Not having the information you need when you need it leaves you wanting. Not knowing where to look for that information leaves you powerless. In a society where information is king, none of us can afford that.

– Lois Horowitz.

University of Alberta

Data Mining Flow Graphs in a Dynamic Compiler

by

Adam Paul Jocksch

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Examining Committee

Abstract

This thesis introduces FlowGSP, a general-purpose sequence mining algorithm for flow graphs. FlowGSP ranks sequences according to the frequency with which they occur and according to their relative cost. This thesis also presents two parallel implementations of FlowGSP. The first implementation uses Java$^{\mathrm{TM}}$threads and is designed for use on workstations equipped with multi-core CPUs. The second implementation is distributed in nature and intended for use on clusters.

The thesis also presents results from an application of FlowGSP to mine program profiles in the context of the development of a dynamic optimizing compiler. Interpreting patterns within raw profiling data is extremely difficult and heavily reliant on human intuition.

FlowGSP has been tested on performance-counter profiles collected from the IBM$^{\circledR}$ WebSphere$^{\circledR}$ Application Server. This investigation identifies a number of sequences which are known to be typical of WebSphere® Application Server behavior, as well as some sequences which were previously unknown.

Acknowledgements

I would like to thank my supervisor Nelson, first and foremost, for his endless support, encouragement, and especially for his patience.

I would like to thank Osmar Zaïne for his advice in the planning stages of my research. His advice on existing data mining algorithms was invaluable.

I would also like to thank the IBM Testarossa JIT development team, specifically Marcel Mitran, Joran Siu, and Nikola Grcevski. They have provided me with an abundance of technical information and guidance throughout my research.

The research leading up to this thesis was made possible by funding from the IBM Center for Advanced Studies (CAS) and by grants from the Natural Science and Engineering Research Council.

Contents

List of Tables

List of Figures

List Of Acronyms

AGI	Address–Generation Interlock	15
AOT	Ahead-Of-Time	13
API	Abstract Programming Interface	18
BTB	Branch Transition Buffer	65
CFG	Control Flow Graph	10
CPMF	Central Processor Measurement Facility	16
EFG	Execution Flow Graph	22
FDO	Feedback–Directed Optimization	10
JDBC	Java Database Connectivity	40
JEE	Java Enterprise Edition	16
JVM	Java Virtual Machine	53
GSP	Generalized Sequential Pattern	9
ISA	Instruction Set Architecture	15
JIT	Just-In-Time	10
MPI	Message Passing Interface	18
PDF	Profile–Directed Feedback	13
SVM	Support Vector Machine	62
SPEC	Standard Performance Evaluation Corporation	62
TLB	Translation Lookaside Buffer	14

Chapter 1

Introduction

Modern optimizing compilers are powerful tools in the pursuit of improved program performance. Ultimately the ability of optimizing compilers to achieve good program performance is reliant on the quality of the optimizations performed. These optimizations are the result of many person-hours of hand tuning and investigation [6].

The development of these optimizations is a long process heavily dependant on the intuition and skill of the compiler developer. Typically the investigation of new optimization opportunities starts with the examination of current program performance. Once an opportunity has been identified, the developer can then design and test an optimization. The entire process from investigation to implementation can be extremely lengthy.

Typically, performance defects that merit the attention of a compiler designer come in two distinct forms. The first consists of events that occur infrequently but incur a large cost. The second type of defect is that which incurs a modest cost, but occurs very frequently. Identification of the first type of defect is important to program performance and it is relatively simple to address. The second type of performance issue is also important although much more difficult to discover.

Hardware performance counters are a common method for evaluating the performance of applications. Low-level information such as instruction cache misses, Translation Lookaside Buffer (TLB) misses, and data cache misses can illuminate many aspects of program behavior that may be potential targets for new optimizations. However, raw hardware profiles can be extremely verbose making the identification of performance defects difficult. Many optimization opportunities may go unnoticed simply due to the volume of information that must be examined. If performance defects in hardware profiles could be identified more rapidly, then new optimizations could be developed at a much faster pace.

Large enterprise applications, in particular, are an excellent example of how difficult it may be to extract meaningful performance information from hardware performance profiles. For example, the IBM® WebSphere® Application Server is a Java Enterprise Edition (JEE) enterprise server written in Java™. A typical Application Server profile consists of thousands of individual methods, no one of which comes remotely close to dominating the total execution time. Such profiles are often referred to as "flat" profiles as the histogram of execution time per region of code shows few discernible peaks. Performance improvements are not likely to be achieved by considering individual methods; the entirety of the Application Server must be taken into account when designing code transformations: a daunting task.

The field of data mining is devoted to the identification of patterns in data sets. An automated analysis of the hardware profile data may be able to identify performance issues. In order to accomplish this, two goals must be accomplished:

1. A data structure must be designed to represent the data contained in the hardware profile. This data structure must allow both the frequency and cost of defects to be identified.

2. An algorithm must be developed to search the data structure for patterns that correspond to performance defects in the profiling data. This algorithm needs to rank patterns in accordance with both their frequency and cost.

Generally, the hardware profiles of enterprise applications such as the WebSphere® Application Server are very flat. A profile is referred to as flat if no single method accounts for a significant proportion of the total execution time. In other words, the histogram of execution time spent on each region of code shows no discernible peaks. This characteristic of enterprise application profiles makes identification of worthwhile performance defects even more problematic.

The thesis presented in this document is:

> A suitable data mining algorithm should be able able to discover patterns in a flat profile. Some of these patterns should enable compiler designers to create new code transformations to improve the runtime performance of applications.

This thesis presents FlowGSP, a modification of the Generalized Sequential Pattern (GSP) algorithm [32], and uses it to perform data mining on an Execution Flow Graph (EFG) constructed from performance-counter data and Control Flow Graph (CFG) information extracted from the compiler.

The goal of this system is to aid in the identification of patterns in WebSphere-Application profiles which may be indicative of new opportunities for code transformation.

The main contributions of this thesis are:

- The development of an EFG to model the information contained in hardware-profile data. EFGs contain information that allows for the identification of patterns based on their frequency and on their relative weight.

- The development of FlowGSP, a data-mining algorithm designed to mine EFGs for patterns corresponding to frequent and/or costly performance defects in hardware profiles.

- The development of two parallel implementations of FlowGSP in Java: one based on Java threads for use on multi-core workstations and one based on sockets for use on distributed systems. These implementations reduce the total execution time of FlowGSP on WebSphere Application Server profiles to the point where it is practical to use FlowGSP in a compiler development environment.

- A report of the use of FlowGSP to identify performance defects in hardware profiles of the WebSphere Application Server running on the IBM System z10$^{\text{TM}}$architecture.

Chapter 2 provides background information on the technologies on which FlowGSP is based. Chapter 3 details the FlowGSP algorithm and Chapter 4 outlines its implementation. A parallel implementation and its performance is given in Chapter 5. Chapter 6 lists patterns discovered through the use of FlowGSP and the performance opportunities exploited by their discovery. Works related to this thesis are discussed in Chapter 7.

Chapter 2

Background Information

Mining interesting sequences from hardware profiles requires knowledge from multiple disciplines in computing science. Knowledge of machine learning (specifically unsupervised learning and data mining), compiler architecture, parallel algorithm construction, and low-level system architecture is required in order to properly address the mining problem. This section briefly outlines background information that may be needed to understand FlowGSP, its implementation, and its application to mining hardware profiles.

Section 2.1 provides a high-level overview of machine learning techniques, with emphasis on data mining. Section 2.2 discusses the difference between edge profiling and path profiling. Hardware performance counters are discussed in Section 2.3, and Section 2.4 discusses specifics of the IBM System z10 architecture on which the profiles were collected. The WebSphere Application Server, which is the application studied in this work, is discussed in Section 2.5. Section 2.6 discusses measuring the performance of parallel algorithms.

2.1 Machine Learning

Machine learning, or statistical learning, can be defined as the use of statistical techniques to automatically extract meaningful information from data. Machine learning is divided into two categories: supervised learning and unsupervised learning. The goals and methodology of these two types of machine learning can greatly differ and as such it is important to differentiate between them.

Unsupervised machine learning techniques attempt to automatically assign labels to an unlabeled data set, for example by finding patterns or identifying clusters of similar items [35]. Unsupervised learning algorithms are not search algorithms. Search problems look for an entry or entries in a data set that have certain characteristics. Unsupervised learning is focused on the discovery of previously unknown relationships between data points.

Supervised machine learning attempts to predict labels on unlabeled data. Construction of the predictor requires a labelled “training set” representative of the input data. A model is then constructed from the training data, and this model is used to predict the appropriate label for novel, unlabeled data [35].

2.1.1 Data Mining

When unsupervised learning is applied to large databases of information, the process commonly is referred to as *data mining*.

Typically, the type of database on which data mining is performed has the following characteristics. The database consists of a series of records, each having a number of attributes. Attributes are not constrained to any specific type; there can be boolean, numeric, or string-valued attributes. It is not required that each record have values for all possible attributes.

A subset of data mining relevant to this work is *frequent itemset mining*. An itemset, denoted $(\alpha_1, \alpha_2, ..., \alpha_n)$, consists of a set of n items $\alpha_i \in \alpha, \alpha_i \neq \alpha_j, i \neq j$, where α is the set of all possible items and $|\alpha| \geq n$. We say that an itemset matches a record in the database if every item in the itemset is also in the record. In frequent-itemset mining the level of interest or *support* of an itemset is proportional to the frequency with which it appears in the database. The goal of frequent-itemset mining is to discover all itemsets whose support in the database is greater than a minimum threshold.

An extension of frequent itemset mining is association-rule mining. Association-rule mining attempts to discover rules of the form $\beta \rightarrow \gamma$ where $\beta = (\beta_1, ..., \beta_j)$, $\beta_i \in \alpha, 1 \leq i \leq j$ and $\gamma = (\gamma_1, ..., \gamma_k)$, $\gamma_i \in \alpha, 1 \leq i \leq k$. These rules are treated as implications in the database. That is to say if β are present for a record in the database, then the items in γ are also present with a certain probability. The *confidence* of an association rule is the percentage of time that the consequent is also present out of all the times that the antecedent is present. This can be denoted mathematically as: $freq(\beta\gamma)/freq(\beta)$. The goal of association-rule mining is to discover all association rules with support greater than a given threshold and confidence greater than a given threshold.

For both frequent-itemset mining and association rule mining, the size of a rule or itemset is defined to be the number of items contained within it.

The Apriori algorithm developed by Srikant *et al.* [30] is a frequent association rule mining algorithm. The Apriori algorithm is so named because it is based on what the authors call the apriori principle: in order for a pattern to be frequent, all of its sub-patterns must be frequent as well [30].[1] Apriori searches for all rules up to a given size with a minimum support and confidence.

Apriori is an iterative generate-and-test algorithm. The first iteration is given an initial set of candidate itemsets consisting of all possible sequences of size 1. The database is scanned, and the support and confidence of each itemset is calculated. Any candidates that do not meet the minimum support and confidence thresholds are removed. The candidate itemsets for the next iteration are generated by joining the surviving items of the previous generation of candidates. This process continues until either a set number of iterations has been completed or no candidates survive the pruning process.

