual functions can be avoided by using *templates* and *inline* functions. Figure 2 illustrates how we can keep object-oriented features and avoid virtual functions at the same time.

The compiler will actually generate two *Alg* classes that have nothing to with each other, and use no virtual functions. With the help of inline functions we avoid parameter passing and let the compiler do proper code optimization. Using this technique we can do everything that object-oriented programming requires (i.e. abstraction, data encapsulation, polymorphism, inheritance).

```
class AlgBase{
  void f(args);
};
inline void AlgBase::f(args){ ... }

class SpecAlg {
  void f(args);
};
inline void SpecAlg::f(args){ ... }

template <class T>
class Alg : public T{
  void g(){
    ...
    f(values);
    ...
  }
 };

int main(){
  if(cond)
  {
    Alg<Algbase> alg(...);
    ...
  }
  else
  {
    Alg<SpecAlg> alg(...);
    ...
  }
}
```

Figure 3: OO programming with templates and inline

The aforementioned technique has some further advantages and also minor drawbacks. For example, one can declare a boolean template argument, and it acts as a compile-time constant (equivalent to the much less elegant #define), thus any branches on that expression will be optimized away. A slight disadvantage is that the template construct results in a code bloat: all functions (with template arguments) will be compiled separately for all actually used instantiations (template parameter combinations). Furthermore, implementing template-based modularity needs careful design, and in some rare cases (such as circular references of classes) a very good understanding of the C++ programming language is necessary.

3.1 Our IO framework

As many of the authors have noticed that time spent in output routines can dominate the total running time, the FIMI contributions are rich in IO routines and tricks to speed up outputting: buffering, string representation caching, fast integer to string conversion, calling low-level buffer manipulating methods, etc. As we already mentioned the comparisons would be more fair, if all implementations used the same IO routines, ideally the fastest one. In our FIM environment we aimed to develop an IO framework that is as fast and as flexible as possible. Our goal was not only to provide a fast IO framework, but we also wished to maintain the possibility of exchanging any part of it, and therefore to be able to conduct a comprehensive set of experiments. As an example for exchangeability consider file handling in C++ which can be done in three ways: using a file descriptor and low-level OS support routines, a `FILE*` and the standard C I/O library, or the `iostream` library. In our framework the file representation is a template parameter of the class that is responsible for reading in a transaction or writing out an itemset, and the representation is wrapped by a class that exports a unified interface. As the wrapper function is inlined, this does not give any overhead compared to simply incorporating the underlying file representation method into the output class.

The most important part of the IO framework is the *cached depth-first decoder class*. Its basic idea was already used in [8] (and we suppose in [9] as well) but probably due to its technical aspect it was not described in detail in any paper. Our experiments showed that when the output is huge (for example in the case of database connect with low support threshold) then rendering the item identifiers to string and writing them out takes considerable part of the running time. Improving this task has more impact on running time than many algorithmic features.

The functionality of the cached depth-first decoder class is very simple. It maintains a stack of items, whose content, together with a support value, can be written to the file. Items can only be placed to and removed from the top of the stack. For the sake of efficiency a character buffer and a second (parallel) stack that stores positions of this buffer are maintained behind the scenery. When inserting an item, its string representation is copied to the position of the buffer given by the top element of the second stack, and the new last position is pushed on top of it. This way writing out two itemsets of size ℓ that differ only in their last item needs only two conversions from integer to string instead of 2ℓ.

Even this can be saved if the string representations of the frequent items are generated and stored before the decoder class is used. Since most FIM algorithms recode the items so that only frequent items are considered and the new codes are contiguous starting from zero or one, the lookup of the string representation of an item is done in constant time and requires insignificant memory compared to the memory need of the algorithm itself.

Please note, that this depth-first output class not only suites depth-first algorithms (eclat [11], fp-growth [6], etc. and their variants [5]) but also many other algorithms as well. For example APRIORI is a breadth-first algorithm, nevertheless outputting the result is done with a recursive traversal of a tree in a depth-first manner, no matter if the frequent itemsets are written out in candidate generation, infrequent candidate removal phase or at the end of the algorithm.

