Heavyweight Pattern Mining in Attributed Flow Graphs

Carolina Simões Gomes
Intuit Canada
Edmonton, AB, Canada
Email: carolina.sgomes@gmail.com

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
Email: amaral@cs.ualberta.ca

Joerg Sander
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
Email: jsander@ualberta.ca

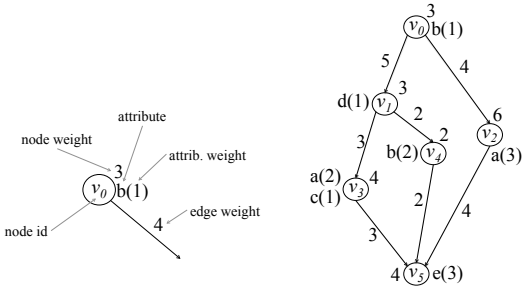
Joran Siu
IBM Canada Software Laboratory
Markham, ON, Canada
Email: joransiu@ibm.ca

Li Ding
Amazon Canada
Toronto, ON, Canada
Email: ldingca@yahoo.com

***Abstract*—This paper defines a new problem - heavyweight pattern mining in attributed flow graphs. The problem can be described as the discovery of patterns in flow graphs that have sets of attributes associated with their nodes. A connection between nodes is represented as a directed edge. The amount of load that goes through a path between nodes, or the frequency of transmission of such load between nodes, is represented as edge weights. A heavyweight pattern is a sub-set of attributes, found in a dataset of attributed flow graphs, that are connected by edges and have a computed weight higher than an user-defined threshold. A new algorithm called AFGMiner is introduced, the first one to our knowledge that finds heavyweight patterns in a dataset of attributed flow graphs and associates each pattern with its occurrences. The paper also describes a new tool for compiler engineers, HEPMiner, that applies the AFGMiner algorithm to Profile-based Program Analysis modeled as a heavyweight pattern mining problem.**

I. Introduction

Flow graphs are an abstraction used to represent elements (*e.g.,* digital data, goods, electric current) that travel through a network of nodes (*e.g.,* computers, physical locations, circuit parts). Flow graphs are often used in the modelling of logistics problems. An attributed flow graph (AFG) is a single-entry/single-exit graph with a *source node* that has no incoming edges and a *sink node* with no outgoing edges.The flow starts in the source node and is directed to the sink node. All other nodes in the AFG, if they exist, must have at least one incoming and one outgoing edge. In addition, AFGs have attributes in their nodes, representing *e.g.* types of goods at a given network node, and weights in nodes and edges, representing *e.g.* the amount of goods stored at that node and flowing between any two nodes. An example of an AFG is shown in Figure 1.



(a) Elements of an AFG (b) Example

Fig. 1. Example of attributed flow graph.

Existing graph-mining algorithms are all severely limited in their applicability to AFGs. None of them is able to find general sub-graph patterns in AFGs that take into account multiple node attributes as well as weights in nodes and edges. The only work on mining patterns in AFGs is the FlowGSP algorithm proposed by *Jocksch et al.* [1]. However, FlowGSP can only find sub-path patterns, while the algorithm described in this work finds all the patterns that FlowGSP finds and also finds additional patterns that encompass multiple sub-paths in the dataset.

This paper presents AFGMiner, the first algorithm, to the best of our knowledge, to address the problem of mining AFGs for general sub-graph patterns. AFGMiner takes as input a set of AFGs and a support measure, $MinSup$, that takes into account the attribute weights, node weights and edge weights of the occurrences of patterns. AFGMiner returns all patterns P, called *heavyweight patterns*, whose support $MinSup(P)$ is higher than a threshold. This threshold is user-specified as commonly assumed in pattern-mining approaches. The main contributions of this paper are as follows:

1) Definition of the Attributed-Flow-Graph Mining problem to find Heavyweight Patterns.
2) Development of different versions of AFGMiner, an algorithm that mines for heavyweight patterns in attributed flow graphs, including a parallel version with a work-distribution heuristic to maintain workload balance between multiple threads.
3) Development of HEPMiner, a tool that automates the analysis of hardware-instrumented profiles, as an application of AFGMiner. Patterns discovered by HEPMiner indicate potential, non-obvious, opportunities for compiler and architecture-design improvements.

