*

Heavyweight Pattern Mining in Attributed Flow Graphs

Carolina Simões Gomes
EMC Corporation of Canada
6020 104 Street NW
Edmonton, AB, Canada

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, AB, Canada

Joerg Sander
Department of Computing Science
University of Alberta
Edmonton, AB, Canada

Joran Siu
IBM Toronto Software Laboratory
8200 Warden Ave.
Markham, ON, Canada

Li Ding
IBM Toronto Software Laboratory
8200 Warden Ave.
Markham, ON, Canada

ABSTRACT

This paper is concerned with the discovery of patterns in *attributed* 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. This paper introduces a new algorithm, called AFGMiner, that finds heavyweight patterns in a dataset of attributed flow graphs and associates each pattern with its occurrences. The paper also describes two new tools, HEPMiner and SCPMiner. They use AFGMiner and can be utilized by compiler and application developers, respectively, to solve Profile-based Program Analysis modeled as a heavyweight pattern mining problem.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Management—*Database Applications,Data Mining*; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods, and Search, Graph and tree search strategies*

General Terms

Data Mining

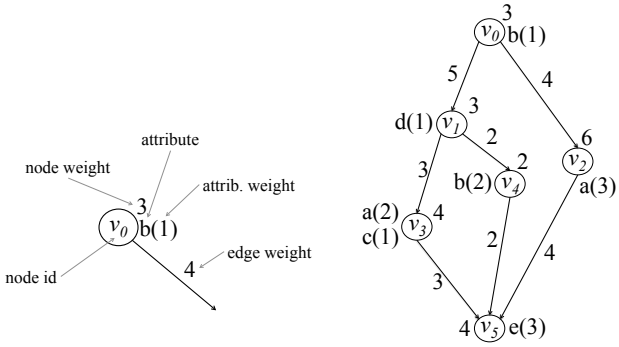
Keywords

ACM proceedings, LaTeX, text tagging

*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

1. 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. Weights associated with the edges represent either the size of the load that is transmitted from the source to the destination node, or the frequency of transmission of a token between these nodes. The nodes of a flow graph may have associated weights that can represent, for instance, the capacity of the node, or the cost of storing payload at the node, or the delay of processing a token at that node. If additional data, in the form of attributes and their values, are associated with the nodes, we have attributed flow graphs. 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. An example of an AFG is shown in Figure 1.



(a) Elements of an AFG (b) Example

Figure 1: Example of attributed flow graph.

The presence of weights associated with edges, nodes and attributes allows AFGs — either a single AFG or a set of AFGs — to create a rich description of dynamic applications. It might be best to illustrate an AFG thtrough an example that models logistics. Figure 2 shows a snapshot of the transportation by air of goods between cities. In this

illustrative example, an AFG models the volume of sea food, medicine, and flowers processed through the cargo terminals of several airports. Airports are represented as nodes, the type of goods processed at an airport as attributes of each node, the volume of each type of good handled in an airport as attribute weights, the storage capacity at each airport as node weights, and the volume capacity of the flights transporting goods between cities are represented as weighted directed edges. With similar data for multiple transportation companies, an analyst may use a set of AFGs to model a transportation network with each AFG, for instance, corresponding to a single company.

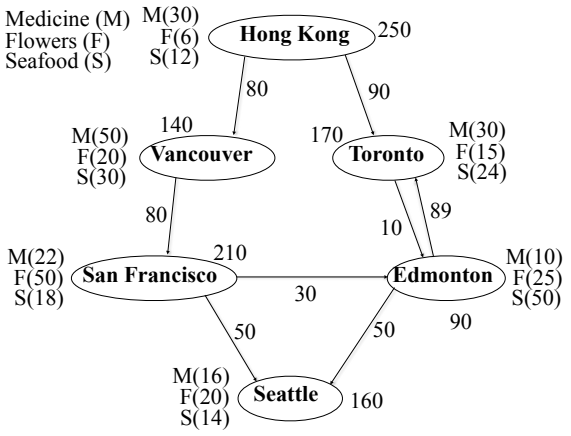


Figure 2: AFG representing transportation of goods between airports.

In many applications modelled by AFGs, complex structural subgraph patterns may be prevalent. These subgraphs are often unknown and non-trivial to detect, but they may represent important characteristics of the application. Given the richness of the model, the notion of "prevalence" or "support" of such patterns, as a measure of interestingness or importance to analysts, is often difficult to capture by simply counting the number of occurrences of the pattern in the AFG. Instead, when modelling an application with AFGs the interesting and useful patterns that an analyst wants to detect are also characterized by node attributes and a notion of support that may include node weights, attribute weights and edge weights.

Different applications may benefit from using different, application-specific, or even analysis-specific, definitions of a support metric that takes more than the frequency of pattern occurrences into account. Section 5 discusses two applications of AFGs in compiler optimization and software profiling. Although the remainder of this paper will not discuss logistics-based problems, the example in 2 seeks to underscore that AFGs and methods to discover supported patterns in AFGs are broadly applicable. However, such a method that takes the full structural information present in AFGs into account while searching for patterns does not exist.