2.1.2 Frequent-Sequence Mining

An extension of frequent itemset mining is *frequent sequence data mining*, or frequent sequence mining. Sequences in frequent sequence mining, denoted $\langle I_1, I_2, ..., I_n \rangle$, consist of a series of itemsets I as previously defined. Databases mined for frequent sequences usually contain a number of data sequences, each of which is a totally ordered sequence of records. Any itemset I discoverable through frequent itemset mining corresponds to the sequence $\langle I \rangle$ which can be discovered through frequent-sequence mining. Thus frequent-itemset mining is a subset of frequent-sequence mining.

A sequence $\langle I_1, I_2, ..., , I_n \rangle$ matches a series of records $R_1, R_2, ..., R_n$ if and only if:

- $\forall \alpha \in I_i, \alpha \in R_i, 1 \leq i \leq n$.
- For each $R_i, i < n$, R_i immediately precedes R_{i+1}.

Many algorithms allow for an additional parameter g which specifies the maximum distance between R_i, R_{i+1} in a sequence.

The support of a sequence is typically defined to be the number of data sequences in which the sequence appears. Most frequent sequence mining algorithms do not count support for multiple instances of a sequence in the same data sequences. Confidence is not included as a metric of support unless the algorithm is also mining for association rules.

2.1.3 GSP

Our algorithm is based on a frequent-sequence algorithm called the Generalized Sequential Pattern (GSP) algorithm [32]. GSP is an iterative generate-and-test algorithm.

[1]Note that this has no connection with *a priori* or *a posteriori* reasoning.

GSP does allow for flexibility in how to determine if a sequence matches a series of records by introducing two additional parameters: maximum gap size and maximum window size. The maximum gap size determines how many vertices may occur between R_i and R_{i+1}. The maximum window size determines how many records to consider in unison when matching each itemset in a sequence. Formally, a window size of w means that a itemset I will match a series of records $R_i, R_{i+1}, ..., R_{i+w}$ if $\forall \alpha \in I, \alpha \in R_i \cup R_{i+1} \cup ... \cup R_{i+w}$.

2.2 Compiler Technology

Hardware profiles are not the only source of information regarding characteristics of a given program. Compilers contain detailed representations of the program being compiled. This information is used to perform a myriad of code transformations aimed at improving program performance. Understanding both the design process for new code transformations and the internal data structures of the compiler is important if we are to search for patterns that are of use to compiler developers.

Compilers can be divided into two distinct categories: static compilers and Just-In-Time (JIT), or dynamic, compilers. Static compilers are the most common, converting source into native executable code prior to program execution. JIT compilers run in tandem with an interpreter or virtual machine, dynamically compiling sections of the interpreted program to native code during runtime.

The most relevant aspects of compiler technology for this thesis are the representation of program flow, the collection of information about the relative frequency of execution of different portions of the program through profiling, and the use of this information, in a process called Feedback–Directed Optimization (FDO), to improve the performance of a program. The remainder of this section discusses these aspects.

2.2.1 Control Flow Graphs (CFGs)

Graphs are ubiquitously used as a way to represent the flow of execution in a program. Nearly all optimizing compilers use a graph called CFG to represent the program being compiled. As the name suggests, a CFG represents possible flow of control in the program.

Formally, a CFG is a graph $G = \{V, E, F_e\}$ defined as follows:

- V is a set of vertices.
- E is a set of directed edges (v_x, v_y) where $v_x, v_y \in V$.
- F_e is a function mapping edges in E to integer values.

Vertices in V represent units of execution in the program, usually basic blocks. Edges indicate program flow between vertices. The function F_e maps each edge $e \in E$ to an integer value indicating the frequency of execution of e. A CFG is a flow graph. Therefore the sum of the frequencies F_e on the edges leading into any given vertex must be equal to the sum of the frequencies on the edges leading out of the vertex.

The scope of a CFG is the same as the compilation unit of the compiler that created it. static compilers typically analyze a single source file at a time. As a result, the CFGs constructed by static compilers consist of multiple single-entry single-exit regions corresponding to different procedures in the source file. JIT compilers typically analyze individual methods in isolation, constructing a single CFG for each method.

2.2.2 Edge Profiling vs. Path Profiling

It is important to distinguish edge profiling from path profiling. Edge profiling records only the number of times that each edge is the CFG was traversed during execution. Path profiling records how many times an entire *path* was executed. The difference between the two forms is best illustrated with an example.

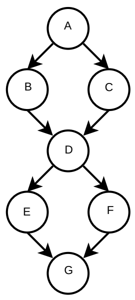


Figure 2.1: Example of a portion of a program CFG.

Consider the graph presented in Figure 2.1 to be an excerpt from a program CFG. Consider as an example that the region of code represented by the CFG in Figure 2.1 is executed five times, taking the following paths on each execution:

1. $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G$

2. $A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$

3. $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$

4. $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G$

5. $A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$

Table 2.1 shows how the above information would be encoded as a path profile. Contrast this profile with table 2.2 which shows the same information encoded as an edge profile.

Path	**Freq.**
$A \to B \to D \to F \to G$	2
$A \to C \to D \to E \to G$	2
$A \to C \to D \to F \to G$	1

Table 2.1: Example of path profiling information.

Edge	**Freq.**
$A \to B$	2
$A \to C$	3
$B \to D$	2
$C \to D$	3
$D \to E$	2
$D \to F$	3
$E \to G$	2
$F \to G$	3

Table 2.2: Example of edge profiling information.

The edge profiling information shown in Table 2.2 can be computed from the path profiling of Table 2.1. However, the path profiling information cannot be derived from the edge profiling. For instance, the path profiling shown in Table 2.3 could result from the same edge profiling information in Table 2.2.

Path	**Freq.**
$A \to B \to D \to E \to G$	2
$A \to C \to D \to F \to G$	3

Table 2.3: Possible path profiling constructed from edge profiling information.

FlowGSP mines edge profile data for sequences of attributes. Thus FlowGSP mines an upper bound for the possible execution of the sequences in the program. For example, the upper bound on path frequencies for the edge profiling of Table 2.2 is given in Table 2.4.

Path profiling is a much more precise representation of program execution. Path profiling is often referred to as *tracing*. However, for large programs, or long periods of execution, path profiling becomes increasingly impractical due to the amount of storage space required. Edge profiling, while less exact, is much more practical in these circumstances.

2.2.3 Dynamic Optimization

The heuristics that guide most standard compiler code transformations are based on *static program information*, that is to say information that can be obtained by analyzing the program source code or CFG. Unfortunately, static information is insufficient to fully predict a program's behavior at run

Path	**Freq.**
$A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$	2
$A \rightarrow B \rightarrow D \rightarrow F \rightarrow G$	2
$A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$	2
$A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$	3

Table 2.4: Upper path-profiling bounds based on edge profiling information.

time. For this reason it is difficult, if not impossible, to achieve optimal program performance using code transformations that are based on static information alone. In order to more accurately model program behavior, statistics obtained during program execution can be used in addition to static information. Such additional information is called *dynamic information*. When code transformations use this information the process is called *dynamic optimization*.[2]

In addition, dynamic optimization also allows the collection of edge-frequency information to add to the CFG of the program being compiled. This collection is a form of edge profiling.

The rest of this section outlines the most common ways that modern optimizing compilers collect runtime data.

Feedback-Directed Optimization

FDO, sometimes referred to as Profile–Directed Feedback (PDF), refers to the process where collected runtime information from the previous execution of a program is used to make optimization decisions about subsequent executions. The dynamic information required to make these optimizations is usually obtained *via* instrumentation hooks (in the case of Ahead-Of-Time (AOT) compilers) or *via* online instrumentation (in the case of JIT compilers).

For AOT compilers, the data collected by the instrumentation hooks is written to disk after execution has finished. This data is then fed back into the compiler *via* a command line parameter on subsequent compilations and used to supplement the static program information. This process is known as *Static Feedback Directed Optimization* as compilation occurs off-line, *i.e.* while the program is not executing.

JIT compilation runtime data can be utilized while the program is still being executed. Programs run under a JIT compiler can be recompiled multiple times during a single execution run with each compilation ideally resulting in improved performance over the previous version. As such, programs run under JIT compilers usually have to execute for a period of time before they reach a steady state. This process is known as *Dynamic Feedback Directed Optimization* because compilation occurs while the program is executing.

[2]Even though the compiler literature often talks about static and dynamic "optimizations", in most compiler development environments the designers implement heuristic-based code transformations. Often, the "optimal" case is not even defined.

Iterative Compilation

If the process of iterative search is applied to the optimization space of a program under an AOT compiler, then the process is known as Iterative Compilation [18]. The idea behind iterative compilation is that with each execution and recompilation performance is improved, even if slightly, over the previous version. While iterative compilation does result in very fast code, it does so at the cost of many recompilations and executions of the target program. While this large cost does make Iterative Compilation unfit for general purpose computing, the technique can still be used for specialized cases where performance is crucial and it is difficult or impossible to update the application once it has been deployed. Embedded systems and library optimization are fields in which iterative compilation can provide substantial benefits [18, 6].

JIT compilation does bear some similarities with iterative compilation. However, iterative compilation is separate from JIT compilation because iterative compilation is done offline. Domains that make frequent use of iterative compilation, such as embedded systems, are unlikely to use JIT compilers. In addition, JIT compilers include compilation time in the program runtime. Iterative compilation is therefore unattrative from the perspective of a JIT compiler.

2.3 Performance Counters

Performance counters, or hardware profiling, allow program performance at the hardware level to be recorded. Events such as instruction cache misses, pipeline stalls, and Translation Lookaside Buffer (TLB) misses to name a few are recorded by specialized hardware and then made available to the user. Typically this information is obtained by sampling the machine state periodically after a certain number of CPU cycles. This sampling period varies and can often be adjusted to suit the application being profiled.

Typically performance counter data is gathered after execution has finished, however it is also possible to gather this information while execution is occurring through the use of specialized libraries [27]. Schneider *et. al.* develop a custom run-time library to collect hardware counter information about instruction-cache misses. The work done by Schneider *et al.* in this area only involves a small number of performance counters [27]. It is unclear whether the performance overhead of such libraries would become unmanageable on architectures with a large number of performance counters. Determining this overhead, however, is outside the scope of this thesis.

Performance counters are platform-specific entities; the types of events that are recorded and the manner in which this recording is done varies from architecture to architecture.

2.4 z10 Architecture

The z10 is an in-order super-scalar CISC mainframe architecture from IBM [37]. “In-order” refers to the fact that no hardware reordering of instructions occurs during execution. The z10 is an iterative advancement over the existing z9$^{\mathrm{TM}}$architecture [29], which is in turn an evolution of the s390$^{\mathrm{TM}}$ [26].

In modern architectures it takes multiple CPU cycles to decode, prepare, and execute even a single assembly instruction. The z10 is a pipelined machine where, at any given moment, multiple instructions are at different stages of decoding or execution in order to increase instruction throughput. Each core in the z10 has its own associated pipeline. The z10 pipeline is optimized so that register-register, register-storage, and storage-storage instructions share the same pipeline latency [37].