4) Complexity and performance analysis of AFGMiner, comparison against the FlowGSP algorithm and qualitative analysis of patterns found by HEPMiner when applied to the DayTrader benchmark running on IBM's WebSphere Application Server [2].

II. Problem Definition

An attributed flow graph (AFG) G, that belongs to a dataset DS of AFGs, is defined as $G = \langle V, E, A, \alpha, w^E, w^V, w^{A,V}, l \rangle$ where:

1) V is a set of nodes.
2) E is a set of directed edges (v_a, v_b) where $v_a, v_b \in V$.
3) A is the set of all possible attributes.
4) α is a function mapping nodes $v \in V$ to a subset of attributes, *i.e.,* $\alpha(v) = \{a_1, ..., a_k\}$, $a_i \in A$, $1 \leq i \leq k$.
5) w^E is a function assigning a weight $w \in [0, 1]$ to each edge $e \in E$ so that $\Sigma_{e \in E_{DS}} w^E(e) = 1$ where E_{DS} is the set of edges from all AFGs that belong to DS, *i.e.,* w^E is normalized over DS.
6) w^V is a function assigning a weight $w \in [0, 1]$ to each node $v \in V$ so that $\Sigma_{v \in V_{DS}} w^V(v) = 1$ where V_{DS} is the set of nodes from all AFGs that belong to DS, *i.e.,* w^V is normalized over DS.
7) $w^{A,V}$ is a function assigning a weight $w \in [0, 1]$ to each attribute a of each node $v \in V$, $a \in \alpha(v)$, with the constraint that $w^{A,V}(a) \leq w^V(v)$.
8) l is a function assigning a unique integer label $l(v)$ to each node $v \in V$, according to a depth-first traversal of the graph. $l(V)$ denotes the result of applying the labeling function to all nodes in V. Given an edge (v_i, v_j), the edge is called a *forward* edge if $l(v_i) < l(v_j)$; otherwise, if $l(v_i) \geq l(v_j)$ the edge is called a *backward* edge or *back-edge*; the node v_i is called the edge's *from-node* and the node v_j is called the edge's *to-node*.
9) G is a flow graph, *i.e.,* for any node v^* it holds that $\Sigma_{(x,v^*) \in E} w^E((x, v^*)) = \Sigma_{(v^*,y) \in E} w^E((v^*, y))$.

A directed graph $g = \langle V_g, E_g, A_g, \alpha_g, w_g^E, w_g^V, w_g^{A,V}, l_g \rangle$ is a *sub-graph* of G, if V_g is a subset of the nodes in G, E_g is a subset of the edges in G that exist between nodes in V_g, A is a subset of the attributes in G, and the functions $\alpha_g, w_g^E, w_g^V, w_g^{A,V}, l_g$ are the corresponding functions in G, restricted to the corresponding domains of nodes, edges and attributes in g.

A *pattern* is a graph $P =< V_P, E_P, A, \alpha >$, where V_P is a set of nodes with attributes from A associated to them by α, and E_P is a set of directed edges; note that a pattern does not include weights. A pattern is an abstraction in which the nodes represent sets of attributes that are relevant under a *support criteria*, and the edges represent an ordering for such attribute sets.

An *occurrence* g of a pattern P is a sub-graph of G that "matches P with a maximum gap size k_{max}"; *g matches P with a maximum gap size k_{max}* if there is a function m that maps every node v in P to a node in g so that for every edge (v_i, v_j) in P, there is a path in g that starts at node $m(v_i)$ and ends at node $m(v_j)$, and that has at most k_{max} nodes between $m(v_i)$ and $m(v_j)$. This definition of an occurrence of a pattern allows a number k_{max} of "mismatches" or "don't-care nodes" when mapping each node in the pattern P to a sub-graph in G, and k_{max} can be chosen by a user in order to allow approximate matches in applications where such an approach is useful. A k_{max}-match of a pattern P is an occurrence g_P of P that was found by using a maximum gap size of k_{max}. As an example, in Figure 2 we have a pattern on the left and an AFG in which a 0-match, an 1-match and a 2-match of the pattern can be found.