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. A number of algorithms proposed in the literature perform sub-graph mining only on undirected and unweighted graphs, with no attributes [21] [18] [10] [7] [12]. Such algorithms mine datasets of graphs for *frequent sub-graphs*, which are patterns that occur more than a certain number of times in the dataset. An extension to the classic frequent sub-graph mining algorithm gSpan [21] calculates the support value of patterns in edge-weighted graphs with undirected edges and with no node attributes [13]. For all the aforementioned algorithms, the only way to support multiple attributes per node would be to convert each node with n attributes into n nodes with one attribute each (the attribute thus being a node label), and adding an edge of weight 0 between every pair of them. The problem with this approach is that it makes the algorithms significantly more time-consuming due to the increase in number of nodes. This graph conversion alsolimits the application of search-space pruning techniques.

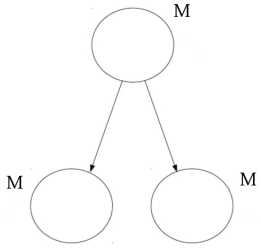


Figure 3: Pattern in the goods distribution example.

The only work on mining patterns in AFGs is the FlowGSP algorithm proposed by *Jocksch et al.* [15]. However, FlowGSP can only find sub-path patterns, while the algorithm described in this work finds all the patterns that FlowGSP does and 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 F that may be formulated by taking 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 $F(P)$ 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. HEPMiner allows compiler developers to mine for heavyweight patterns that represent particular *execution patterns*. Discovered patterns indicate potential, non-obvious, opportunities for compiler and architecture-design improvements.

4. Development of SCPMiner, a tool that automates the analysis of profiled application source-code. SCPMiner allows application developers to mine for *source-code patterns* and thus improves the performance of large software systems through changes at the source-code level.

5. Complexity and scalability analysis of AFGMiner, comparison against the FlowGSP algorithm and qualitative analysis of patterns found by HEPMiner and SCPMiner tools when applied to benchmarks.

2. 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 $g =< V_g, E_g, A_g, \alpha_g, w_g^E, w_g^V, w_g^{A,V}, l_g >$ 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 edge 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 $k - match$ of a pattern P is an occurrence g_P of P that was found by using a gap size of k_{max}. As an example, in Figure 4 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.

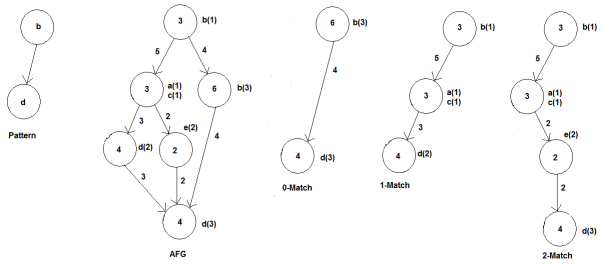


Figure 4: Example of k-matches of a pattern.

Given a dataset DS of AFGs, a *support measure* F that defines the support of patterns P in DS, and a threshold value T, the problem we address in this paper is to find all patterns P for which $F(P) > T$. We call the patterns for which $F(P) > T$ *heavyweight patterns*, and we call the problem itself **Heavyweight Pattern Mining in Attributed Flow Graphs** (for short, *heavyweight pattern mining*).

3. THE AFGMINER ALGORITHM

AFGMiner generates and tests candidate patterns of 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. Each ocurrence 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*.

3.1 Canonical Labeling

It is possible that two different patterns, when extended, generate child patterns that are isomorphic. It is inefficient to mine for redundant patterns, so redundancy should be detected. The well-known concept of *canonical labeling* of patterns is used to detect redundancy [21]. The idea is to map each sub-graph pattern to an identifier string called *DFS Code*. The string representation, or DFS Code of an AFG of n edges is $(e_0)(e_1)...(e_n)$, where each edge $e_i = (v_i, v_j)$ is represented by a string: $([l(v_i) : (a_{i_0})\ (a_{i_1})...(a_{i_m})]$ $[l(v_j) : (a_{j_0})(a_{j_1})...(a_{j_p})])$, where m and p are the number of attributes present in the edge's *from-node* and *to-node*, respectively. DFS Codes can 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.

In order for a newly-generated sub-graph pattern to be considered redundant, and thus discarded, its DFS Code should be already present in H, a hash-set whose elements are the distinct DFS Codes of all patterns that have been processed. However, this approach may consume large amounts of memory because the number of candidate patterns generated may be very high. Instead, the approach adopted in the prototype implementation of AFGMiner is to record the DFS Codes only for patterns considered heavyweight.

When producing the DFS Code of a sub-graph pattern, it is a requirement to label the edges and nodes of the pattern. Nodes are labelled as follows. The attribute set of the node is used as its preliminary label, by forming a bit vector in which each attribute is represented by a bit presence or absence of the attribute. Edge labels are simply the labels of the edges's *from-node* and *to-node* in that order. The edge labels are sorted by considering that, given any two edge labels, the one with the lower *from-node* label comes first, and in case those are the same, the one with the lower *to-node* label comes first. The minimum DFS spanning tree of the sub-graph pattern is then computed using the sorting order of the edge labels *in lieu* of edge weights. Finally, the order in which pattern nodes are visited during a pre-order traversal of this minimum spanning tree determines their final and unique-per-pattern label, which is used to sort the edges. The resulting sorted edge labels of two patterns is the same if the patterns are isomorphic.