2.4.1 Address Generation Interlock

The z10 employs an Address–Generation Interlock (AGI) pipeline. AGI pipelines are designed to avoid hazards introduced by load-use dependencies in the instruction pipeline [12]. However, this form of pipeline design introduces another type of hazard. AGI stalls occur when a memory address that is required by an instruction has not yet been computed when the instruction reaches a certain stage in the pipeline. The missing address causes execution to stall until the address generation completes. The hardware cannot let other instructions proceed ahead of the blocked instruction because the z10 is an in-order machine.

There are two common ways to avoid AGI stalls. The first is to ensure that an address calculation and corresponding use are spread far enough apart. The second is to select instructions from the System z10 Instruction Set Architecture (ISA) that are designed to minimize AGI penalties [37]. In either case, the onus falls on the compiler to produce code that avoids AGI stalls. Failure to do so can result in a significant decrease in program performance.

2.4.2 Page Sizes

Most computer architectures transfer data between long-term storage, such as hard disks and main memory, in fixed-length contiguous blocks called pages. A page normally contains 4 KB of data.

IBM System z10 allows for large pages that contain 1 MB of data [36]. Large pages are turned off or on *via* a command-line parameter given to the compiler.

2.4.3 Performance Counters on z10

The z10 Central Processor Measurement Facility (CPMF) includes numerous hardware performance counters. There are two types of counters: sampled counters and event-based counters [17]. Sampled counters are, as the name suggests, periodically sampled. Every set number of clock cycles the architecture is queried, and it's state recorded. If events of interest are in the process of occurring, then the corresponding counters are incremented. Event-based counters are automatically incremented each time the event occurs. Sampled counters record a measure of how much time was spent handling various events, and event-based counters measure how often these events occurred.

Instructions on the z-series are typically grouped in pairs. This often results in performance-counter information for one of the instructions being associated with the other and vice versa. While this clustering of instructions does introduce imprecision into the data, this imprecision is extremely localized and when taken over a large enough profile should not significantly affect overall trends in the data.

2.5 WebSphere Application Server

The WebSphere Application Server is a full-featured Java Enterprise Edition (JEE) server developed by IBM and written in Java [1]. A key characteristic of the WebSphere Application Server that makes it interesting for study is that execution time is spread relatively evenly over hundreds of methods. This even and thin distribution of execution time is a typical characteristic of enterprise applications and other middleware. In addition, there are generally very few loops that are executed during the processing of a query. Nagpurkar *et al.* stated that while each method occupies no more than 2% of total execution time instruction-cache misses make up 12% of total execution time [22]. In addition, if we want to capture 75% of all instruction-cache misses we must aggregate roughly 750 methods. Therefore, optimizing any one method is not likely to make a significant impact on overall program performance. Characteristics such as this require program behavior to be examined beyond the scope of a single method in order to accomplish efficient optimization of WebSphere Application Server. At the same time it is impractical to thoroughly optimize the entirety of WebSphere Application Server due to its size and number of methods. Thus, decisions made about how and what to optimize must yield as much global benefit as possible whilst keeping compilation overhead to a minimum.

2.5.1 Profiling WAS

WebSphere Application Server is typically run using the IBM Testarossa® JIT compiler. For this reason, and the reasons discussed in Section 2.2.3, it is important that when attempting to profile the Application Server that the proper amount of burn-in time be allowed to pass to ensure that the majority of the code being executed has been compiled to native code. For the purposes of this work, whenever profiling data is being discussed it is assumed that this data has been collected after the burn-in period and consists almost entirely of code produced by the JIT compiler. Specific details about data collection are addressed in Chapter 4.

2.6 Parallel Performance

The typical metric for program performance is raw execution time. However, this measure of performance is insufficient for programs which execute in parallel. Speed is still ultimately the goal; however the number of parallel components being executed needs to be considered as well.

The metric by which the performance of parallel programs is measured is how quickly program execution time decreases as the amount of parallelism is increased. This is referred to as *speedup*, a number representing the program's execution time relative to the sequential case. Speedup s is calculated as follows:

$$s = \frac{t_{parallel}}{t_{sequential}}$$

where $t_{sequential}$ is the execution time of the sequential version of the algorithm and $t_{parallel}$ is the execution time of the parallel version of the same algorithm. A value of $s > 1$ indicates a performance improvement over the sequential case, whereas a value of $s < 1$ indicates worse performance.

2.6.1 Linear Speedup

Ideally, if a task is split into n equal subsections we would expect the work to take $\frac{1}{n}$ of the amount of time, *i.e.* achieve a speedup of n. This is referred to as *linear speedup*, as the curve $speedup = f(n)$ is exactly equal to the line $f(x) = x$.

However, linear speedup is not always obtained in practice because some problems cannot be completely decomposed into parallel portions. Say that the running time of an algorithm takes time $t = p + q$, where p is the amount of time taken to execute the potentially parallel portion(s) of the algorithm and q is the amount of time taken to execute the portion(s) of the algorithm that cannot be parallelized. The maximum amount of possible speedup is $\frac{t}{q}$, as even if we have infinite parallel resources to make p insignificant the algorithm will still take time q to execute.

When considering sublinear, linear, or superlinear speedups, it is important to consider the factors that limit performance. In the early days of parallel programming most applications were bound by the capacity of the processor to execute instructions. Therefore, adding a second processor was expected to reduce the execution time at most by half. That is, a linear speedup was the best that one could hope for. Sub-linear speedups were explained by poor load-balancing, communication and sequencing overhead, contention for common resources, *etc*. Super-linear speedups typically indicated an error in the measurement of performance.

However, in contemporary computers, the capacity of the processor to execute instructions is no longer the limiting factor for performance. In many applications the processor is idle for most of the time. The memory hierarchy and contentions in the network are more likely to limit performance. The relationship between the number of processing nodes and performance is no longer linear. In these architectures, non-linear relations, both sub- and super-linear, between performance and the number of processing nodes should be expected.[3]

2.6.2 Memory Organization

An important factor to consider when discussing parallel systems is the type of memory organization in use. Typically organization falls into three categories: shared memory, distributed memory, and distributed-shared memory.

In shared memory, each worker has access to a single, shared pool of memory. Communication between workers is usually implicit; one worker will write to an area of memory and another worker will read the same area. Shared memory is typical of threaded systems such as `pthreads` or Java threads.

In distributed memory systems, such as clusters, each worker executes within its own address space and communication between workers must occur explicitly. This communication is usually performed through some external Abstract Programming Interface (API) such as UNIX or Windows sockets or Message Passing Interface (MPI). Distributed memory systems are commonly found in single-processor clusters.

Distributed-shared memory is a hybrid organization where groups of processors communicate *via* messages with processors in other groups. Each group of processors has access to a common

[3]An example of a super-linear speedup would be a case where a problem does not fit entirely into cache when operating on a single CPU. Adding a second CPU doubles the amount of available cache so that the problem now fits entirely inside the combined CPU caches. Alba investigates superlinear speedups in the domain of Parallel Evolutionary Algorithms; many of his conclusions are also relevant to general parallel computing[3].

area of shared memory. Distributed-shared memory is typical of clusters with multi-core nodes.

Chapter 3

FlowGSP

The theoretical basis for an algorithm is as important as its potential application. FlowGSP is based on GSP [32], which is a well-established algorithm for mining frequent sequences in a database. However, a sequence of records is a poor choice to represent program control flow. Therefore, in order to effectively mine program data a new data structure must be defined to accurately represent program behavior. Graphs are commonly used to represent the structure of a program in most compilers, and therefore it makes sense to develop a graph-based data structure on which mining can be performed. Such a data structure is introduced in Section 3.1.

Once this new data model has been defined it is then necessary to define how to measure the support of frequent sequences in this data structure. The goal of FlowGSP is to discover frequent and/or costly sequences of attributes in an execution flow graph. In the original GSP algorithm, the support of a sequence was defined as the number of data sequences in which the candidate sequence occurred [32]. Given the motivation behind the development of FlowGSP this definition is insufficient. The traditional definition ignores sequences that occur multiple times in the same data set. In the scope of the execution of a program a sequence that occurs multiple times in the same method is of interest to compiler developers. In addition to the frequency of a sequence, FlowGSP is also interested in the cost of these sequences. For these reasons the definition of support for a subpath must differ from the classical definition of support used by GSP. Section 3.2 defines support over our new data structure.

3.1 Edge and Vertex Weighted Attributed Flow Graphs

While our work was focused on mining hardware profiling data, there are many other applications where mining weighted and attributed flow graphs may be useful. This section formally describes such a data structure as well as how support values are calculated for subpaths within it.

3.1.1 Formal Definition

Let $G = \{V, E, A, F, W\}$ be an Execution Flow Graph (EFG) such that:

- V is a set of vertices.
- E is a set of edges (v_a, v_b), where $v_a, v_b \in V$.
- $A(v) \mapsto \{\alpha_1, ..., \alpha_k\}$ is a function mapping vertices $v \in V$ to a subset of attributes $\{\alpha_1, ..., \alpha_k\}, \alpha_i \in \alpha, 1 \leq i \leq k$ where α is the set of all possible attributes
- $F(e) \mapsto [0, 1]$ is a function assigning a normalized frequency to each edge $e \in E$. *i.e.* $\sum\limits_{e \in E} F(e) = 1$.
- $W(v) \mapsto [0, 1]$ is a function assigning a normalized weight to each vertex $v \in V$. *i.e.* $\sum\limits_{v \in V} W(v) = 1$.

The constraint below holds because G is a flow graph.

$$\sum_{(x,v_0) \in E} F((x, v_0)) = \sum_{(v_0,y) \in E} F((v_0, y))$$

F and W are completely independent quantities. In fact, it is this independence on which FlowGSP is based. Rather than define the importance of a sequence merely by its frequency FlowGSP also considers the weight of the sequence in question.

Example

Figure 3.1 gives an example of a vertex-weighted attributed flow graph with $\alpha = \{A, B, C, D, E\}$. The same graph with edge weights and vertex costs normalized is given in Figure 3.2. It is the graph in Figure 3.2 that will be used as input to the mining algorithm.

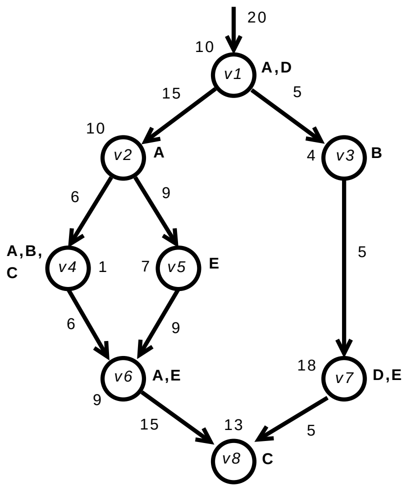


Figure 3.1: An example of a vertex-weighted, attributed flow graph. Edge weights are given along each edge and vertex weights are given next to each vertex. The letters in bold next to a vertex are attributes of that vertex.