Figure 4 shows the experiments of our output tester. It generates all subset of a set of a given size, and calls the decoder class to write out the actual subset. The inverse codes of the items were generated with a random number generator. The tested routines are as follows:

- `normal-simple` is the classic method, it renders each itemset and each item separately to string. (However, the int-to-string conversion routine is the optimized one, not the very slow but generic C-library routine.)
- `normal-cache` renders each itemset separately, but caches

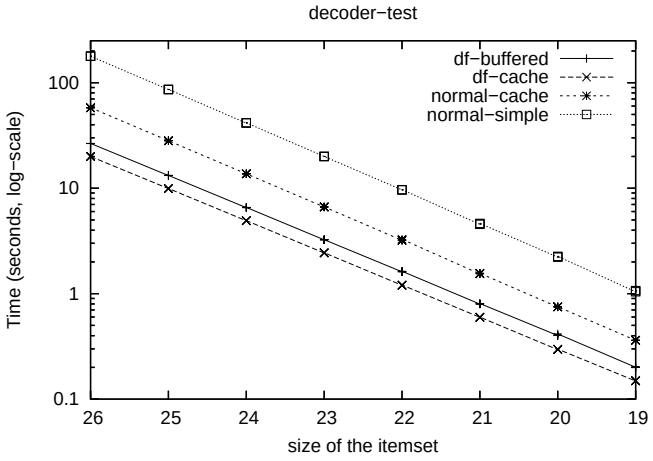


Figure 4: Performance of different output routines.
.

the item identifiers' string representation.

- `df-buffered` uses the depth-first method and reuses the string representation of the previous itemset, but when it appends the new item to the end of the line buffer, it renders it from int to string.
- `df-cache` is the implementation described above, i.e., it reuses the previous line in a depth-first approach, and uses the cached string representation of the upcoming integer.

The results show of the cached depth-first decoder. Generating an output of size 4GB (at set size 25) requires about 10 seconds. There is about a 10-fold running time difference between the original and our optimized method.

Our I/O framework contains routines for other common tasks for frequent itemset mining, such as determining item (or frequent pair) frequencies, filtering infrequent items, recoding item identifiers in frequency ascending or descending order, and inverting that recoding. Our example implementations of the next section use this library and thus only the really algorithm-specific part of the code is different. This enables a fair comparison on all datasets.

4. BENCHMARKING TECHNIQUES

First let us enumerate some desiderata about the benchmarking environment:

1. The benchmark should be **stable**, and **reproducible**. Ideally it should have no variation, surely not on the same hardware.
2. The benchmark numbers should **reflect the actual performance**. The benchmark should be a fairly accurate model of actual hardware.
3. The benchmark should be **hardware-independent**, in the sense that it should be stable against the slight variation of the underlying hardware architecture, like changing the processor manufacturer or model.

It is easy to see that *any two requirements* of the above have serious conflicts and contradictions.

It is clear that different hardware has different performance. The problem is that performance is not linear, thus it is not possible to normalize the measured values with a single performance indicator of the used hardware, and result in a unified performance metric.

PROBLEM 11. *Different algorithms/implementations may stress different aspects of the hardware. A different piece of hardware may be more advanced in one aspect, while provide lower performance in another aspect. Thus it is not possible to migrate a benchmark ranking from a particular hardware to another.*

Recommendation 5. Along benchmark results one should always describe the exact hardware used to measure the performance metrics.

Recommendation 6. Benchmark results should not form a single number based on a ranking is calculated or a graph is plotted. Instead, it should give us a slight idea about which hardware resources are stressed, so that we can extrapolate the performance indicators onto different hardware platform.

Of course this introduces extra complexity into reading benchmark results. But this is required, **performance is not as simple as 'run time in seconds'**.

4.1 Selection of benchmark platform

Basically there are three available platforms to do precise measurements including the hardware-friendliness metrics:

Virtual machine. We define an abstract machine of similar power like our current processors. (Knuth did this with defining the MIX in his famous work[7].) Then we define a cost for each operation also depending on the current state of the hardware (program execution history). This cost function should be as close to actual hardware performance, as possible. We then re-code (or compile) our FIM implementations for this processor and execute it on a proper simulator that counts the total cost and performance metrics.

There are many problems with this approach: definition of the instruction set severely influences implementation performance. The benchmark results will highly depend on the choice of cost function (and possibly its parameters if it is a parameterized cost function). Furthermore, all FIM implementations will have to be recoded or re-compiled for the new architecture. During this re-coding or re-compilation we have to re-invent all the optimization techniques that were (with a huge effort) developed for existing architectures by compiler writers. Thus there will be an additional factor, the quality of the very low-level implementation. Due to these problems, we think using a virtual machine is for the time being out of question for FIMI benchmarks.