3.2 Support Value Policy

The process of generating and testing candidate patterns in AFGMiner may be understood as the exploration of the space of patterns that exist in the dataset of AFGs being mined. The algorithm searches for all the heavyweight patterns present in this space, which can be envisioned as a forest of trees with the root node of each tree being a 0-edge pattern, child nodes of the root node being the 1-edge patterns extended from the 0-edge patterns, etc. Pruning this search space as much as possible is desirable in order to decrease the run-time of the algorithm. AFGMiner adopts an *anti-monotonic* support value policy to enable the pruning of the 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. Therefore, if a pattern is not heavyweight, none of its descendants can be heavyweight. Thus all patterns that do not meet a minimum support criteria can be discarded. The anti-monotonic support value policy allows the algorithm to prune nodes in the search space that it identifies as non-heavyweight patterns.

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 principle behind separately computing the support values of node attributes and edges of a pattern and then choosing the maximum between the two values is that a pattern is relevant if either, or both, the attributes or edges of its occurrences have significant weight in comparison to the sum of all node weights and to the sum of all edge weights in the dataset. The support-value policy selected for AFGMiner is conservative because only the minimum edge weight and the minimum node attribute weight of each occurrence are used in the computation. Less conservative support-value policies could be adopted as long as they are anti-monotonic.

3.3 Matching Patterns to Occurrences

When a candidate pattern is generated, its DFS Code is computed and the algorithm checks if the DFS Code already exists in H. If the DFS Code is not present, the candidate pattern is not redundant and the process of mining for it in the dataset begins. The mining process finds sub-graphs g in AFGs of DS that match the candidate pattern p. AFGMiner may use any sub-graph isomorphism detection algorithm to match patterns and sub-graphs, as long as the algorithm is adapted to take node attributes into consideration when detecting isomorphism. The prototype implementation of AFGMiner adapts VF2 [19], 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 [6]. For the case studies described in this paper, AFGs meet this criteria.

VF2 produces a mapping M between nodes in p and nodes in G, where G contains at least as many nodes as p. M is expressed as a set of pairs $\{n, m\}$, where $n \in p$ and $m \in G$

are nodes. The mapping is said to be an isomorphism if M is a bijective function that preserves the branch structure of both p and G. This mapping is represented as a state s in a State Space Representation (SSR). This state, which we call a *miner state*, goes from empty (no mapping between nodes in p and nodes in G) to a goal state (all nodes in p mapped to respective nodes in G) during the process of finding M. A state transition between any miner state s_1 and its successor s_2 represents the addition of a new pair $\{n, m\}$ to the collections of known pairs that compose M. The addition of new pairs only occurs if n and m form a feasible pair. The rules for a pair of nodes to be considered feasible, in the case of AFGs, are as follows: (i) attributes of m should be a proper superset of attributes in n; (ii) topological structure of n and m should be the same.

3.4 Generation of Candidate Patterns

The input to AFGMiner is a list A_0 of attributes that occur in the dataset. Each one of the possible attributes is used to create a set of 0-edge candidate patterns with a single attribute in their nodes, as shown in Figure 5. The support value threshold in this example is $T = 1/23$. Each one of the candidate patterns p is mined for in the dataset DS, and, if p is heavyweight, its only attribute is added to A_1. A_1 is the list of distinct attributes that belong to 0-edge heavyweight patterns and that are used in the generation of 1-edge candidate patterns.

After all 0-edge heavyweight patterns of a single attribute are found, the attributes in A_1 are combined in pairs and AFGMiner searches for the generated 0-edge candidate patterns that have two attributes each (Figure 6). The attributes that compose the 0-edge candidate patterns of two attributes found to be heavyweight are then combined into sets of three attributes, which form the 0-edge candidate patterns of three attributes each. This continues until no more 0-edge heavyweight patterns with a certain number of attributes can be found.

The process of mining 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 $k > 0$) is called *attribute-set growth*. Attribute-set growth is useful because an attribute set is only used to generate a pattern if all of its sub-sets generated heavyweight patterns.

Each one of the 0-edge heavyweight patterns spans multiple child candidates, one for every attribute in A_1, in which the extension node contains a single attribute from A_1. Thus, all the pairs of nodes connected by an edge in Figure 7 are child candidates for the example. AFGMiner searches for matches for each candidate child pattern in the dataset as soon as the pattern is generated. If a candidate pattern is found to be heavyweight, all its attributes are added to A_2. Attributes in A_2 are then used to generate 1-edge candidate patterns that have two attributes in their extension node. In Figure 8 a candidate is generated for each pair of nodes connected by an edge. For clarity, only candidates generated by the node with single attribute a are shown. The attribute-set growth process then proceeds until no more 1-edge candidate patterns of certain number of attributes in their extension node can be found. In Figure 9, 2-edge patterns are shown.

A set of patterns that have the same number of edges is called a *generation*. AFGMiner produces and mines for candidate patterns generation by generation as described above. 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.

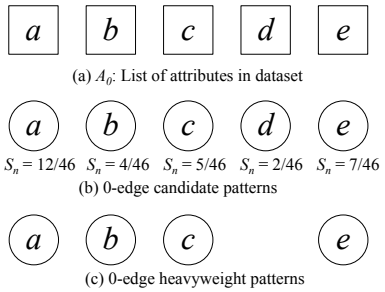


Figure 5: 0-edge candidate patterns.

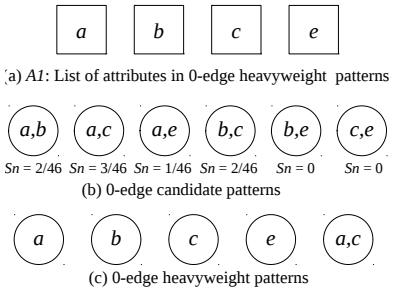


Figure 6: Attribute-set growth for 0-edge candidate patterns.