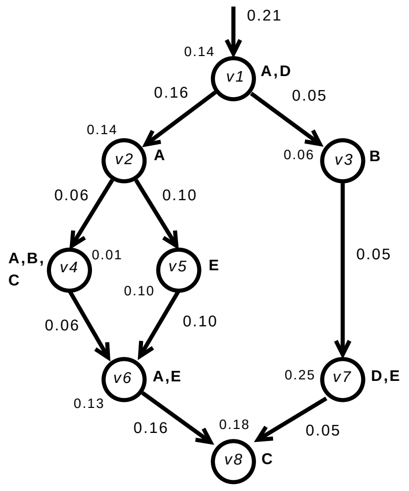


Figure 3.2: The same graph as in figure 3.1 after edge weight and vertex cost normalization

3.2 Calculating Path Support

A sequence of attributes in an EFG has a direct correspondence with a subpath in the same graph. Support metrics are defined in terms of subpaths in the graph because subpaths are more specific than sequences. This definition is then generalized to sequences of attributes.

In our program representation, sequences correspond to frequent or costly subpaths in the graph. A subpath $p_k \in G$ of length l is an ordered set of l vertices. The notation $p_k[i]$ refers to the i^{th} vertex of p_k. By definition in order for p_k to be a subpath, $(p_k[i], p_k[i+1]) \in E$ for all $0 \leq i \leq g-2$. The notation $p_k[i:j]$ refers to the subpath of p_k which consists of the i^{th} to j^{th} vertices inclusive, $i \leq j$.

The support of a path, both in terms of its cost of execution and its frequency of execution, can now be defined.

3.2.1 Frequency Support

In GSP, the support of a sequence was defined as the number of data sequences in which the sequence appeared. This definition was sufficient because the type of database being mined by GSP typically consisted of a large number of short data sequences. However, multiple occurrences of a sequence in an EFG should all contribute towards the support for a sequence. The rationale for this decision is based on the motivating application of FlowGSP. A profile is an aggregation over multiple executions of the same region of code. Therefore, to only allow a single instance of a sequence per EFG could potentially discard many other executions of the same region. This is not dissimilar to the methodologies employed in some algorithms that search for frequent subgraphs [16, 39, 14, 23].

In addition, the data sequences mined by GSP are all total orderings; they do not have the edge-weighted topological structure present in EFGs. In GSP, each occurrence of a candidate sequence is weighted equally. However it does not make sense to assign equivalent importance to two occurrences of a sequence in an EFG with different edge weights as edge weights are a measure of frequency. Therefore the edge weights must also be taken into account when determining the frequency support of a candidate sequence. Based on the two reasons discussed here, the method for calculating the frequency support for a candidate sequence must be redefined.

The definition of the frequency support of a path p_k is based on the frequencies F of the edges that comprise p_k. It can only be assumed that p_k was at most executed the same number of times as the least-frequent edge in the path.

In order to account for the degenerate case where a path consists of only a single vertex and no edges, the frequency support S_f of a single vertex v is defined as follows:

$$S_f(v) = \sum_{(v_a,v)\in E} F((v_a, v))$$

In general, the frequency support of a path p_k is:

$$S_f(p_k) = \min\{F(p_k[0], p_k[1]), \ldots, F(p_k[g-2], p_k[g-1])\}$$

3.2.2 Weight Support

In order to contrast the frequency and weight of a sequence in an EFG an additional metric of support must be defined to represent the weight of a subpath in the EFG. The weight of a subpath is based on the weights of its vertices. The weight support of a path is calculated as follows:

$$S_w(p_k) = \min_{0\leq j\leq g-1}\{W(p_k[j])\}$$

Similarly to the frequency support of a subpath, the maximum weight support of a subpath is limited by the vertex in the path with the smallest weight.

3.3 Sequences of Attributes

Attributes are the method by which information about each vertex is encoded. Each vertex may have as many attributes as is needed. The attributes assigned to each vertex will be the items in the frequent sequences mined by FlowGSP.

Attributes in our representation are binary, taking a value of either true or false. If a given attribute α_i is true for a vertex v_a then v_a has the attribute α_i. Conversely if α_i is false for v_a then v_a does not have α_i. By convention, attributes whose value is false are simply omitted. This interpretation leads to an efficient representation in which only true attributes need to be recorded for each vertex. This is especially important to reduce the storage space required to process large EFGs.

Unfortunately, not all attributes associated with a vertex are binary values. Some attributes are measures of some quantity associated with the vertex, and others may indicate which of a number of classes the vertex may belong to.

Attributes that take a numerical value (integer or otherwise) are converted to a binary representation by comparison with a set threshold. Multiple thresholds may also be used in order to separate the attribute's values into ranges.

Enumerated attributes that can take one of r possible values are represented by r mutually-exclusive binary attributes, where r is a known positive integer.

A sequence $S = \langle s_0, s_1, ..., s_{k-1} \rangle$ of length k is a sequence of k sets of attributes, denoted by $s_i, 0 \leq i < k$. For convenience, the *subsequence* $\langle s_i, ..., s_j \rangle$ of S is denoted as $S[i, j], i \leq j$. If $i = j$ this subsequence is denoted as $S[i]$.

3.3.1 Matching Sequences to Paths

Section 3.3 established how the frequency and weight support of a subpath p_k in an EFG G are calculated. All that remains in order to calculate the supports of a sequence S is to formally establish how S maps to p_k. Once this has been done, the supports are aggregated over all such p_k to determine the supports of S.

A subpath is *minimal* with respect to a candidate sequence S if both $p_k[0]$ and $p_k[g-1]$ contain part of the candidate sequence, that is to say the first and last vertices in the subpath are not skipped. Henceforth all subpaths are assumed to be minimal with respect to the candidate sequence being examined.

A subpath p_k contains a sequence S if $\forall s_i \in S$, $s_i \subseteq A(p_k[i])$. This is a very rigid definition where each set of attributes $s_i \in S$ must occur on the i^{th} vertex of p_k. This definition can be modified to allow for a more flexible matching. Indeed, this same manner of flexible matching was implemented in GSP by Srikant *et al.* [31]; this flexibility has been extended to fit the context of an EFG.

It is not required that the entirety of each set of attributes in a sequence be contained within the attributes of a single vertex in the graph. One of the parameters of the mining algorithm is the maximum windows size, w_{max}. Attributes that are observed on any vertices within w_{max} are considered to belong to the same set within a sequence. Maximum window size can be formally expressed as follows:

$$s_i \in \bigcup_{v \in p_k[l:l+w]} A(p_k[l : l + w]), 0 \leq w \leq w_{max}$$

The shorthand notation $s_i \in A(p_k[l : l + w])$ will henceforth also refer to the above constraint.

The *maximum gap size* g_{max} is the maximum allowable distance between two vertices $v_x, v_y \in$

p_k where the following holds:

- v_x is the last in a series of vertices that contains a set of attributes s_1.
- v_y is the first in a series of vertices that contains a set of attributes s_2
- s_1 and s_2 are consecutive sets of attributes in S.

It is implicit in this definition that all of the vertices in the path between v_x and v_y do not contribute any attributes to S.

Formally, we say that a subpath p_k contains a sequence S if and only if the following criteria hold:

- $\forall s_i \in S$, $s_i \in A(p_k[l : l + w])$ where $0 \leq w \leq w_{max}$
- $\forall s_i, s_j \in S$, where $s_i \in A(p_k[l : l + w])$ and $s_j \in A(p_k[m : m + w'])$
  - $p_k[l : l + w] \cap p_k[m : m + w'] = \emptyset$
  - If $j = i + 1$ then $m = l + w + 1 + g$ where $g \leq g_{max}$

In all cases, the support of a path p_k must take into account every vertex of p_k regardless of whether it contributes attributes to the sequence in question. Thus, for instance, if v has the lowest weight in p_k, $W(v)$ still determines the weight support for any sequence S contained in p_k even if v contributes no attributes to S.

3.3.2 Support of a Sequence

Ultimately, it is the support of a sequence over the entirety of the EFG being mined that is of concern. The support of S in the entire graph can be calculated by aggregating the supports over every subpath that contains S. Calculating support in this manner ensures that the highest support values are assigned to those sequences that match many subpaths in the graph and these subpaths are frequent and/or have high weight.

Given a Sequence S, its frequency and weight supports are calculated by aggregating the frequency and weight supports of every subpath p_k which contains S:

$$S_f(S) = \sum_{p_k} S_f(p_k)$$
$$S_w(S) = \sum_{p_k} S_w(p_k)$$

The mining algorithm is searching for paths in the graph that correspond to sequences of events with high frequency and/or large weight. It may also be of interest to know which sequences have disproportionate levels of frequency support compared to weight support or vice-versa. In order to concisely capture the goals of the mining algorithm two additional measures of support are introduced.

The *maximal* support of a sequence S is:

$$S_M(S) = \max\{S_f(S), S_w(S)\}$$

The *differential* support is:

$$S_D(S) = |S_f(S) - S_w(S)|$$

The rationale behind these definitions is as follows. If one or both of S_f or S_w is high, then it is likely that the sequence will be of interest either because it is frequent or because it is costly. In addition, if there is a large difference between S_f and S_w then this means that the sequence in question is either frequent but not costly, or costly but infrequent.

Example (continued)

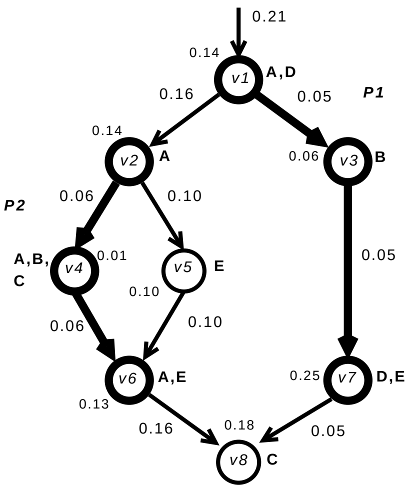


Figure 3.3: Paths $p_1 = \{v_1, v_3, v_7\}$ and $p_2 = \{v_2, v_4, v_6\}$ containing instances of the sequence $\langle(A), (B), (E)\rangle$

Consider the candidate sequence $S_1 = \langle(A), (B), (E)\rangle$. Figure 3.3 identifies two paths, $p_1 = \{v_1, v_3, v_7\}, p_2 = \{v_2, v_4, v_6\}$, containing instances of the sequence S_1 in the normalized graph from Figure 3.2. The support of sequence S_1 is calculated as follows:

$$\begin{aligned}
S_f(p_1) &= \min\{0.21, 0.05, 0.05\} = 0.05 \\
S_f(p_2) &= \min\{0.16, 0.06, 0.06\} = 0.06 \\
S_f &= 0.11 \\
S_w(p_1) &= \min\{0.14, 0.06, 0.06\} = 0.06 \\
S_w(p_2) &= \min\{0.14, 0.01, 0.14\} = 0.01 \\
S_w &= 0.07
\end{aligned}$$

Therefore, the total support for the sequence S_1 is:

$$S_M = 0.11, S_D = 0.04$$

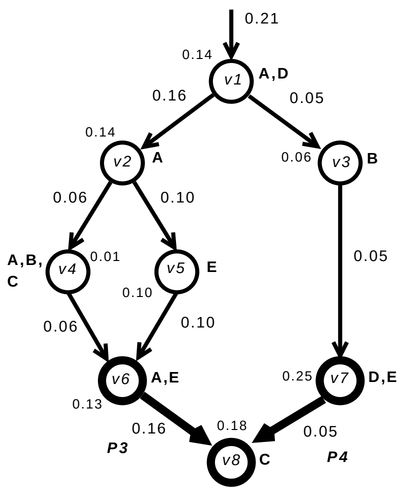


Figure 3.4: Paths $p_3 = \{v_6, v_8\}$ and $p_4 = \{v_7, v_8\}$ containing instances of the candidate sequence $\langle(E),(C)\rangle$.