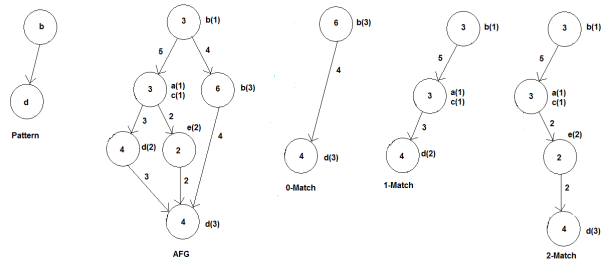


Fig. 2. Example of k-matches of a pattern.

Given a dataset DS of AFGs, a *support measure* $MinSup$ that defines the minimum support of patterns P in DS, the maximum number of attributes allowed in any given pattern node $MaxAttrs$ and a threshold value T. The problem we address in this paper is to find all patterns P with at most $MaxAttrs$ attributes in each node, for which $MinSup(P) > T$. We call the patterns for which $MinSup(P) > T$ *heavyweight patterns*, and we call the problem itself **Heavyweight Pattern Mining in Attributed Flow Graphs** (for short, *heavyweight pattern mining*).

III. The AFGMiner Algorithm

AFGMiner generates and tests candidate patterns with increasing number of edges, then extends those patterns considered heavyweight. It generates candidate patterns of k edges, starting with $k = 0$ (patterns composed of a single node and no edges), and searches for occurrences of such patterns in the dataset by using a sub-graph isomorphism detection algorithm, modified to take attributes into consideration [3]. The prototype implementation of AFGMiner adapts VF2 [4], an algorithm that is faster than alternatives for graphs that are relatively regular, have a large number of nodes and whose nodes have small valency [5], such as the ones found in the case study. Each occurrence found has its node weight and edge weight support values computed, and, when no more occurrences of a pattern are present in the dataset, the support value for the pattern itself is computed by aggregating the support values of its occurrences. If the support value for the pattern is higher than an user-defined threshold, the pattern is heavyweight. It is then output to the user and later extended into candidate patterns with an additional edge, a process called *edge-by-edge pattern extension*. If the pattern is not heavyweight, it is discarded. Edge-by-edge pattern extension works by adding to a pattern either: (i) an edge that connects two of its nodes; (ii) or an edge that connects one of its nodes to a new node called the *extension node*. A pattern that generates other patterns by extension is called a *parent pattern*, while the generated patterns are *child patterns*.

A. Canonical Labeling

Two different patterns, when extended, may generate child patterns that are isomorphic. Thus, redundancy should be detected using the well-known concept of *canonical labelling* [6]. The idea is to map each sub-graph pattern to an identifier string called *DFS Code* after labelling its nodes and edges. DFS Codes can then be lexically ordered in such a way that, if two sub-graphs are isomorphic to each other, they provably have the same minimum DFS Code. The rules that define how to sort DFS Codes depend on the types of graphs being mined [3].

B. Support Value Policy

AFGMiner adopts an *anti-monotonic* support-value policy to enable the pruning of its search space. Under this policy, the support value of a pattern is always lower than or equal to the support value of any of its ancestor patterns. As a consequence, if a pattern is not heavyweight, none of its descendants can be heavyweight. Therefore, all patterns that do not meet a minimum support criteria can be discarded. The support-value policy works as follows. For each occurrence g of a pattern p, two values are calculated: $S_n(g)$, the weight of the attribute with minimum weight amongst all attributes associated with nodes of g; and $S_e(g)$, the minimum edge weight amongst all edges of g. The $S_n(g)$ of all occurrences of p found in DS are then added up, resulting in $S_n(p)$, the node-weight support of p. The $S_e(g)$ of all occurrences of p found in DS are also added up, resulting in $S_e(p)$, the edge-weight support of p. The support value for p is $S_m(p)$, the maximum between $S_n(p)$ and $S_e(p)$. $S_m(p)$ is compared against the support threshold to decide whether p is a heavyweight pattern. The support-value policy selected for AFGMiner is anti-monotonic because only the minimum edge weight and the minimum node attribute weight of each occurrence are used in the computation.