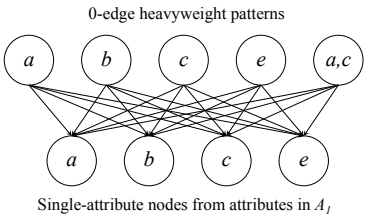


Figure 7: 1-edge candidate patterns.

4. 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 studies presented in this work. Another version of AFGMiner, **AFGMiner-locreg**, addresses this performance issue. It uses the concept of location registration.

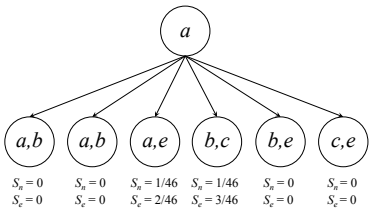


Figure 8: Attribute-set growth for 1-edge candidate pattern node a.

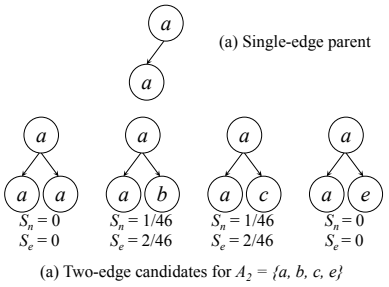


Figure 9: 2-edge candidate patterns.

Location Registration The complete mapping, including nodes and edges, between a candidate pattern p and each of its occurrences g is recorded. 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.

4.1 A Parallel Implementation of AFGMiner

A parallel version of AFGMiner benefits from the multiple cores available in many computing systems. An important challenge in the implementation of a parallel version of AFGMiner-locreg, **p-AFGMiner**, is the distribution of the workload to improve load balancing. p-AFGMiner executes the following steps while a queue Q of heavyweight patterns is not empty (s signals a sequential step, while p signals a parallel step): (i-s) fork into n threads to start the processing of a new generation composed of patterns with k edges.

For $k = 0$, (ii-s) divide the set A_0 among the threads and (iii-p) each thread generates 0-edge candidate patterns using their part of A_0 and searches the database for these patterns — heavyweight patterns form a local queue Q_{TL} and the set of attributes in Q_{TL} form A_{TL}, the local set of attributes.

For $k > 0$, (ii-s) distribute the queue Q of k-edge heavyweight patterns among n thread-local queues Q_{TL}; (iii-p) each thread generates $(k+1)$-edge candidates from patterns in Q_{TL} and computes their support, generating A_{TL} and adding the patterns found to be heavyweight to Q_{TL};

Then: (iv-s) synchronize when all threads finish step (iii) and unify all Q_{TL}s into a single queue Q, and all A_{TL}s into a single A_{k+1} set of attributes; and (v) start processing the next generation, if Q is not empty. 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.

The distribution of Q among threads should balance the workload to improve the performance of p-AFGMiner. 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.

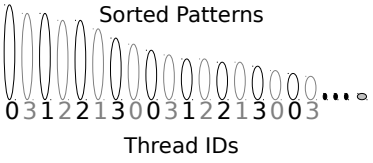


Figure 10: Illustration of pattern distribution heuristic.

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 as illustrated in Figure 10 — assume that numbering in Q starts at zero. In this illustration each ellipse represents a pattern in Q and the size of the ellipse stands for the number of occurrences of the pattern's parent in the database. 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.

5. CASE STUDIES: USING AFGMINER FOR PROGRAM ANALYSIS

The process of optimizing computer systems, compilers, computer architecture and computer applications often involves the analysis of the dynamic behaviour of an application. When this optimization is performed offline it involves the analysis of the runtime profile collected over a single, or many, executions of an application. For many applications it is possible to identify *hot-spots* in the application profile that consist of segments of the application's source code that account for a larger portion of the execution time. The effort to improve any part of the computer system that influences the application performance can then easily focus on these hot-spots. However, there is a class of computer applications that have no such hot-spots. Instead, the execution time is distributed over a very large code base, with no method taking up signicant execution time (*i.e.,* no more than 2 or 3%). Such applications are said to have *flat profiles*.

The execution of such applications also generates very large profiles that are difficult to analyze by manual inspection. Thus, the **Profile-based Program Analysis** (PBPA) problem 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. 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). The EFG is an attributed flow graph that has nodes, edges, weights and attributes whose semantics vary according to the target user of the program analysis. AFGMiner is then applied to the dataset of EFGs, and heavyweight patterns found are the ones relevant to developers.

This section discusses the run-time complexity of AFGMiner when mining EFGs. In general, the complexity of AFGMiner depends on the type of attributed flow graph being mined.

The section also presents two practical applications of AFGMiner to PBPA. The first is HEPMiner, a tool that targets compiler and architecture developers and intends to facilitate the analysis of programs that have been profiled by hardware instrumentation[17].

The second practical application of AFGMiner is SCPMiner, a tool that targets application developers and associates heavyweight patterns with source-code lines that the developer may modify in order to obtain performance improvements.