Now consider the candidate sequence $S_2 = \langle(E),(C)\rangle$. Figure 3.4 gives two paths, $p_3 = \{v_6, v_8\}, p_4 = \{v_7, v_8\}$, containing instances of S_2. Note that p_3 and p_4 share a common vertex in this case. The supports for sequence S_2 are:

$$\begin{aligned}
S_f(p_1) &= \min\{0.06 + 0.10, 0.16\} = 0.16 \\
S_f(p_2) &= \min\{0.05, 0.05\} = 0.05
\end{aligned}$$

$$\begin{aligned}
S_f &= = 0.21 \\
S_w(p_1) &= \min\{0.13, 0.18\} = 0.13 \\
S_w(p_2) &= \min\{0.25, 0.18\} = 0.18 \\
S_w &= 0.31
\end{aligned}$$

Therefore, the total support for the sequence S_2 is:

$$S_M = 0.31, S_D = 0.10$$

3.4 FlowGSP

This section presents FlowGSP, an algorithm for mining sequences of attributes with either high frequency or high cost. Pseudo-code for FlowGSP is presented in algorithm 1.

The parameters to FlowGSP are an EFG G, as defined in Section 3.1, the maximum gap size g_{max}, the maximum window size w_{max}, the number of generations to iterate n_{gen}, the threshold for maximal support $S_{Mthresh}$, and the threshold for differential support $S_{Dthresh}$.

The graph G need not be the entire EFG that is being mined. The actual graph to be mined may be subdivided into independent single-entry single-exit regions and FlowGSP may be applied to each region individually. Support for each candidate is then aggregated over all regions. Currently, inter-region sequences are not considered because a JIT compiler compiles individual methods in isolation.

EFGs may contain cycles. In order to prevent traversing the graph infinitely around a cycle, a list of previously visited vertices is maintained. Children that appear on this list are not added to the queue of vertices. This restriction does not prevent the discovery of sequences that occur across loops in the graph; the list only ensures that FlowGSP starts looking for a matching path exactly once at each vertex.

FlowGSP uses a hash tree H in order to reduce the number of candidate sequences that be examined at each vertex. The creation of the hash tree H and the process of fetching candidate sequences from it is derived from the process described in Srikant *et.al.* [32]. Candidates are added to the hash tree by hashing on each attribute in the sequence, in order. The retrieval of candidates from a node in the hash tree depends on the position of the node in the tree:

- **root node:** Move to the next node in the tree by hashing on each attribute of v and any vertex

Algorithm 1: FlowGSP

```
FlowGSP($G$, $g_{max}$, $w_{max}$, $n_{gen}$, $s_{Mthresh}$, $s_{Dthresh}$)
1:  $G_1 \leftarrow Create\_First\_Generation(\alpha)$
2:  $n \leftarrow 1$
3:  while $G_n \neq \emptyset \wedge n < n_{gen}$ do
4:      $H \leftarrow Create\_Hash\_Tree(G_n)$
5:      $v_0 \leftarrow$ First vertex in $G$
6:      $Q.push(v_0)$
7:      $alreadySeen \leftarrow \emptyset$
8:      while $Q \neq \emptyset$ do
9:          $v \leftarrow Q.pop()$
10:         $alreadySeen \leftarrow alreadySeen \cup v$
11:         $C \leftarrow H.get\_candidates(v)$
12:         for $S \in C$ do
13:             $supports \leftarrow Find\_Paths(S, v, 0, true, g_{max}, w_{max})$
14:             for $(S_w, S_f) \in supports$ do
15:                 $S_w(S) \leftarrow S_w(S) + S_w$
16:                 $S_f(S) \leftarrow S_f(S) + \min\{S_f, S_f(v)\}$
17:             end for
18:         end for
19:         for $v' \in children(v)$ do
20:             if $v' \notin alreadySeen$ then
21:                 $Q.push(v')$
22:             end if
23:         end for
24:     end while
25:     for $S \in G_n$ do
26:         if $S_M(S) < s_{Mthresh} \wedge S_D(S) < s_{Dthresh}$ then
27:             $G_n \leftarrow G_n \setminus S$
28:         end if
29:     end for
30:     if $n < n_{gen} - 1$ then
31:         $G_{n+1} \leftarrow Make\_Next\_Gen(G_n)$
32:     end if
33:     $n \leftarrow n + 1$
34: end while
```

within w_{max} from v. Pass along the set of attributes that we have not yet hashed on.

- **interior node:** Move to the next node in the tree by hashing on each of the remaining attributes passed in. If none remain, add the attributes of the next vertex/vertices to the set of attributes and continue.
- **leaf node:** Return all candidates present on the leaf node.

Rather than hashing on the attributes of all data items with a time stamp in the given window, FlowGSP hashes on the attributes of the current vertex and of all its descendants that fit within the specified window size. For instance, for $w_{max} = 0$, $H.get\_candidates(v)$ in Line 11 of Algorithm 1 would return all sequences that start with an attribute associated with vertex v.

The rest of this section outlines the FlowGSP algorithm in more detail.

3.4.1 Creation of Initial Generation

$Create\_First\_Generation$ takes the set of all possible attributes and returns a set of candidates, where each candidate contains one of the possible attributes and there exists a candidate for every attribute. Formally, this can be expressed as:

$$G_1 = Create\_First\_Generation(\alpha)$$

where the following two constraints hold:

$$G_1 = \{\langle(\alpha_i)\rangle | \alpha_i \in \alpha\}$$
$$\forall \alpha_i \in \alpha, \langle(\alpha_i)\rangle \in G_1$$

For example, suppose $\alpha = \{A, B, C, D, E\}$. The result of calling $Create\_First\_Generation(\alpha)$ would be:

$$G_1 = \{\langle(A)\rangle, \langle(B)\rangle, \langle(C)\rangle, \langle(D)\rangle, \langle(E)\rangle\}$$

Both the weight support and frequency support of each sequence are initialized to zero. For instance, in this example:

$$S_f(\langle(A)\rangle) = 0$$
$$S_w(\langle(A)\rangle) = 0$$

3.4.2 Matching Path Discovery

To discover all subpaths that contain a candidate sequence S FlowGSP employs the following strategy. Each vertex v in the graph is considered as a potential starting point for a subpath p_k that contains a candidate sequence S. The search for the subpath is conducted through a depth-first search starting at v.

subpaths are found in a greedy fashion. That is, FlowGSP searches for the shortest subpath p_k that matches the given sequence S starting at the current vertex v_0. FlowGSP does not include support from a subpath p_n if there exists a subpath p_m such that: $p_m \subset p_n$, p_m and p_n both contain S, and p_m and p_n share the same initial vertex. Consider again the example presented in Section 3.3.1. The sequence $\langle (A), (D) \rangle$ is contained in two minimal subpaths: $p_m = \{v_a, v_b\}$ and $p_n = \{v_a, v_b, v_c\}$ with v_b not contributing any attributes. In this case FlowGSP would only return the subpath p_m because it is the shortest. The rationale behind this decision is that any longer subpath which also contains S has at most the same support as the shorter subpath.[1] An argument for including the supports of p_n while calculating supports for S is that p_n is capturing the event where attribute A is observed, then attribute B is observed twice in succession. However this event will be captured by the sequence $\langle (A), (B), (B) \rangle$ that will be mined in a later iteration. Therefore, given a starting vertex v, FlowGSP finds only the shortest path(s) that contains the current candidate that start at v.

Algorithms 2 and 3 outline $Find\_Paths$ and $Find\_Set$, respectively, which conduct the depth-first search for a subpath that matches a candidate sequence. $Find\_Set$ searches for the next set of attributes in the candidate sequence given the current window size w_{max}. $Find\_Paths$ then searches for the starting point of the next set of attributes in the sequence within the given maximum gap size g_{max}. Both $Find\_Paths$ and $Find\_Set$ return a set of support tuples. Each tuple in this set is formed by a weight support S_w and a frequency support S_f.

$Find\_Paths$ and $Find\_Set$ are mutually recursive. After $Find\_Set$ finds a set of attributes, it calls $Find\_Paths$ to find the rest of the sequence. $Find\_Paths$ in turn calls $Find\_Set$ to find the next set of attributes in the sequence. After the initial call to $Find\_Paths$ returns, the (S_w, S_f) values are added to the frequency and weight support of S. The recursion between $Find\_Path$ and $Find\_Set$ is guaranteed to terminate. The search for a subpath that matches a sequence S will stop when either the sequence has been found, or when the maximum gap and maximum window size has been exhausted and no matching subpath has been found. Therefore the search is guaranteed to

[1]This invariant holds because the frequency and weight supports are based on the edge with the lowest frequency or the vertex with the lowest weight respectively.

terminate in a finite amount of time because w_{max}, g_{max}, and the size of S are all constant.

$Find\_Paths$ takes a parameter g_{remain}, indicating the remaining size of gap that may occur in the current sequence. If $Find\_Set$ returns $\emptyset$ and $g_{remain} = 0$, then $Find\_Paths$ returns an empty set. Similarly, $Find\_Set$ has a parameter w_{remain} that determines how many edges the algorithm should traverse from the current vertex to find all the attributes that belong to the current set of attributes. If $w_{remain} = 0$, then $Find\_Set$ will return $\emptyset$ instead of investigating further vertices.

$Find\_Paths$ also takes a boolean parameter $firstSet$ which is set to true if and only if $Find\_Paths$ is searching for the start of the sequence. This parameter exists solely for the purpose of passing this information to $Find\_Set$.

The initial call to $Find\_Paths$ in algorithm 1 is given zero as the remaining gap regardless of the value of g_{max}, to ensure that the first set of attributes in the sequence starts on that vertex. Therefore the subpath found is minimal.[2]

$Find\_Set$ returns a S_f value of infinity if there are no more itemsets left to find in S. This value of S_f is assigned on Line 5 of Algorithm 3. Infinite support indicates that $Find\_Set$ has not traversed any edges in order to find the current set of attributes. Therefore, there is no meaningful value to return for S_f. The S_f of the entire path is calculated by taking the minimum between the new value and a previously computed value. Therefore, assigning a value of infinity ensures that this new value will not alter the previously calculated value of S_f.