Instrumentation. This is a technique that runs an actual executable program on a simulated processor. Each operation is separately checked and executed in the clean environment. Performance-related events can be collected and detailed, or with a proper cost function, it can be transformed to a simple 'run-time'-like aggregated performance metric.

The advantage of instrumentation is, that the environment is completely clean, the results are deterministic and reproducible. Furthermore, the parameters of the architecture (such as cache size) are arbitrarily tunable. However,

the results depend heavily on the correctness of the simulation (how well the simulated processor resembles the actual one, especially in resource availability), and even more on the choice of the event weighting/cost function. There is one more, huge disadvantage: the instrumentation framework (the simulation of the processor) is very slow, up to 100-fold increase is usual compared to the native run time of the program. This means, that even to perform a fairly straightforward benchmark suite supercomputing capacity is needed.

Run-time measurement. This employs special features of the current processors, which allow the performance of the processor to be monitored real-time. This is done by hardware **performance counters** that can be programmed to count specific performance-related events. The actual event to be counted, with the available and usable counter configurations are hardware dependent, and described by the processor manufacturer. However, the event set always includes in some form the efficiency of different architectural parts of the processor, and the possible causes of execution stalls, such as branch mispredictions, cache misses, instruction dependency stalls, etc. The usage of these counters are supported by performance optimizer software released by the processor manufacturer (like AMD CodeAnalyst^{TM} Performance Analyzer[1], or Intel VTune^{TM} Performance Analyzers[2]), or open source software (like PerfMon for linux, or the hardware-independent PAPI). Depending on the versatility and number of available counters and the events to be measured, more that a single run may be necessary to take all measurements.

The framework we release with an open source license uses the third method, the performance counters of Intel Pentium 4 and Xeon processors under Linux OS. It can be run completely user-space, only the PerfCtr patch is required to be installed on the running kernel. However, to avoid other running programs taint the benchmarks, the precautions described after Problem 8 should be adhered to as much as possible.

4.2 Sample analysis and visualization

In this subsection we give sample figures from our benchmark system, and show how the results can be interpreted. We do not go into detailed analysis neither in depth (algorithm and feature selection), nor in breadth (mining task selection), as the focus of this paper is benchmarking. We fully utilize the proposed method in an upcoming work that aims a complete in-depth analysis.

All measurements were taken on a workstation with Intel Pentium 4 2.8 Ghz processor (family 15, model 2, stepping 9) with 512 KB L2 cache, hyperthreading disabled, and 2 GB of dual-channel FSB800 main memory. The system runs a stripped-down installation of SuSE Linux 9.3, kernel 2.6.11.4-20a (SuSE version) with PerfCtr-2.6.15 patch installed.

The sample evaluation uses the dataset BMS-POS. The classic run-time diagram of the analyzed implementations is shown as Figure 5. Note that the metric shown is actually a calculated metric. We measure the number of (user-time) clockticks spent executing the current program with the PerfCtr framework. Thus the figures should be precise independently of the resolution of the OS timer. To get a run time in seconds, we simply normalized the resulting values with the nominal speed of the processor, 2.8 billion clockticks/second.

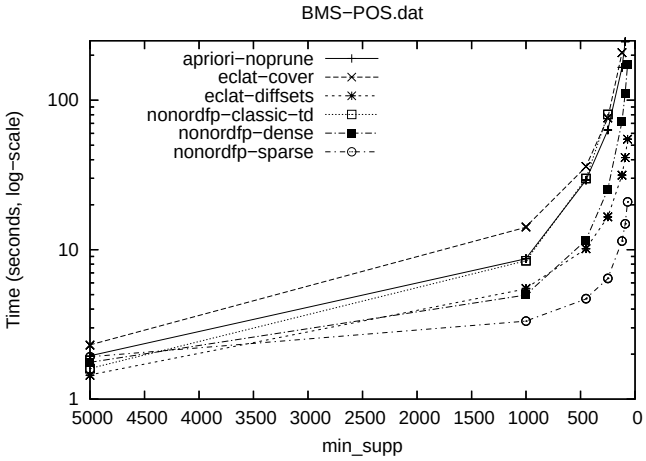


Figure 5: Run time of the example implementations on dataset BMS-POS.