5.1 Sub-Graph Mining in Bounded Treewidth Graphs

A structured program is one that combines sub-programs to compute a function, and does so by using any combination of three fundamental control structures: (i) sequential execution of sub-programs; (ii) selective execution of certain sub-programs instead of others by evaluating boolean variables; and (iii) execution of a sub-program repeatedly until a boolean variable is true. More specifically, a structured program is free of *goto*-clauses. The classic work of *Bohm and Jacopini* shows that any program using *goto* can be converted into a *goto*-free form [3]. This conversion is done by (*e.g.,* C and C++) compilers as part of the transformation from source-code to a compiler-specific and language-agnostic Intermediate Representation (IR). *Thorup* shows that graphs representing the control flow of structured programs (*i.e.,* Control Flow Graphs) have tree-width of at most six [20]. Therefore, CFGs of structured programs have small tree-width, or, more generally, bounded tree-width.

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 [8]. *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 [8]. Because CFGs have bounded tree-width, so do the AFGs that represent CFGs, *i.e.,* EFGs. In the applications of AFGMiner shown in this paper, AFGMiner runs in incremental polynomial time because the problem being solved is fundamentally the problem of finding frequent connected sub-graphs in a dataset of bounded tree-width flow graphs. The addition of weights in nodes and edges and weighted attributes to nodes does not change the complexity of the algorithm.

The fact that nodes, edges and attributes are weighted and edges are directed does not interfere in the complexity of the algorithm, but the generation of attribute sets of increasing size when creating new candidate patterns does. 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. We can thus say that AFGMiner has incremental polynomial complexity when applied to PBPA.

5.2 A Tool for Compiler Developers and Computer Architects

HEPMiner is a program performance analysis tool that requires information about the static control flow of the program and its behavior at run-time. Dynamic information about the program is obtained by profiling its methods and includes which instructions are executed and which hardware events are triggered as a result of instruction execution. When the program is profiled, compiler log files (typically one per running thread) are created with edge profiling information, *i.e.,* basic blocks and flow edges of each method and their execution frequency measured in CPU cycles. A hardware profile is also generated and contains sequences of instructions and associated hardware events, captured by performance counters during run-time. Each instruction has a corresponding number of cycles during which it was executed, and the number of cycles in which each event was active.

The information from compiler log files and hardware pro-

file goes into database tables later read by HEPMiner, that assembles one EFG per profiled method. An EFG node is created for each assembly instruction in the profile; instructions in the same basic block are connected by new flow edges and the frequency of such edges is the same as the frequency of those basic blocks that they connect. The nodes at the end of each basic block are connected to the first nodes of all subsequent basic blocks as determined by edges in the CFG of the method, and the frequency of these edges is set to the frequency of the corresponding CFG edges. The weight of each EFG node is the number of sampling ticks associated with the corresponding assembly instruction that the node represents, and the attributes of a node are the attributes associated with the same instruction. Performance counters can mostly be directly converted into attributes.

With all EFGs assembled, HEPMiner uses the AFGMiner algorithm to find *heavyweight execution patterns* (HEP). HEPs are sets of hardware-related events captured by performance counters and associated with assembly instructions that were executing when the hardware-related events happened. Examples of such events are the occurrence of instruction and data cache misses, directory misses, address-generation interlocks. All these events that happen at program runtime result from effects that the assembly instructions being fetched, decoded and executed have on the pipeline of instructions, and from the interactions between such instructions and data that they load and store to and from caches and memory. These events affect program performance and it is thus important for compiler and architecture developers to understand when and why they happen, and which sets of events, correlated with which sets of instructions, take up more execution time. If the profile of a program is flat, manually searching for time-consuming correlations between hardware events and instructions becomes a challenge. HEPMiner automates this process, and finds non-obvious correlations represented by execution patterns that take up significant execution time when their occurrences are considered in aggregation.

5.3 AFGMiner to the Help of Application Programmers

Application developers often work on performance-sensitive applications. They measure the performance of such applications in order to detect the fragments of source-code (*e.g.,* source-code lines) that take up more execution time than others. The idea is to rewrite fragments or to move fragments in such a way that the resulting compiled code is more efficient. In contrast to compiler developers, however, application developers do not necessarily have the required knowledge to analyze low-level profiles that relate performance measurements to control flow or to hardware behavior, *i.e.,* edge, path and hardware-instrumented profiles. As a consequence, application developers typically profile their applications using tools such as *tprof* [2] to collect memory and run-time data that are associated with higher-level program constructs, *e.g.,* functions and source-code lines. Although higher-level profiling tools work well for the identification of performance bottlenecks in applications that have discernible hot methods, they do not help developers in the analyses of flat-profile applications. That is because the tools are not able to indicate to developers specific code regions that should be modified to improve performance. Randomly selecting code regions to modify is clearly not a good solution either. A better alternative is to select code regions that have similar functionality in terms of operations, and modify those code regions similar to one another that, in aggregation, take up significant execution time. SCPMiner is able to mine for such code fragments.

In order to aggregate code fragments that have similar functionality, source-code features are used. Examples of features are the number of operations (*e.g.,* addition, subtraction), the number of statements involving each data type (be it primitive or user-defined), the type of statement (*e.g.,* if-conditional, while-statement, return-statement), among others. The features are collected by parsing the target application's source-code, converting it into an abstract syntax tree (AST) and traversing the AST. When the AST is traversed, basic blocks and the appropriate flow edges are connected to generate CFGs. Simultaneously to the composition of CFGs from the AST, features are collected from the code fragments that generated each basic block. A feature vector is associated with every basic block and each feature is a weighted attribute. The result is a set of EFGs where each node is a basic block with features. Then, hierarchical cluster analysis is performed to group basic blocks with similar feature vectors. After clustering, a single attribute is associated with each of the EFG nodes. This attribute is simply the label associated with the cluster to which the EFG node belongs according to the clustering algorithm.