C. Generation of Candidate Patterns

AFGMiner mines for candidate patterns with an increasing number of attributes in their single node (in the case of 0-edge patterns) or in their extension node (in the case of k-edge patterns with $k > 0$), starting with a single attribute, and then combining attributes such that an attribute set is used to generate a pattern only if all of its sub-sets generated heavyweight patterns. This process is called *attribute-set growth*. A set of patterns that have the same number of edges is called a *generation*. Patterns of a certain generation are only created and mined after all the heavyweight patterns of the previous generation have been found. This is important because it allows the algorithm to use only the A_k set of distinct attributes present in the $(k-1)$-th generation to compose the patterns of the k-th generation, thus restricting the number of candidate patterns produced.

IV. Algorithm Improvements

The previous section described the original version of AFGMiner, called **AFGMiner-iso** (*iso* stands for isomorphism detection). AFGMiner-iso visits all nodes in the dataset for every pattern searched, making it potentially slow when analyzing large datasets composed of thousands of AFGs with hundreds of nodes each, as in the case study presented in this work. Another version of AFGMiner, **AFGMiner-locreg**, addresses this performance issue. It uses the concept of location registration. That is, it creates a complete mapping between a candidate pattern p and each of its occurrences g. If p is found to be heavyweight, this mapping is kept, otherwise it is discarded. Then, when a heavyweight pattern p is extended into its child patterns c, in order to find occurrences of c the algorithm only checks the mappings between p and its occurrences g. In order to generate c, p has one of its nodes, v, extended by adding to it an edge e, and may also have an extension node q connected to e. The idea of location registration is to check each mapping between p and occurrences g for: (i) the node v_g in g that corresponds to v; (ii) if v_g is connected to an edge e_g that corresponds to e and (iii) in case c was extended from p by adding an extension node, check if e_g connects a node q_g, corresponding to node q, to v_g. If the algorithm is able to find appropriate v_g, e_g and q_g attached to the occurrence g, then the sub-graph that is composed of g plus e_g and q_g is an occurrence of c.

A. A Parallel Implementation of AFGMiner

A parallel version of AFGMiner, **p-AFGMiner**, benefits from the multiple cores available in many computing systems. p-AFGMiner initially distributes A_0 among threads, and, as heavyweight patterns are found, local queues of parent patterns Q_{TL} and local sets of attributes A_{TL} are formed by each thread. Once all threads are synchronized, all Q_{TL}s are unified into a global queue Q, and all A_{TL}s into a global A_{k+1} set of attributes. If Q is not empty, the algorithm starts processing the next generation. The dataset of AFGs, DS, is read-only during the entire run of p-AFGMiner, allowing the maintenance of thread-local versions of variables and thus reducing synchronization. An important challenge in the implementation of this parallel version is the workload distribution to improve load balancing. The number of occurrences of the parent of a pattern p in DS is the most important factor determining the time required to search for occurrences of p. A reasonable heuristic tries to balance the number of parent-pattern occurrences assigned to each thread.

The pattern-distribution heuristic created for p-AFGMiner sorts the m patterns in Q by decreasing order of the number of parent-pattern occurrences. The heuristic then does a round-robin assignment of patterns to threads following an increasing order for the patterns with an even position in the sorted Q and in decreasing order for patterns with an odd position in the sorted Q. This simple $O(n)$ heuristic is effective for a moderate number of threads and for limited variations in the number of parent-pattern occurrences. Experimental evaluation revealed that this workload-distribution heuristic lowered the execution time of p-AFGMiner, on average, by 6% when compared with a naive workload distribution method that simply distributes patterns among the threads without any sorting.