Much more detailed information is available on Figure 6. While it corresponds only to one particular mining task (in this case dataset BMS-POS at support threshold of 1000), it gives us some idea about why one implementation is faster than the other. The exact numbers are listed in the Full Execution Profile on Figure 7. The following causes of performance difference should be noted:

- `nonordfp-classic-td` executes slightly less instructions than the winner, `nonordfp-sparse`. However, its performance is severely hindered by a large amount of cache misses which stall the execution unit. This shows the performance penalty of using a classic node-based representation with linked lists, against a compact array-based representation and linear scan. The non-memory related stalls are on the same order of magnitude.

- `nonordfp-dense` trails behind `nonordfp-sparse` approximately for the same reasons. Furthermore, the effect is slightly (but not considerably) enhanced by an increase in the actual number of executed instructions.

- Also the same effect can be observed when comparing `eclat-diffset` to `apriori-noprune`. While eclat practically never waits for memory, apriori has a huge penalty on cache misses.

- It is important to note that in case of the two `eclat` variants, the prefetch efficiency hides practically all effects of memory latency. This is due to the nature of the algorithm: `eclat` operates by calculating intersections (`eclat-cover`) or differences (`eclat-diffset`) of long sorted arrays. Sequential memory access has a huge advantage over scattered one.

- Nevertheless, `eclat` is not performing very well, because it has another huge disadvantage: the merge routine contains a large amount of conditional branches, which are extremely badly predictable. (These data-dependent branches are almost 50-50% random in the two directions.) This results in a stunning ¿100% overhead of bogus instructions, those instructions that were

executed on mispredicted branches and were rolled back.

- The comparison of `eclat-cover` and `eclat-diffset` gives some really interesting results. Traditionally it is believed that diffsets are well suited to dense datasets, and covers to sparse datasets, because the respective representation gives shorter lists to merge. However, in the displayed case, `eclat-cover` and `eclat-diffset` require roughly the same amount of memory accesses (30% difference at most, depending on which metric we look at), while in the run time we see over 2-fold increase. The amount of memory accesses hint that the total length of the lists to be merged/intersected is the same. The implementation using covers loses particularly on two points:

  1. The inner loop is more complex, or has to be executed more times in case of covers than diffsets (this is shown by the increased number of non-bogus instructions).
  2. There are much more data-dependent conditional branches which are badly predictable in the evaluated cover-implementation than the diffset-implementation. `eclat-cover` has over a billion mispredicted branches as opposed to any of the fp-growth implementations which has roughly 40 million.

5. CONCLUSION

We showed several problems regarding the current practices of evaluating Frequent Itemset Mining algorithms/implementations.

To work around these problems, our contribution is a sophisticated benchmark environment and an initial library for common procedures in Frequent Itemset Mining including an efficient I/O framework. This library, along with implementation modularization techniques pointed out in this paper could reach fair and in-depth comparison, and eventually help us get a better insight not only on *what* is fast for a FIM task, but also *why*.

It is important to note that the problems and possible solution techniques are not strongly connected to the task of Frequent Itemset Mining and mostly apply to other fields of applied algorithmic nature, thereby making our work relevant to a wider audience than the community of Frequent Pattern Mining.

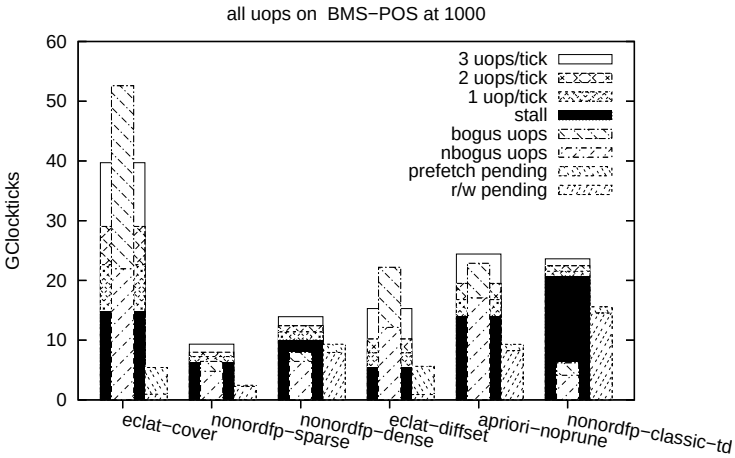
6. REFERENCES

[1] AMD CodeAnalyst^{TM} Performance Analyzer. `http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html`.