$Find\_Set$ takes two parameters that together represent the set of attributes the algorithm is searching for. s_{left} contains the attributes that we have yet to find, and s_{found} contains the attributes that were previously located. On the initial call to $Find\_Set$ from $Find\_Path$, s_{left} contains the entire set of attributes and $s_{found} = \emptyset$.

$Find\_Set$ also takes two boolean parameters, $firstSet$ and $startOfFirstSet$. $firstSet$ is set to true if and only if $Find\_Set$ is looking for the first set of attributes in the sequence, and $startOfFirstSet$ is true if and only if $Find\_Set$ is searching for the start of the first set of attributes. The rationale behind these parameters is as follows.

$firstSet$ is used on line 21 to ensure that we find the shortest sequence of vertices which matches the current sequence. If the current vertex contains all of the attributes previously found on the first set of a sequence then the subpath being explored is not minimal. Therefore $firstSet$ returns the empty set. The shorter subpath will be discovered on a future call to $Find\_Paths$.

$startOfFirstSet$ is required on line 18 to ensure that we do not allow a vertex which does not

[2]No steps need to be taken to ensure that $p_k[g-1]$ (*i.e.* the last vertex on the path) contributes to the sequence because the search for a matching path terminates at this point

contribute to the first item set to occur at the start of the subpath.

Algorithm 2: Algorithm to find all paths that contain a sequence S starting at a vertex v.

```
$Find\_Paths(S, v, g_{remain}, firstSet, g_{max}, w_{max})$
 1: $supports \leftarrow Find\_Set(S[0], \emptyset, S, v, w_{max}, firstSet, firstSet, g_{max}, w_{max})$
 2: if $supports \neq \emptyset$ then
 3:     return $supports$
 4: end if
 5: if $g_{remain} \leq 0$ then
 6:     return $\emptyset$
 7: end if
 8: for $v' \in children(v)$ do
 9:     $supports' \leftarrow Find\_Paths(S, v', g_{remain} - 1, false, g_{max}, w_{max})$
10:     for $(S_w, S_f) \in supports'$ do
11:         $S_w \leftarrow \min\{S_w, W(v)\}$
12:         $S_f \leftarrow \min\{S_f, F((v, v'))\}$
13:         $supports \leftarrow supports \cup \{(S_w, S_f)\}$
14:     end for
15:     return $supports$
16: end for
```

Example

Figure 3.5 gives a small example of an EFG to illustrate the behavior of $Find\_Path$ and $Find\_Set$. For this EFG $\alpha = \{A, B\}$, $g_{max} = 0$, and $w_{max} = 0$.

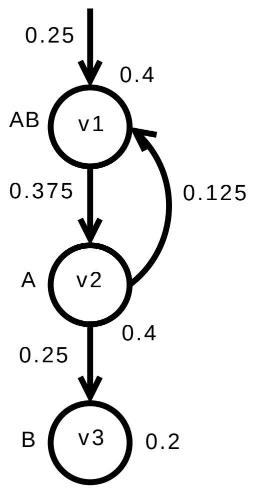


Figure 3.5: Small example of an EFG

Consider that $FlowGSP$ has reached the vertex v_2. Hashing on the attributes contained in $A(v_2)$, the candidates that could potentially start on v_2 in generation G_2 are $S_1 = \langle (A), (A) \rangle$, $S_2 = \langle (A), (B) \rangle$, and $S_3 = \langle (A, B) \rangle$. $FlowGSP$ then makes the following calls to $Find\_Paths$:

- $Find\_Paths(S_1, v_2, 0, true, 0, 0)$. $Find\_Paths$ immediately calls:

Algorithm 3: Algorithm to find the next set of attributes in the sequence.

```
$Find\_Set(s_{left}, s_{found}, S, v, w_{remain}, firstSet, startOfFirstSet, g_{max}, w_{max})$
1:  $supports \leftarrow \emptyset$
2:  $k \leftarrow |S|$
3:  if $s_{left} \subseteq A(v)$ then
4:      if $k = 1$ then
5:          $supports = \{(W(v), \infty)\}$
6:          return $supports$
7:      end if
8:      for $v' \in children(v)$ do
9:          $supports' \leftarrow Find\_Paths(S[1, k-1], v', g_{max}, false, g_{max}, w_{max})$
10:         for $(S_w, S_f) \in supports'$ do
11:             $S_w \leftarrow \min\{S_w, W(v)\}$
12:             $S_f \leftarrow \min\{S_f, F((v, v'))\}$
13:             $supports = supports \cup \{(S_w, S_f)\}$
14:         end for
15:     end for
16:     return $supports$
17: else
18:     if $startOfFirstSet \wedge A(v) \cap s_{left} = \emptyset$ then
19:         return $\emptyset$
20:     end if
21:     if $firstSet \wedge s_{found} \subseteq A(v)$ then
22:         return $\emptyset$
23:     end if
24:     if $w_{remain} \leq 0$ then
25:         return $\emptyset$
26:     end if
27:     $S_{left} \leftarrow s_{left} \setminus A(v)$
28:     $S_{found} \leftarrow s_{found} \cup (A(v) \cap s_{left})$
29:     for $v' \in children(v)$ do
30:         $supports‘ \leftarrow Find\_Set(s_{left}, s_{found}, S, v', w_{remain} - 1, firstSet, false, g_{max}, w_{max})$
31:         for $(S_w, S_f) \in supports'$ do
32:             $S_w \leftarrow \min\{S_w, W(v)\}$
33:             $S_f \leftarrow \min\{S_f, F((v, v'))\}$
34:             $supports \leftarrow supports \cup \{(S_w, S_f)\}$
35:         end for
36:     end for
37:     return $supports$
38: end if
```

- $Find\_Set(\{A\}, \emptyset, S_1, v_2, 0, true, true, 0, 0)$. $S_1[0] = A$ and therefore $S_1[0] \subseteq A(v_2)$. $Find\_Paths$ is called to search for $S_1[1]$ starting with the children of v_2.
    - $Find\_Paths(S_1[1], v_1, 0, false, 0, 0)$. $Find\_Paths$ immediately calls:
        - $Find\_Set(\{A\}, \emptyset, S_1[1], v_1, 0, false, false, 0, 0)$. $S_1[1] = A$ and therefore $S_1[1] \subseteq A(v_1)$. $Find\_Set$ returns the tuple $(0.4, \infty)$ because $|S_1[1]| = 1$ and therefore we have found the entire sequence.

      $Find\_Set$ returned a non-empty set, therefore $Find\_Paths$ returns the same set.

  $Find\_Set$ sets $S_w = \min\{0.4, 0.4\} = 0.4$ and $S_f = \min\{\infty, 0.125\} = 0.125$. $(0.4, 0.125)$ is added to the set of support tuples.

  $Find\_Set$ now calls $Find\_Paths$ to search the next child of v_2.
    - $Find\_Paths(S_1[1], v_3, 0, false, 0, 0)$. $Find\_Paths$ immediately calls:
        - $Find\_Set(\{A\}, \emptyset, S_1[1], v_3, 0, false, false, 0, 0)$. $A(v_3) = B$, and therefore $S_1[1] \not\subseteq A(v_3)$. $Find\_Set$. $Find\_Set$ returns $\emptyset$ because $w_{remain} = 0$.

      $Find\_Paths$ also returns $\emptyset$ because $Find\_Set$ returned $\emptyset$ and $g_{remain} = 0$.

  $Find\_Set$ returns the supports accumulated thus far because all of the children of v_2 have been explored.

$Find\_Paths$ returns the set of supports returned by $Find\_Set$. These supports are added to the support values of S_1.

$FlowGSP$ then repeats the above procedure, calling $Find\_Paths$ for S_2 and S_3.

In this example, the role of $Find\_Paths$ is diminished because $g_{max} = 0$. If $g_{max} > 0$ then $Find\_Paths$ would explore children of the current vertex in the event that $Find\_Set$ returns $\emptyset$. This exploration continues until the maximum gap size has been reached.

3.4.3 Candidate Generation

The creation of new candidates prior to the start of the next iteration by $Make\_Next\_Gen$ is handled exactly as in Srikant *et al.* [32]. A brief description is included here.

New candidates are created by joining compatible candidates in the current generation. Candidates are defined as compatible if the *suffix* of the first candidate is equal to the *prefix* of the second. The suffix of a sequence is created by removing the first *attribute* from the sequence. Note that this is very different from removing the first set of attributes from the sequence. For example, the suffix of the sequence $\langle(A, B), (C)\rangle$ is $\langle(B)(C)\rangle$. The prefix of a sequence is likewise formed by removing the last attribute of the sequence. The prefix of $\langle(A, B),(C)\rangle$ is $\langle(A, B)\rangle$. A prefix and suffix are defined to be equal if and only if they are identical. For instance, the suffix $\langle(B),(C)\rangle$ and the prefix $\langle(B, C)\rangle$ are not equal as they are similar, but not identical.

By this rule, the sequences $\langle(A, B),(C)\rangle$ and $\langle(B),(C),(D)\rangle$ are compatible to be joined because $\operatorname{suffix}(\langle(A, B),(C)\rangle) = \langle(B),(C)\rangle = \operatorname{prefix}(\langle(B),(C),(D)\rangle)$. The sequences $\langle(A, B),(C)\rangle$ and $\langle(A, B),(D)\rangle$ are not compatible as the suffix of the first sequence, $\langle(B),(C)\rangle$, is not equal to the prefix of the second sequence, $\langle(A, B)\rangle$.

To join candidates, the attribute removed to create the prefix of the second sequence is appended to the first sequence. Joining $\langle(A, B),(C)\rangle$ and $\langle(B),(C),(D)\rangle$ yields the sequence $\langle(A, B),(C),(D)\rangle$. Similarly, joining $\langle(A, B),(C)\rangle$ with $\langle(B),(C, D)\rangle$ yields $\langle(A, B),(C, D)\rangle$.

The lone exception to this rule is when two sequences with only one attribute are being joined because both the suffix and the prefix are empty. In this case, the removed attribute must be added both as part of the last set of attributes and as part of a new set of attributes. Joining $\langle(A)\rangle$ with $\langle(B)\rangle$, for example, yields the sequences $\langle(A, B)\rangle$ and $\langle(A),(B)\rangle$.

The rationale behind this method of candidate generation is to only generate candidates that have the potential to meet the minimum support requirements. A sequence cannot have higher support than any individual subsequence [32]. Candidate sequences are generated in this manner in order to avoid the creation of sequences that cannot have support greater than the given threshold.

Final Comments

This Chapter defined an execution flow graph: a data structure that models topologically-ordered, attributed data with weights for both frequency and cost. Within this data structure, the support of subpaths was defined both in terms of their frequency and their cost. Using this data structure, maximal and differential support were defined. This enabled us to construct a data mining algorithm that can search for frequent and/or costly sequences in an EFG.

FlowGSP was also formally defined to search for all sequences that occur with maximal or differential support greater than a given threshold. Discussion can now turn to the application of FlowGSP to the problem of mining frequent sequences from hardware profiles.

Chapter 4

Implementation

Now that FlowGSP has been established its application to WebSphere Application Server profiles can be discussed.