V. Case Study: Using AFGMiner for Program Analysis

The **Profile-based Program Analysis** (PBPA) problem, a common challenge among compiler engineers and computer architects, is defined as follows. Given a profile *Prof* obtained from an execution of a computer program, automatically discover operation patterns in the execution of *Prof* that, in aggregation, account for a sufficiently large fraction of

the program's execution time. If PBPA is properly handled, developers are then able to focus their optimization efforts on those areas in the program code that correspond to occurrences of the relevant operation patterns. In order to solve PBPA, we convert it to a heavyweight pattern mining problem, by modeling the program as a dataset of attributed flow graphs named *Execution Flow Graphs* (EFGs). EFGs are control flow graphs with added attributes (hardware-related events captured by performance counters) and associated weights (CPU cycles spent on instructions and events). Our tool HEPMiner then uses the AFGMiner algorithm to find *heavyweight execution patterns* (HEP): sets of hardware-related events associated with assembly instructions that were executing when the events happened [3].

A. Sub-Graph Mining in Bounded Treewidth Graphs

Tree-width is a measure of how similar to a tree a graph is. It is a very useful property because several NP-hard problems on graphs become tractable for the class of graphs with bounded tree-width, including sub-graph isomorphism detection and, as a consequence, frequent sub-graph mining of connected graphs [7]. *Horvarth and Ramon* discovered a level-wise sub-graph mining algorithm that lists frequent connected sub-graphs in incremental polynomial time in cases when the tree-width of the graphs being mined is bounded by a constant [7]. In addition, *Thorup* shows that graphs representing the control flow of structured programs (*i.e.,* CFGs) have tree-width of at most six [8]. Since EFGs have the same topology as CFGs, they also have bounded tree-width. Thus, when applied to the PBPA problem, AFGMiner runs in incremental polynomial time because the problem being solved is fundamentally finding frequent connected sub-graphs in a dataset of EFGs. The addition of weights in nodes and edges and weighted attributes to nodes does not change the complexity of the algorithm, but the generation of attribute sets of increasing size when creating new candidate patterns could. However, the number of attributes that an extension node of a k-edge candidate pattern with $k > 0$, or that the single node of a 0-edge candidate pattern, can have is bounded by the size of A, *i.e.,* by the number of possible attributes that each pattern node may contain. As a consequence, the attribute-set growth component of the algorithm has a constant complexity, while the sub-graph mining component has incremental polynomial complexity.

VI. Performance Evaluation Methodology

The experimental evaluation of the algorithm in the context of HEPMiner was performed on an Intel Core 2 Quad CPU Q6600 machine, running at 2.4 GHz and with 3 GB of RAM. It uses profiles from the DayTrader Benchmark, running on IBM's WebSphere Application Server, an IBM z196 mainframe [9] and JIT-compiled using the IBM Testarossa JIT compiler. The WebSphere Application Server is a Java® Enterprise Edition (JEE) server [2]. It has a very flat profile, with its execution time spread relatively evenly over 2,566 methods, as is typical of large business applications.

Three important parameters in the experiments are: the minimum support threshold (MinSup) and the maximum allowed size of the attribute set in each candidate pattern node (MaxAttrs), both described in Section II; and the *Minimum Hotness Method* (MMH) value. The MMH is calculated by dividing the sum of all CPU cycles associated with each one of DayTrader's profiled methods by the cycles associated with the entire program run. This parameter is used in the experiments simply to limit the methods analyzed by the algorithm to those that contribute most significantly to total program run-time, and are thus more likely to contain patterns of interest to compiler engineers. For experiments **A**, **B** and **C**, the MMH is kept at 0.001(*i.e.* in DayTrader, the 278 hottest methods are selected for mining). For experiment **D**, the MMH has values 0.001 (278 methods), 0.003 (56 methods) and 0.005 (23 methods). For all experiments except **A**, MaxAttrs is kept at 5.