[2] Intel VTune Performance Analyzers. `http://www.intel.com/software/products/vtune/`.

[3] Bart Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.

[4] Bart Goethals and Mohammed J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining*



Instructions on how to read this diagram:
The height of the wide bars centered around the ticks show the actual run-time (the total clockticks used by the program). The colors/patterns of these bars show how well the program utilized these clockticks: the top-most part shows the amount of clockticks during which three u-ops were executed, while the bottom-most part shows the time during which the program execution was stalled for some reason (i.e., no operations were executed during that clocktick).
The narrow bars centered around the ticks show the total number of u-ops that were executed. The bar is divided into two, the upper part show the bogus u-ops, those u-ops that were speculatively executed on a mispredicted branch, and thus were rolled back. The ratio of the lower-to-upper part of this bar shows the branch prediction inefficiency.
The narrow bars beside the wide ones show the front-side bus activity, the total number of clockticks during whose at least one read/write operation was pending (i.e., data transfer time including memory latency). The upper part of these bars show the time consumed by prefetch reads (when the processor speculatively transfers data from the memory into the cache for further availability), while the lower part shows actual reads or writes. The main difference is that the delivery of data during actual reads and writes presumably stalls the execution pipeline (these are the cache misses). If the ratio of prefetch (top part) to actual wait (bottom part) is high, then a huge amount of cache misses are avoided by the prefetch mechanism, thus achieving a considerable performance gain.

Figure 6: Complex hardware-friendliness diagram of implementations.