AFGMiner is used to mine the dataset of AFGs and to find heavyweight *source-code patterns* (SCP). SCPs are sets of cluster labels, as described above, connected by flow edges and associated with source-code lines that represent the pattern occurrences. Modifying these code patterns may lead to application performance improvements because the pattern occurrences are known to take up significant execution time when aggregated, and the fact that they have similar functionality makes it easier for developers to understand what should be changed in the code, as well as modify all occurrences at once.

6. PERFORMANCE EVALUATION METHODOLOGY

In the experimental evaluation of AFGMiner, the tools HEPMiner and SCPMiner were separately tested in order to analyze the patterns that AFGMiner is able to uncover and how scalable the algorithm is in terms of run-time. Experiments for HEPMiner were performed on an Intel Core 2 Quad CPU Q6600 machine, running at 2.4 GHz and with 3 GB of RAM. The operating system installed in the machine was a Microsoft Windows XP Professional Edition, Service Pack 3. Experiments for SCPMiner ran on an AMD Athlon II Neo K125 running at 1.70 GHz and with 4 GB of RAM, of which 3.75 GB were usable. The operating system installed in the machine was a 64-bit Windows 7 Home Premium, Service Pack 1.

6.1 HEPMiner Evaluation

The experimental evaluation of the algorithm in the context of HEPMiner used profiles from the DayTrader Benchmark, running on WebSphere Application Server, on a z196 mainframe and JIT-compiled using the IBM Testarossa JIT com-

piler. The WebSphere Application Server is a Java® Enterprise Edition (JEE) server developed by IBM and written in Java [11]. It has a very flat profile, with its execution time spread relatively evenly over 2,566 methods, as is typical of large business applications. The IBM System z is a generation of mainframe products by IBM [1]. The IBM zEnterprise System 196, or z196 model, is a CISC mainframe architecture with a superscalar, out-of-order pipeline.

AFGMiner-locreg, as used by HEPMiner, had its run-time scalability tested in two experiments, A and B, by running the algorithm on DayTrader methods. Three important parameters that were taken into account in this analysis are the *Minimum Hotness Method* (MMH) value, the minimum support threshold (MinSup) and the maximum allowed size of the attribute set in each candidate pattern node (MaxAttrs). 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. For experiments **A**and **B**, the MMH was kept at 0.001, meaning that only methods with execution time that exceeds 0.1% of program run-time are selected for mining (*i.e.* in DayTrader, 278 of 2,566).

Experiment **A** compared the run-times of AFGMiner-locreg with respect to changes in MaxAttrs. MinSup was kept at 0.001, meaning that only those patterns consuming more than 0.1% of program run-time are considered heavyweight. The goal was to measure the impact on run-time of attribute set sizes in candidate pattern nodes. The higher the number of attributes allowed, the more candidate patterns can be potentially generated, which is why modifying this parameter is a way of controlling the memory consumed by the algorithm and, we conjectured, also its run-time. *Experiment* **B** compared the run-times of AFGMiner-locreg with respect to changes in the number of EFG nodes visited by the algorithm, by changing MinSup but keeping MaxAttrs at 5.

Patterns found by AFGMiner-locreg were also compared to the ones found by the FlowGSP algorithm, which finds only sub-path patterns in AFGs. 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, an expert compiler engineer from IBM's JIT Compiler Development team verified the usefulness of patterns identified by the HEPMiner tool, as described in Section 8.

6.2 SCPMiner Evaluation

The experimental evaluation of SCPMiner focused on verifying if the patterns found by the tool are useful to developers (Section 8). The benchmarks *bzip2*, *gobmk*, *mcf* and *namd*, all from the SPEC CPU2006 [4] suite, were analyzed by an expert developer from IBM's Multi-core Performance Tooling team, using the source-code of the benchmarks and previous knowledge about them. He compared source-code lines associated with heavyweight pattern occurrences, pointed by SCPMiner, to source lines indicated by an internal IBM tool being developed by the team. This internal tool highlights the source-code lines that consume the most CPU cycles when a program is profiled by *tprof*. The IBM tool is thus only useful when the profile of analyzed applications is not flat. As a consequence, for purposes of validation, the four benchmarks we analyzed do not have flat profiles, so that the results of SCPMiner and the IBM tool could be more easily compared.

7. PERFORMANCE EVALUATION

This section describes the performance results of HEPMiner and SCPMiner.

7.1 HEPMiner Results

In our experiments using HEPMiner, AFGMiner-locreg was always faster than AFGMiner-iso, and increasing the number of threads in p-AFGMiner consistently decreased runtime. 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 subgraphs that must be tested for isomorphism with candidate patterns. It causes the number of isomorphism tests to grow linearly with the number of candidate patterns, while in AFGMiner-iso it grows exponentially for general AFGs and polynomially for AFGs with bounded tree-width (such as EFGs). This observation led to our conjecture that the number of nodes visited when mining for each candidate pattern is the major factor determining the run-time of AFGMiner, as opposed to other factors such as the MaxAttrs. We tested this conjecture in Experiments **A** and **B** described in Section 6, and the results are in Sub-section 7.2.