Experiment **A** compares the run-times of AFGMiner-locreg with respect to changes in MaxAttrs. MinSup is kept at 0.001. Increasing MaxAttrs potentially causes more candidate patterns to be generated, which is why modifying this parameter is a way of controlling the memory consumed by the algorithm and also its run-time. *Experiment* **B** compares the run-times of AFGMiner-locreg with respect to changes in the number of EFG nodes visited by the algorithm. The number of EFG nodes visited is controlled by changing MinSup. *Experiment* **C** compares run-times of AFGMiner-locreg and p-AFGMiner with 2, 4, 6 and 8 threads, by changing MinSup. *Experiment* **D** compares run-times of AFGMiner-locreg and p-AFGMiner with 2, 4, 6 and 8 threads by changing MMH. MinSup is kept at 0.001. Patterns found by AFGMiner-locreg were also compared to the ones found by the FlowGSP algorithm. FlowGSP was modified to support variations in MMH, MinSup and MaxAttrs, and run on DayTrader methods with the same parameter values used for Experiment **B** above. In addition, expert compiler engineers from IBM's JIT Compiler Development team verified the usefulness of patterns identified by the HEPMiner tool, as described in Section VIII.

VII. Performance Evaluation

In the experiments using HEPMiner, AFGMiner-locreg was always faster than AFGMiner-iso. As an example, when running with MMH and MinSup of 0.001, AFGMiner-iso took approximately 11 hours to complete the mining process, while AFGMiner-locreg took 40 minutes and p-AFGMiner with 8 threads took 15 minutes. The dramatic decrease in run-time when comparing AFGMiner-locreg to AFGMiner-iso is expected because location registration decreases the number of dataset sub-graphs that must be tested for isomorphism with candidate patterns.

We also compared patterns found by AFGMiner and FlowGSP. Figure 3 shows the number of output patterns for different MinSup values. As expected, AFGMiner found all patterns found by FlowGSP, but also found additional patterns composed of multiple sub-paths. For flat profiles, the trend is for AFGMiner to find many more patterns than FlowGSP as the MinSup is decreased, with the sequential patterns found by both being the parent patterns of the sub-graph patterns found exclusively by AFGMiner.

Relevant observations from the experiments are:

1) The MaxAttrs value is not as relevant a factor in the run-time performance of AFGMiner as the number of EFG nodes visited during the mining process.

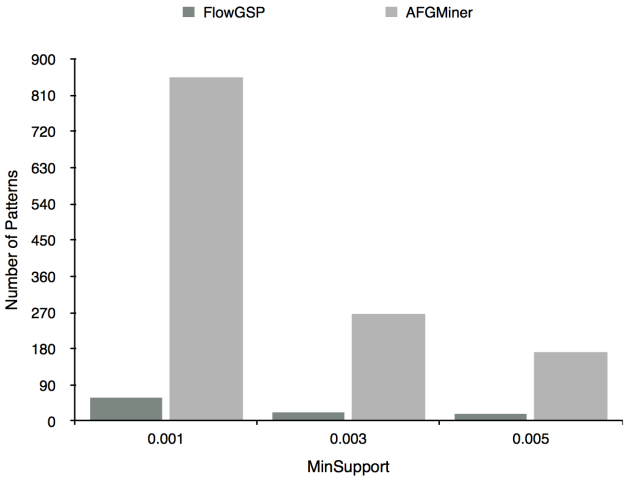


Fig. 3. Patterns found by FlowGSP and AFGMiner

2) The run-time of AFGMiner increases moderately with the number of EFG nodes visited.
3) Although increasing the number of threads logically decreases run-time, there is a diminishing effect as MinSup/MMH increase, as time spent on data loading and temporary bookkeeping of patterns and pattern occurrences - both done serially by p-AFGMiner - start dominating the total run-time.

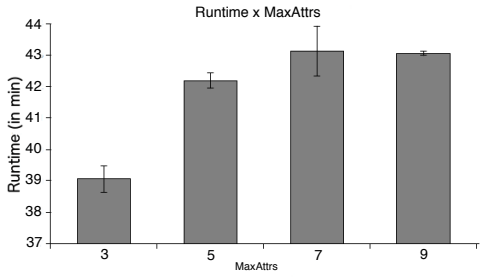


Fig. 4. Experiment A.