FULL EXECUTION PROFILE

	nonordfp dense	nonordfp sparse	nonordfp classic-td	apriori noprune	eclat cover	eclat diffset
Time						
tsc avg	13,946,344,480	9,316,631,548	23,625,371,534	24,418,796,557	39,713,892,504	15,371,783,716
+-	67,047,032	13,729,232	6,967,722	88,581,299	340,152,728	203,764,648
Execution						
Instructions nbogus	5,413,034,086	3,479,408,110	2,782,247,626	12,287,347,990	20,340,660,748	10,265,793,425
Instructions bogus	1,255,528,294	1,320,346,512	1,535,708,709	4,315,850,408	28,792,606,997	8,553,364,068
uops nbogus	6,448,306,722	4,799,136,145	4,139,347,343	17,049,735,428	21,936,853,711	12,134,065,896
uops bogus	1,511,087,197	1,657,126,546	2,077,069,010	5,839,449,756	30,660,104,045	10,071,911,323
uop from TC build	4,316,441	5,555,662	4,998,851	5,408,185	395,994	3,817,971
uop from TC deliver	8,367,199,104	7,030,056,441	8,439,051,063	25,461,603,334	58,919,083,775	24,748,570,113
uop from ROM	213,755,506	327,188,333	213,407,122	434,048,470	121,036,777	120,934,817
FSB/Memory events						
Count of writes	79,566,256	11,341,904	68,774,846	49,640,164	6,812,220	6,522,120
Count of reads (incl. prefetch	168,317,318	39,971,942	251,711,898	146,056,832	100,708,228	112,688,598
Count of r+w+prefetch	246,784,892	51,862,510	320,483,650	198,141,084	107,297,474	119,180,260
Ticks of r+w+prefetch pending	9,333,003,274	2,458,365,028	15,571,190,166	9,318,981,140	5,424,657,616	5,624,807,160
Count of r+w	134,141,834	30,610,382	188,385,644	111,049,656	12,066,130	11,226,952
Ticks of r+w pending	7,947,664,718	2,244,115,972	14,536,000,500	8,239,955,570	970,765,866	900,680,634
store operations (nbogus)	524,358,888	701,508,105	647,941,634	2,569,378,903	455,990,928	344,428,466
load operations (nbogus)	1,379,447,092	1,646,228,087	1,301,423,517	5,118,315,577	4,422,871,852	3,140,679,837
128bit mmx operations	123,298,382	5,068,819	37,679	0	0	0
Branches						
Function calls	13,896,223	16,792,119	13,878,730	134,797,905	5,860,583	5,602,314
Indirect branches	16,747,576	21,078,499	16,725,092	140,985,437	7,545,318	7,443,164
Total conditional branches	1,345,731,870	420,551,853	477,601,235	2,118,233,826	6,991,388,257	3,719,260,359
Mispred cond branches	34,929,891	36,802,712	40,132,062	141,613,646	1,088,360,184	321,259,792
Mispred noncond branches (?)	319,223	453,523	298,023	380,772	2,369	4,861
Stall causes						
MemoryOrderBuffer load (pcs)	830,309,349	202,608,169	5,560,914,552	785,927,299	39,497,734	37,816,201
Lack of store buffer	3,139,504,337	1,444,306,416	885,554,235	2,820,176,466	68,776,266	67,397,197
Memory cancel	2,364,288,764	869,214,080	801,985,760	361,524,915	56,222,156	56,156,928
Split memory access	28,628	28,641	28,637	5,759	871	900
WC buffer evictions	161,435,891	290,548,591	308,327,051	384,695,637	50,740,548	50,014,831
L1 read miss	172,012,034	178,449,046	250,334,607	369,703,235	360,009,247	182,271,597
L2 read miss	57,447,430	32,716,944	129,914,412	76,089,398	25,436,079	25,290,465
Execution histogram						
TickOf uop stall	10,021,440,271	6,240,622,748	20,671,650,354	13,917,119,211	14,801,445,447	5,414,343,311
TickOf uop 1	1,371,043,060	1,023,246,833	859,892,764	2,858,956,464	7,830,923,868	2,458,339,586
TickOf uop 2	1,011,261,551	736,536,448	931,457,183	2,736,975,862	6,418,788,673	2,317,353,459
TickOf uop 3	1,520,275,270	1,323,090,135	1,159,475,923	4,908,240,355	10,668,342,420	5,103,006,616
TickOf nbogus uop stall	10,688,282,384	6,927,514,440	21,520,910,519	16,423,833,026	28,349,705,419	9,709,376,650
TickOf nbogus uop 1	1,206,548,172	891,827,770	736,711,385	2,430,209,501	4,627,072,599	1,607,600,195
TickOf nbogus uop 2	845,852,944	605,173,766	691,985,048	2,082,264,477	2,918,477,521	1,401,800,070
TickOf nbogus uop 3	1,183,336,652	898,980,188	672,869,272	3,484,984,888	3,824,244,869	2,574,266,057
Percent uop stall	71,972	66,935	87,508	56,987	37,264	35,404
Percent uop 1	9,846	10,975	3,640	11,706	19,715	16,074
Percent uop 2	7,262	7,899	3,943	11,207	16,160	15,153
Percent uop 3	10,918	14,191	4,908	20,098	26,859	33,368
Percent nbogus uop stall	76,761	74,302	91,103	67,252	71,374	63,489
Percent nbogus uop 1	8,665	9,565	3,118	9,951	11,649	10,512
Percent nbogus uop 2	6,074	6,490	2,929	8,526	7,347	9,166
Percent nbogus uop 3	8,498	9,642	2,848	14,270	9,628	16,832

Figure 7: Sample execution profile for BMS-POS dataset at support threshold of 1000.

.

Implementations (FIMI'03), volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[5] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.

[7] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[8] Balázs Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 1. November 2004.

[9] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 1. November 2004.

[10] Ke Wang, Liu Tang, Jiawei Han, and Junqiang Liu. Top down fp-growth for association rule mining. In *PAKDD '02: Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pages 334–340, London, UK, 2002. Springer-Verlag.

[11] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 12–15 1997.

APPENDIX

A. SHORT INTRODUCTION TO MODERN PROCESSOR HARDWARE

This section is to give a very short introduction into those properties of modern computing hardware which any programmer optimizing for performance should be aware of.

It is not true that modern processors can execute an instruction (or several instructions) in a single clocktick. The circuits needed to execute an instruction are very deep and complex, but only simple and shallow circuits are able to reach clockspeeds of several GHz. Instruction execution is thus divided into several clockticks (12 for AMD Athlon and 20 for Intel Pentium 4), but to keep the hardware utilized, each of these *pipeline stages* can be processing a different instruction at a particular clocktick. So although as much as 20 clocks might be needed for a particular instruction to finish execution, still (theoretically) in every clocktick an instruction could be finished, giving an average throughput of 1 instruction per clocktick. Furthermore, the pipeline stages are designed so and execution units are multiplied so that theoretically on average more than one instruction can be executed per clocktick.