Section 2.2.1 discussed CFGs and how they are used to model program behavior. However, CFGs usually only encode information such as edge or block frequency. Frequency information is usually sufficient for making code transformation decisions. However, there is a wealth of information about the behavior of a program that is not captured by CFGs.

Hardware profiles are rich with low-level details about what occurred during program execution. However, profile information is typically unstructured and contains no control-flow information. Each method in the profile merely contains a list of instructions, the sampling ticks incurred on each instruction, and any associated hardware-counter information. Combining this detailed, low level, information with the high-level control flow information in the CFG creates a reasonably accurate model of program execution that can then be mined.

The general philosophy behind the implementation of FlowGSP is that the algorithm must be correct first, and then efficient. While a number of optimizations were required in order to ensure that FlowGSP was able to operate entirely in main memory, for example, there was not an extensive amount of time spent optimizing small details of the algorithm.

The platform for this research is a cluster of 16 nodes, each node being equipped with dual quad-core AMD 2350 CPUs with 8GB of RAM. The database containing the profiling data is hosted on a separate machine running a dual-core AMD CPU with 2GB of RAM. The database server is running the express edition of IBM's DB2® database server.

All profiles were collected using WebSphere Application Server 6.1, 64-bit edition running on the IBM System z10$^{\mathrm{TM}}$architecture running Linux for System z$^{\mathrm{TM}}$.

FlowGSP is implemented in Java due to its portability, robust threading capabilities (discussed in chapter 5), and mature database interaction capabilities through the Java Database Connectivity (JDBC) libraries.

Sections 4.1 and 4.2 discuss the collection and storage of information required to construct the EFG. Section 4.3 discusses the actual construction of the EFG. Section 4.3.1 discusses the specific attributes used in mining z10 profiles. Considerations specific to this experimental setup are discussed in Section 4.4. Section 4.5 discusses division of the EFG for mining purposes.

4.1 Data Collection

This experimental evaluation of FlowGSP uses a WebSphere Application Server profile collected over five minutes. This profile is collected after the Application Server reaches a steady state because it is the code which has already been natively compiled by the JIT which is of interest. WebSphere Application Server is deemed to be in a steady state when throughput shows little to no discernible change for a period of two minutes. An average hardware profile produces 450 MB of uncompressed data, and the compiler log produces roughly 6 GB of data. Although many compiler attributes could be derived from the log, the focus is on CFG information.

Five minutes was the longest profile that could be collected. The amount of hard disk space required to collect the profiling data is much larger than the 450 MB required to store the resulting profile. Also significant is the amount of hard drive space required to output the compiler log.[1] Given the amount of available disk space on the z10 machine used for these experiments, five minutes was the longest profile that could be collected. Given additional disk space, there is no reason that a longer profile could not be collected.

Many profiles were collected while developing and testing FlowGSP. Profiles were collected to test different compiler configurations and hardware profiling features.

Only slight modifications to the IBM Testarossa JIT compiler were required for our experiments. An annotation was added to each instruction in the compiler log to allow the instruction to be mapped back to its basic block.

[1]At the time of writing, the IBM Testarossa JIT compiler was not able to output logs in compressed format.

4.2 Data Storage

The profile data and CFG data to be mined is stored in a relational database. Population of hardware-profile information in the database was done using an in-house tool provided by IBM. CFG information is read from the compiler logs and added to the database containing the profiling information. A relational database allows easier access to arbitrary sections of the graph at the cost of some increased post-processing. Therefore, in order to allow the easy subdivision of work, a database is more desirable than alternative formats such as a flat text file.

The information in the database is organized into the following tables:

Symbol: The symbol table contains a record for each method (or symbol) in the profile.

Disassm: The disassm table contains the raw assembly code from each method/symbol. This table includes information such as the instruction opcode, operands, and the offset in bytes of the instruction from the beginning of the method.

IA: The Instruction Address (IA) table contains the actual profiling information collected during execution.

Listing: The listing table contains information gleaned from the included compiler logs. This information includes which bytecode and basic block associated with each instruction.

CFGNode: The CFGNode table contains information pertaining to basic blocks extracted from the compiler logs. The number, associated symbol, and frequency of each block is recorded.

CFGEdge: The CFGEdge table records all information about inter-basic block edges. The type (in, out, exception-in, or exception-out), source block, destination block, and frequency of each edge is recorded.

In order to reduce storage requirements only edges between basic-blocks are stored. Intra-basic block edges are inferred based on instruction offset. As stated earlier, many Application Server profiles were collected. The most up to date profile contains 102,865 individual assembly instructions, 30,430 basic blocks, and 44,178 inter-basic block edges.

Most database servers enforce strict levels of isolation to ensure that queries remain independent.[2] This isolation results in slower database performance but is required to ensure absolute correctness in the presence of multiple connections. However, most levels of query isolation assume that data is both being written and read to the database. Multiple concurrent queries cannot affect each other's results because FlowGSP only reads information from the database. Therefore, the level of query isolation can be reduced to the lowest possible setting in order to improve performance.

[2]The level of query isolation in a database management server refers to how each query obtains locks on portions of the database to ensure the atomicity and reproducibility of each transaction.

4.3 Construction of Execution Flow Graphs from Profiling Data

This section explains how an EFG is constructed from the information contained in the compiler log and the hardware profile.

The assembly instruction is the unit of execution used in the graph. Therefore a vertex is created for each assembly instruction in the profile. Instructions within the same basic block are connected into a path according to their offsets and the frequency of each edge is set to the frequency of the basic block. The vertices at the end of each basic block are connected to the first vertices of all subsequent basic blocks as determined by edges in the CFG. The frequency of these edges is set to the frequency of the corresponding edges in the control flow graph.

The weight of each vertex is equal to the number of sampling ticks incurred on the corresponding assembly instruction. The attributes of a vertex are the attributes associated with the assembly instruction the vertex represents. In the most recent profile the maximum number of attributes observed on any given vertex was 70, with an average of 1.4 attributes per vertex. A detailed list of possible attributes is given in Section 4.3.1.

Raw edge frequencies and vertex weights may be of different magnitudes. Thus, in order to directly compare frequency and weight, both quantities are normalized. Edge frequencies are normalized with respect to the sum of all edge frequencies in the graph, and vertex weights are normalized with respect to the sum of all vertex weights.

During execution, data for each method is fetched independently and used to build a single-entry single-exit EFG. This method-based construction allows the graph to be easily partitioned as may be required for a parallel implementation. Construction of the graph in this format is efficient because profile data in the database is indexed by unique method id.

4.3.1 Attributes

The majority of hardware counters were directly converted into attributes. This conversion was done by thresholding against the value zero because counters are integer-valued. In other words, an attribute representing a counter is present on a vertex if any ticks were recorded for the corresponding instruction.

A main goal in the development of FlowGSP is the discovery of patterns that lead to the development of new code transformations. In light of this goal, as much information as possible is included in the EFG. An abundance of information may lead to the discovery of false positives, *i.e.* patterns that are identified as being of interest but from which no new code transformations can be developed.

The occasional discovery of false positives is preferable to failing to discover a sequence of genuine interest. If some hardware counters proved to be of little use after further study they are removed. As an example, there are many counters representing catch-all situations that are not deemed to be of interest. These counters were initially included as attributes, however early experiments revealed that they provided little new information about the program. Therefore, they were dropped from the list of attributes. Other attributes were removed after they proved to be too ubiquitous to be useful; an attribute that appears on nearly every vertex is of little use when attempting to identify interesting patterns in the data because it tends to occur in nearly every sequence.

Attributes are encoded as integer values on each node and a symbol table is used to look up the actual names of the attributes if required. Integer comparison is significantly cheaper than string comparison. Moreover, storing attributes as strings would lead to both an increase in required memory as well as an increase in computation time because the EFGs being mined consist of thousands of nodes.

Not all attributes are based purely on counter information, some are calculated based on other attributes of the instruction. The prologue of a method is a stub automatically inserted by the compiler prior to the actual method code. The prologue is responsible for setting up the environment in which the method operates. For example, to manipulate the stack pointer appropriately. If the offset is below a given threshold, the instruction is determined to be part of the method prologue and assigned the `Prologue` attribute. Whether or not an instruction is in the prologue is of interest because the prologue of a method is more likely to incur cache misses if the call to the method was not predicted correctly.

Similarly the `JITtarget` attribute is assigned to an instruction if it is the entry point of the method when it is called from JITted code. This entry point differs from the entry point executed when the method is called from interpreted code. This attribute is of interest because it can help to determine performance differences when methods are called from native code versus interpreted code.

Some attributes are not integer-valued; rather they take on one of a finite number of discrete values. For instance, the `Opcode` attribute is the opcode of the current instruction (Store, Load, Branch, etc.). Such attributes are of interest because they tie the hardware events with the instruction being executed at the time. The `InlineLvl` attribute is the inlining level of the current instruction. An instruction from an inlined method will have an inlining level of 1, an instruction from a method inlined within an inlined method will have an inlining level of 2, and so on. If the instruction was not inlined from any other method then this attribute is not present. This attribute is relevant because excessive inlining may cause performance degradation.

4.3.2 Consequences of Edge Profiling

The CFG used to create the EFG only contains frequencies on individual edges. As such the CFG, and consequently the resulting EFG, is a form of edge profiling. Therefore, only *possible* sequences of attributes can be discovered.

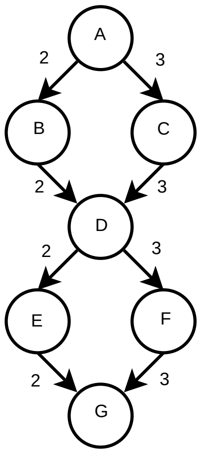


Figure 4.1: Example of CFG annotated with edge profiling information.

Consider again the example presented in Section 2.2.2, re-illustrated in Figure 4.1. According to the edge profiling information it could be inferred that the path $A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$ was executed twice because every edge on the path was executed two times. However, if the path profiling information is examined it is discovered that the path $A \rightarrow B \rightarrow D \rightarrow E \rightarrow G$ was never actually executed. Unfortunately without path-profiling information it is impossible to determine whether or not a path in the graph was executed with absolute certainty.

Therefore, within the limitations of edge profiling the best that can be expected is the identification of *possible* sequences of attributes in the EFG. The potential severity of this issue is directly related to the number of junctions crossed by the sequence in question. Therefore shorter sequences, which should statistically cross fewer junctions, will likely not be as significantly effected. By keeping the number of generations reasonable, and therefore the length of sequences, error due to edge profile inaccuracy can be controlled.

4.4 Architecture Specific Considerations

As mentioned in Section 2.4, instructions on the System z10 are grouped in pairs, introducing a small amount of noise into the profiling data. To compensate for this, most runs of FlowGSP are run with $w_{max} = 1$. The idea behind this decision is that the vertices representing instructions in the same pair are grouped into the same series of vertices. It is possible that this strategy could backfire, *i.e.* instructions not in the same grouping could end up in the same series of vertices. However, statistically speaking the odds of this occurring are the same as instructions from the same grouping being associated. Therefore the amount of additional imprecision introduced is likely to be minimal.