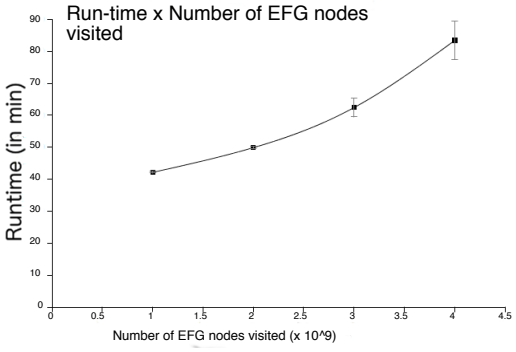


Fig. 5. Experiment B.

VIII. Qualitative Analysis

This section discusses the patterns obtained by both HEP-Miner. Results were analyzed by compiler engineers from the IBM Canada Software Laboratory. They found HEPMiner to be a useful tool and were able to not only validade results according to previous knowledge about the benchmark, but also to make new observations about the run-time behavior of DayTrader when executed by the z196 hardware. The observations are summarized below.

1) AFGMiner found heavyweight patterns that contain an edge from a branch instruction leading to a node that has instruction cache misses as one of its attributes. Reviewing the occurrences of such patterns shows that this edge represents the taken path from the branch, which confirms expectation that instruction cache misses should be observed only on the taken branch target.
2) From prior analysis, the experts at IBM know that 30% of overall CPU cycles in JITted code are assigned to method prologues, and 40% of instruction cache misses are correlated with method prologues. This was confirmed in the results output by AFG-Miner.
3) An attribute that represents a non-taken, correct-direction branch prediction was found to be dominant by AFGMiner. The fact that this attribute shows up highlights that the JIT compiler performs well when ordering basic blocks to optimize for fall-through paths. This discovery led to the creation of performance counters that take into account the ratio

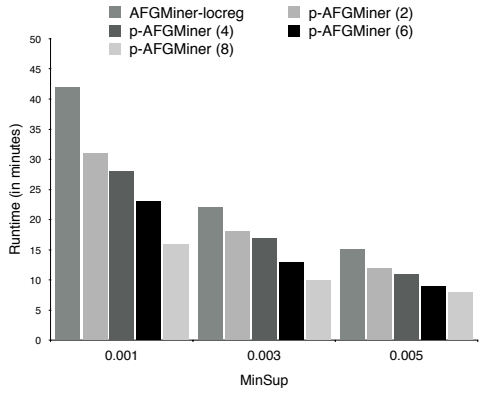


Fig. 6. Experiment C.

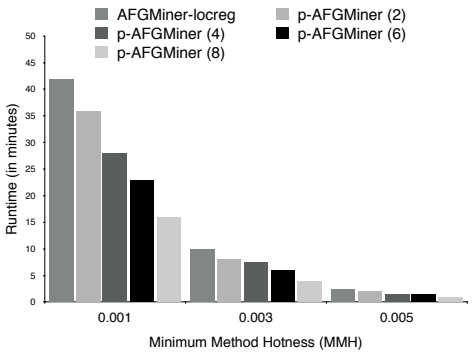


Fig. 7. Experiment D.

of taken and non-taken branches.

4) The relative support values for a sequence of three attributes, present in several patterns, were found to be relevant by the IBM developers. The attributes are an address-generation interlock, followed by a directory or data cache miss, and then an instruction used to load a compressed referenced field from an object. This discovery led to the implementation of the pattern in the compiler's instruction scheduler in order to reduce its negative effect on the run-time of future compiled applications.

IX. Related Work

FlowGSP is a sequential-pattern mining algorithm that finds patterns whose occurrences are sub-paths of AFGs [10], [1]. AFGMiner differs from FlowGSP in that it is able to find not only sequential patterns, but also patterns whose occurrences are sub-graphs of AFGs. AFGMiner takes less time than FlowGSP to find each pattern. In addition, AFGMiner is able to map each pattern to all its occurrences and output this mapping to the user.