This pipelined architecture of processors raises several issues, some of which programmers should be aware of, while others should be taken into account by compilers.

Complex instruction set. x86 processors belong to the category of CISC (complex instruction set computer). This means that the elementary instruction the processor can execute can include many operations, like loading a data element from memory, adding another data element to it, and storing back the result into the same memory address. These complex instructions would require more advanced treatment from processors than simple ones. To simplify processor design, instructions are decomposed into one or more μ-operations (or u-ops) and these u-ops are fed into the execution pipeline. Different kind of u-ops use different execution hardware and can be executed concurrently (categories are like: loads, stores, integer arithmetics, floating point/multimedia arithmetics); some units may even be multiplied, like two or even three independent integer ALU/processor.

Data dependency. If there is an instruction in the program that depends on the output of a previous instruction, then its execution cannot begin until that previous instruction is finished.

Branch prediction. When the program reaches a conditional branch, the direction of control flow cannot be known until the branch condition is evaluated. Thus there can be no instructions fed to the pipeline until the condition instruction finishes its execution. This results in wasted processor resources. To enable maximum instruction throughput, processor hardware predicts the condition outcome, and feeds the instructions of the respective branch into the pipeline. However, these instructions will not be committed to the architectural state until the branch condition is evaluated, and if the prediction turns out to be false, these instructions are rolled back, and the pipeline begins with executing the instruction on the correct branch. Branch prediction is based on the previous encounters of the same branch, thus typical branches like loop conditions can be well predicted, as they usually branch towards the inside of a loop, and a misprediction occurs only at the relatively rare condition of exiting that loop.

Another factor that has considerable effects on execution performance is the **efficiency of memory hierarchy**. The basic problem is that processor speeds have increased much faster than main memory access speeds. The difference is so much that today up to 100 clockticks are necessary to load a value from the main memory. Furthermore, memory access is effective only in relatively long bursts of reads and writes. To minimize the latency and data transfer speed effects, processors incorporate relatively small but very fast memories that **cache** the contents of the main memory. There are at least two levels of such cache, reading a data element from the first level (L1) cache takes usually one clocktick, while reading a data element from the second level (L2) cache may take a few clockticks. Typical sizes of these caches are as fol-

lows: few ten KB for L1 cache (16–32 KB in Intel Pentium 4, 64–128 KB in AMD processors), while 512 KB–2 MB for L2 cache. Non-mainstream (value market) processors may have considerably smaller caches.

When a data that is loaded was recently used, there is a high chance of finding it in the cache memory. However, when the program has to process a large amount of data (sequentially), then almost all data accesses will be cache misses, thus the execution engine will wait for the memory, then process the data segment it got, then issue the next memory read request, wait for the memory, etc. The memory interface and the execution units will be alternately idle. To overcome this, the **prefetch** mechanism was introduced to make the memory interface and execution unit concurrently busy. Prefetch operates by loading the data for the next iteration of the processing loop in advance into the fast cache memory so that it will be instantly available when requested by the execution engine. This is implemented in hardware for well predictable memory reads (namely sequential reads), while the programmer has to take care for it in non-sequential memory accesses.

Out of order execution. When an instruction currently in the execution pipeline requires data from the memory that is not found in the cache, it has to wait until the data is loaded from main memory. To keep the execution units of the processor busy, the processor looks ahead for other instructions that have all necessary input data available so that their execution can proceed. Thus only the instructions that depend either on the unavailable data or the result of such instructions are hindered in execution.

Summary. We say that the processor (or the execution pipeline) is *stalled* when there is no instruction whose execution could proceed. This can be caused by various reasons: instructions may depend on the completion of another instruction (by using its output data as input); instructions may require data from main memory that is not available in the fast cache memory, thus they have to wait until the memory access cycle completes; there may have been a mispredicted conditional branch when the speculatively executed instructions had to be flushed from the pipeline; or there may be another resource constraint (lack of some sort of temporary buffers, like store buffers, register renaming buffers, etc.). Execution speed of an algorithm (instruction flow) does not only depend on the number of instructions it contains, but also on the count and severity of the stalls that the particular instruction flow causes on the hardware used.