Memory consumption in HEPMiner is mostly determined by the size of DS. The support value has little influence, because the majority of candidate patterns are discarded, or, if considered heavyweight, deleted from memory when output to the user. The entire dataset, already in graph format, is kept in the RAM along with all mining-specific data structures. For very large datasets, another version of the tool could be developed that saves part of DS to disk and retrieves parts of it when requested to do so by the mining algorithm.

We also compared patterns found by AFGMiner and FlowGSP. Figure 11 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.

7.2 Scalability Analysis

From the results of Experiments **A** (Figure 12) and **B** (Figure 13) we conclude that 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. Figure 12 shows that, in the DayTrader Benchmark, it is more likely that heavyweight patterns have around 7 attributes per node, because making MaxAttrs higher than 7 does not produce statistically significant differences in runtime. It is worth noting that the effects of MaxAttrs on

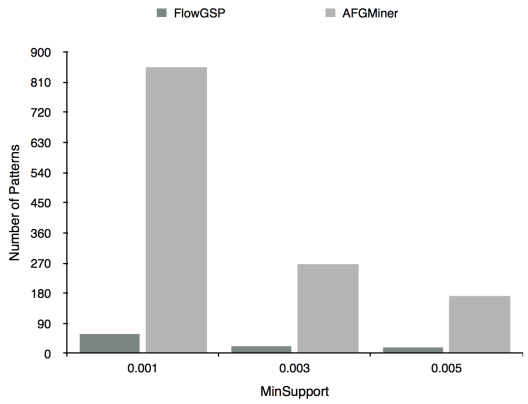


Figure 11: Patterns found by FlowGSP and AFGMiner

the performance of AFGMiner is dependent on the program being analyzed. If the program's run-time behavior causes more hardware events to happen, then the influence of MaxAttrs is more visible.

Experiment **B** changes the support value in order to increase the number of EFG nodes that the algorithm must visit. Figure 13 shows that the run-time of AFGMiner scales well enough with the number of EFG nodes visited. The performance of HEPMiner was deemed sufficient for it to be adopted as a handy tool by compiler developers in difficult analysis cases such as applications with flat profiles. However, better candidate pruning techniques could be developed to make AFGMiner faster. If more application-specific versions of the algorithm are developed in the future, those could take into account knowledge about which attributes are more likely to appear in the heavyweight patterns leading to more effective pruning of candidates.

8. QUALITATIVE ANALYSIS

This section discusses the patterns obtained by both HEPMiner and SCPMiner. Results were analyzed by expert developers from the IBM Canada Software Laboratory.

8.1 HEPMiner Results

The compiler engineer found HEPMiner to be a useful tool and was 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 the engineer's expectation that instruction cache misses should be observed only on the taken branch target.

2. From prior analysis, the expert at IBM knows 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 AFGMiner.

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 of taken and non-taken branches, and according to the IBM engineer was a very good confirmation to the compiler development team.

4. The output by AFGMiner shows patterns with low support value that have a certain attribute that is actually an instruction that is part of an asynchronous check sequence. This check sequence is used as a cooperative point in the method to allow for garbage collection and/or JIT compilation-related sampling mechanisms that determine which method is being executed.

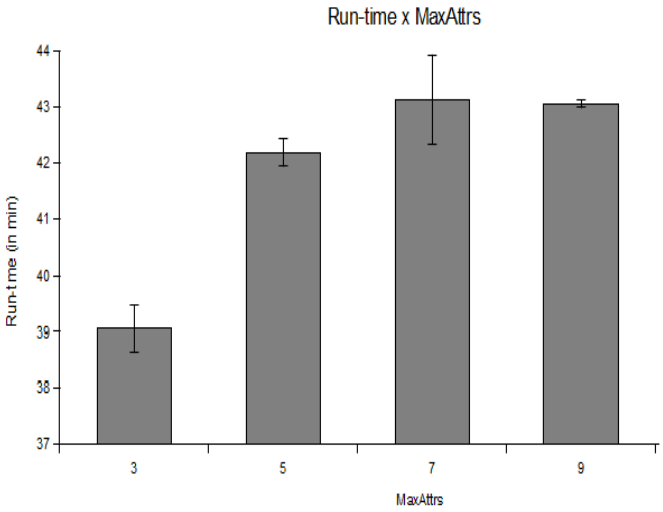


Figure 12: Experiment A.

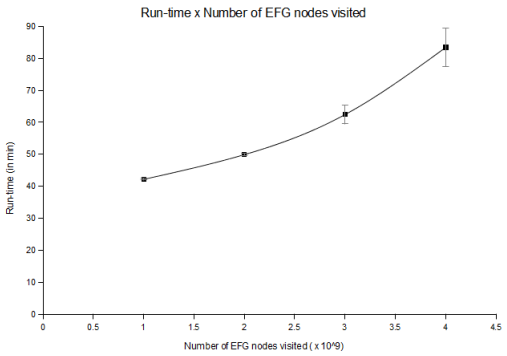


Figure 13: Experiment B.

Such checks are placed at method entries and within hot loops. The expert at IBM confirmed that, given that DayTrader is a very flat benchmark, it is correct that the pattern should have such a low support.

5. Various patterns found by AFGMiner highlight switch-statements. Switch statements are not very common in the DayTrader code, but the JIT compiler does have an opportunity to convert between branch tables and if-statements when handling conditionals, or even to use a combination of both. The appearance of such patterns with relatively low support confirms the characteristics of DayTrader.

6. The relative support values for a sequence of three attributes, present in several patterns, were found to be relevant by the IBM developer. 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.