gSpan is a classic sub-graph mining algorithm. The main difference between AFGMiner and gSpan is that AFGMiner is able to handle multiple node attributes and uses breadth-first search with eager pruning when generating candidate patterns, while gSpan follows a depth-first approach [6]. *Fast Frequent Subgraph Mining* (FFSM) is another well-known sub-graph mining algorithm for undirected graphs, and its novelty was the introduction of embeddings that make the mining process faster. AFGMiner-locreg also uses embeddings, but has to record the complete mapping between sub-graph patterns and occurrences, making them more memory-consuming. Gaston is a more recent frequent path, tree and graph miner integrated into a single algorithm. In contrast to AFGMiner, however, Gaston only mines for patterns in non-attributed, undirected and unweighted graphs [11]. *Horvart et al.* describes a sub-graph mining algorithm for outerplanar graphs [12]. Similarly to AFGMiner the algorithm runs in incremental polynomial time due to outerplanar graphs being similar enough to trees.

A work related to HEPMiner is that of *Dreweke et al.*. It uses sub-graph mining as part of a code transformation to extract code segments [13]. HEPMiner differs from this work in that it is an external performance analysis tool that helps compiler developers to reach conclusions about the performance of applications of interest, and detect improvement opportunities.

Conclusion

This work defined heavyweight patterns and the problem of Heavyweight Pattern Mining. It presented AFGMiner, a heavyweight pattern mining algorithm that is generic enough to be applied to any problem that requires mining of attributed flow graphs, and is able to find sub-path and sub-graph patterns. AFGMiner was improved from its original version to use location registration. In addition, a parallel version of AFGMiner with location registration was developed that uses a workload distribution heuristic to better balance the mining work performed by different threads.

The tool HEPMiner, used for profile-based program analysis, was created as an useful application of AFGMiner. Heavyweight patterns discovered by this tool were positively evaluated by compiler engineers from the IBM Canada Software Laboratory in the context of analyzing flat-profile applications. The tool gave them useful insights on the behavior of such applications, leading to the creation of new performance counters and improvements to instruction scheduling in IBM's Testarossa JIT Compiler.

Acknowledgments

This research is partially funded by a grant from the Natural Science and Engineering Research Council of Canada through a Collaborative Research and Development grant and by the IBM Centre for Advanced Studies

References

[1] A. Jocksch, J. N. Amaral, and M. Mitran, "Mining for Paths in Flow graphs," in *10th Industrial Conference on Data Mining*, Berlin, Germany, July 2010, pp. 277–291.

[2] IBM Corporation, "WebSphere Application Server," http://www-01.ibm.com/software/websphere/, March 2009.

[3] C. S. Gomes, "Heavyweight pattern mining in attributed flow graphs," Master's thesis, University of Alberta, Edmonton, AB, Canada, 2012.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, October 2004. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2004.75

[5] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001, pp. 188–199.

[6] X. Yan and J. Han, "gSpan: graph-based substructure pattern mining," in *International Conference on Data Mining (ICDM)*, Maebashi City, Japan, December 2002, pp. 721–724.

[7] T. Horváth and J. Ramon, "Efficient Frequent Connected Subgraph Mining in Graphs of Bounded Treewidth," in *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ser. ECML PKDD '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 520–535. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87479-9_52

[8] M. Thorup, "All structured programs have small tree width and good register allocation," *Inf. Comput.*, vol. 142, no. 2, pp. 159–181, May 1998. [Online]. Available: http://dx.doi.org/10.1006/inco.1997.2697

[9] "IBM zEnterprise System Technical Introduction," http://www.redbooks.ibm.com/redpieces/pdfs/sg247832.pdf.

[10] A. Jocksch, M. Mitran, J. Siu, N. Grcevski, and J. N. Amaral, "Mining Opportunities for Code Improvement in a Just-in-Time Compiler," in *Compiler Construction (CC)*, Paphos, Cyprus, March 2010.

[11] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Knowledge Discovery and Data Mining (KDD)*, Seattle, WA, USA, 2004, pp. 647–652.

[12] T. Horváth, J. Ramon, and S. Wrobel, "Frequent subgraph mining in outerplanar graphs," in *Knowledge Discovery and Data Mining (KDD)*, Philadelphia, PA, USA, 2006, pp. 197–206.

[13] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-Based Procedural Abstraction," in *Code Generation and Optimization (CGO)*, San Jose, CA, USA, March 2007, pp. 259–270.