8.2 SCPMiner Results

A software engineer from the IBM Multi-core Performance Tooling team analyzed the patterns output by SCPMiner and the occurrences linked to each pattern. The main findings are as follows.

1. The top patterns of each one of the benchmarks had occurrences whose associated source-code lines were also indicated by the IBM tool. This indicates that SCPMiner is able to find the relevant patterns in an application.

2. However, many patterns uncovered by SCPMiner did not seem relevant to the IBM developer: he did not consider the pattern occurrences to be related to one another in any meaningful way, and therefore would not consider such occurrences to form a pattern if he were to analyze the programs himself. With that we concluded that the cluster analysis implemented in SCPMiner presents mixed results, not completely aligned with the perception that application developers have of what a pattern should be. Future development of SCPMiner should consider alternative clustering analysis in order to better align the results of the tool with the expectations of application developers.

9. RELATED WORK

FlowGSP is a sequential-pattern mining algorithm that finds patterns whose occurrences are sub-paths of AFGs [16, 15]. 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 [21]. *Fast Frequent Subgraph Mining* (FFSM) is another well-known sub-graph mining algorithm, and its novelty was the introduction of embeddings that make the mining process faster. AFGMiner-locreg also uses embeddings. However in FFSM, an embedding does not take into consideration the edge relations for each occurrence of the sub-graph because mined graphs are undirected. In contrast, AFGMiner has to record the complete mapping between sub-graph patterns and occurrences, which makes the embeddings more memory-consuming. Gaston is a more recent sub-graph mining algorithm. It is based on the fact that substructures typically mined for in datasets, such as sub-paths, sub-trees, and sub-graphs, are contained in each other. This property allows the search to be split into steps of increasing complexity. A frequent path, tree and graph miner are thus integrated into a single algorithm. Gaston follows the breadth-first approach to search, and, similarly to AFGMiner, was implemented with and without embeddings. In contrast to AFGMiner, however, Gaston only mines for patterns in non-attributed, undirected and unweighted graphs [18]. *Horvart et al.* describes a sub-graph mining algorithm for outerplanar graphs, which are a strict generalization of trees [9]. Similarly to AFGMiner the algorithm runs in incremental polynomial time due to the properties of outerplanar graphs, *i.e.,* they are similar enough to trees.

The work that inspired the creation of SCPMiner is the code-clone detection tool *Deckard* presented by *Jiang et al.* [14]. The clone detection algorithm created for Deckard, similarly to SCPMiner, converts the source-code of the program to be analyzed into its AST. The algorithm characterizes sub-trees of the AST as vectors of source-code characteristics that capture structural information about the represented code. It then uses efficient hashing and near-neighbor querying for numerical vectors to cluster such sub-trees into code clones. SCPMiner, analogously, converts the AST into a set of CFGs with source-code characteristics associated with each basic block composing the CFGs, and clusters the blocks as an initial step to find source-code patterns. However, SCPMiner does not use a sophisticated clustering scheme such as Deckard's. Adapting Deckard's clustering scheme to SCPMiner would, in all likelihood, greatly improve the quality of patterns found by AFGMiner.

A work related to HEPMiner is that of *Dreweke et al.*. It applies sub-graph mining to help compiler developers in their program-improvement efforts [5]. The work presents a new approach to an existing code transformation called procedural abstraction. The approach consists of composing data flow graphs from the target program and mining, using gSpan, for frequent sub-graph patterns that represent the code segments to be extracted. HEPMiner differs from this work in that it is not a code transformation, but rather 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 both sub-path and sub-graph patterns. AFGMiner was improved from its original version to use the concepts of 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 tools HEPMiner and SCPMiner, targeted at compiler and application developers, respectively, were created as useful applications of AFGMiner. The heavyweight patterns obtained by such tools were positively evaluated by experts from the IBM Canada Software Laboratory and gave them useful insight into the run-time behavior of analyzed programs.

10. 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

11. REFERENCES

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

[2] The tprof tool. http://perfinsp.sourceforge.net/tprof.html.

[3] C. Böhm and G. Jacopini. Classics in software engineering. chapter Flow diagrams, Turing machines and languages with only two formation rules, pages 11–25. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[4] S. P. E. Corporation. Spec benchmark. http://www.spec.org.

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

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

[7] S. Han, W. K. Ng, and Y. Yu. FSP: Frequent Substructure Pattern Mining. In *Information, Communications Signal Processing, 2007 6th International Conference on*, pages 1 –5, dec. 2007.

[8] 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*, ECML PKDD '08, pages 520–535, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *International Conference on Data Mining (ICDM)*, pages 549 – 552, Melbourne, Florida, USA, November 2003.

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

[12] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 13–23, London, UK, 2000.

[13] C. Jiang, F. Coenen, and M. Zito. Frequent sub-graph mining on edge weighted graphs. In *Proceedings of the 12th international conference on Data warehousing and knowledge discovery*, DaWaK'10, pages 77–88, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

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

[16] 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.

[17] P. Nagpurkar, H. W. Cain, M. Serrano, J.-D. Choi, and R. Krintz. A Study of Instruction Cache Performance and the Potential for Instruction Prefetching in J2EE Server Applications. In *Workshop of Computer Architecture Evaluation using Commercial Workloads*, Phoenix, AZ, USA, 2007.

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

[19] 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.*, 26(10):1367–1372, October 2004.

[20] M. Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, May 1998.

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