4.5 Graph Division

As JIT compilers make most of their optimization decision at an intra-method scope, it makes sense that the patterns most interesting to JIT compiler developers would be those that occur wholly within methods. For this reason, FlowGSP mines each method as an independent graph. As discussed in Section 5.1, this is also a natural way of decomposing the problem into sub-problems.

4.6 Sequential Performance

As will be outlined in Chapter 5, while sequential performance of FlowGSP is not excruciatingly slow, in order for FlowGSP to be a useful tool in a compiler development shop the execution time must be reduced well beyond its current time of almost six hours to mine a five-minute profile.

Given the previously discussed independent nature of the EFG produced by WebSphere Application Server profiles, a parallel implementation may be an excellent way of increasing throughput.

Final Remarks

Chapter 3 introduced EFGs and showed that they are a useful structure for data mining. This chapter described how to practically construct an EFG using hardware profiling data and information from the CFG from compiler logs. This chapter also defined a set of attributes suitable for representing the types of events found in System z10 hardware profiles.

Chapter 5

Parallel Performance

Data mining is a computationally intensive problem. It can take many hours of CPU time to mine for all frequent or heavily weighted sequences in an EFG, especially if the number of attributes is large. Given that FlowGSP is intended as an aid to compiler developers, decreasing the turnaround time between runs would greatly increase the algorithm's utility as a tool. Given the inherently parallel nature of the EFGs created from WebSphere Application Server profile data, a parallel implementation of FlowGSP may very well accomplish this goal.

When deciding to parallelize an algorithm, it is important to remember that the amount of potential parallelism is limited by the portion of the algorithm which must be executed sequentially. For example, if 10% of the running time of an algorithm is spent in code that cannot be parallelized, then the maximum possible speedup given infinite resources is a 10-fold speedup. For FlowGSP, there is no single portion of the algorithm which *must* be executed sequentially. There are, however, regions for which the overhead caused by work division and resource contention out-weighs the benefits of parallelization. Fortunately these regions occupy a small portion of the total execution time and therefore the potential for parallelism is great.

Both a multi-threaded shared memory implementation and a multiple-processor distributed memory system are presented. These two implementations are intended to leverage the common parallel architectures in use today. The distributed implementation is aimed at high-performance computing clusters that operate on a distributed memory model. The threaded implementation is aimed at workstations with multiple-core CPUs operating under a shared memory model. While clusters are the dominant platform for high-performance computing, workstations with 8 or more cores are becoming increasingly common. The opportunities presented by multi-core architectures should not be overlooked.

Section 5.1 discusses the decomposition of FlowGSP into parallel subproblems. Section 5.2 discusses the threaded implementation and Section 5.3 discusses the distributed implementation.

5.1 Parallel Decomposition

Prior to discussion of the two implementations, it is important to analyze how FlowGSP can be effectively partitioned into parallel subproblems. A proper parallel implementation of an algorithm is of little use if the underlying data can not be properly divided into parallelizable units of work.

Each iteration of FlowGSP can be broken down into the following steps:

1. Find all instances of candidate sequences in the graph and calculate the support of each instance.

2. Prune all candidate sequences that have S_M and S_D below the minimum support threshold.

3. Create the next generation of candidate sequences.

Step 1 is the most likely candidate for parallelization. It accounts for the majority of execution time and is easily decomposed. In fact, it is possible to decompose step 1 in a number of ways depending on the graph being mined. On one hand, work could be divided by assigning each worker a subset of the single-entry single-exit regions in the graph, each corresponding to a method in the profile. Work could also be divided by assigning each worker a subset of the current generation of candidate sequences.

Steps 2 and 3 could also be potentially parallelized. However, as indicated later in this chapter, these steps occupy a small portion of total execution time. Therefore the benefit to optimizing these steps is unclear.

Step 3 was not parallelized for either of the implementations. The decision not to parallelize candidate generation was made primarily due to the amount of communication that would be required to distribute the work and collect the results. All the candidates would need to be distributed to all of the worker threads because all possible pairs of candidate sequences must be checked for join compatibility. In addition, for later generations, the number of candidates generated could be potentially large. Aggregating the resulting lists of new candidates from all of the worker threads is also likely to be expensive. Furthermore, candidate generation occupies a small portion of execution time. For these reasons, the overhead of work distribution and result collection would likely outweigh the benefits of parallelization.

5.2 Threaded Implementation

This section presents a multi-threaded implementation of FlowGSP. This implementation is designed to leverage the popular trend towards increased use of multi-core CPUs in workstations and personal computers.

For the threaded implementation, work is divided by assigning each thread a subset of the methods to mine. Each thread searches for instances of all of the current candidate sequences within the methods assigned to it. To ensure that only one thread can update the support of a sequence at a time, the updating of support values for the candidates is handled via the Java `synchronized` primitive.

Work is assigned to the worker threads by a single master thread that is also in charge of pruning candidates with inadequate support and of calculating he next generation of candidates. For the threaded implementation neither of these operations are parallelized. Pruning is not parallelized due to the simplicity of the operation and to the extremely low percentage of total execution time it occupies.

All workers share a common database connection because the threaded version of the algorithm is run in a shared memory environment.

5.2.1 Work Division

As discussed, WebSphere Application Server is comprised of a very large number of small to medium-sized methods. Hence, a simple round-robin scheme is adopted to assign methods to worker threads. While this method is certainly not ideal in terms of achieving optimal load balancing among the threads, from our experiments it appears that this strategy is adequate. The vast majority of the methods are comprised of 50–250 instructions, with the largest method containing 725 instructions. Therefore, mining each method independently naturally results in the over-decomposition of the problem necessary for adequate load balancing.

5.2.2 Performance Analysis

The performance of the threaded implementation is evaluated by mining a WAS profile with the following parameters:

1. $g_{max} = 0$

2. $w_{max} = 0$

3. $n_{gen} = 10$

4. $s_{thresh} = 0.01$

The number of threads is varied from 1 (single-threaded) to 8, because the test machine is equipped with 2 quad-core CPUs. Testing with more than 8 threads is not likely to yield further speedup because there would be more threads than the number of available cores. FlowGSP was run in each configuration ten times in order to obtain statistically significant results in the presence of system noise. The data from our experiments is given in Tables 5.1 and 5.2. Each value reported is the mean of ten runs with a 95% confidence interval according to the Student's t-distribution. All values rounded to the nearest second. In all cases execution terminated after seven iterations as no candidates met the minimum support threshold.

Threads	**Execution Time**
1	30,717$\pm$271
2	16,965$\pm$50
3	12,703$\pm$105
4	10,631$\pm$255
5	8,800$\pm$82
6	16,982$\pm$162
7	16,373$\pm$89
8	6,726$\pm$23

Table 5.1: Total running time for FlowGSP, in seconds.
7

	Execution time (s)						
N	**Gen. 1**	**Gen. 2**	**Gen. 3**	**Gen. 4**	**Gen. 5**	**Gen. 6**	**Gen. 7**
1	1,743$\pm$7	10,829$\pm$150	6,319$\pm$71	6,712$\pm$80	3,546$\pm$25	1,192$\pm$6	373$\pm$2
2	926$\pm$3	5,942$\pm$29	3,539$\pm$16	3,748$\pm$17	1,974$\pm$5	626$\pm$2	208$\pm$2
3	679$\pm$3	4,428$\pm$44	2,649$\pm$28	2,807$\pm$28	1,517$\pm$5	451$\pm$3	168$\pm$2
4	580$\pm$23	3,654$\pm$103	2,210$\pm$62	2,340$\pm$57	1,295$\pm$326	391$\pm$44	155$\pm$5
5	478$\pm$5	3,017$\pm$43	1,826$\pm$18	1,946$\pm$27	1,069$\pm$8	311$\pm$2	148$\pm$1
6	460$\pm$32	2,982$\pm$198	2,895$\pm$198	4,925$\pm$342	4,079$\pm$29	2,038$\pm$21	105$\pm$0
7	412$\pm$32	2,546$\pm$12	2,544$\pm$13	4,313$\pm$25	3,913$\pm$19	1,965$\pm$8	106 $\pm$1
8	357$\pm$2	2,320$\pm$13	1,416$\pm$6	1,494$\pm$5	811$\pm$4	227$\pm$2	148$\pm$2

Table 5.2: Running times for FlowGSP by generation, in seconds, for N threads.

Figure 5.1 gives the speedup of the threaded implementation as a function of the number of threads. The error bars indicate a 95% confidence interval on the data. FlowGSP achieves excellent speedup when the number of threads is low, as indicated by Figure 5.1. However, the rate of speedup decreases as the number of threads increases.

Figure 5.2 breaks down the speedup by generation in order to identify the cause of the performance degradation. Again, the error bars indicate a 95% confidence interval. Figure 5.2 shows that

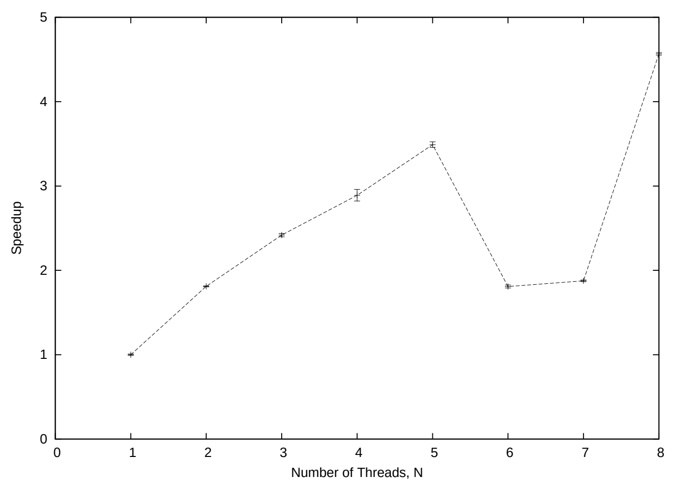


Figure 5.1: Total speedup as a function of the number of threads N.

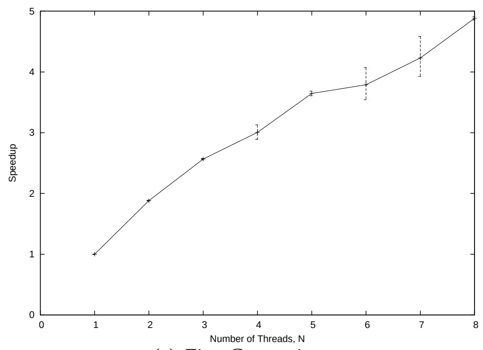
we obtained similar speedups on all iterations of the algorithm. Therefore we cannot isolate any particular iteration or iterations as a cause of the poor speedup in later generations.

Figure 5.2 also shows that when the algorithm is run with 6 or 7 threads the speedup obtained is extremely poor. The machine used for these performance runs is a part of a cluster of machines; it is possible that other nodes on the cluster experienced increased use during this period. Other nodes being used would effect the results of this experiment as all network traffic from each node is routed through a single master node. It is also possible that increased network congestion at the time of the experiment could infuence the execution time of the algorithm because the database server is hosted on a seperate machine. However, a repeated run of the algorithm with 7 threads again produced data consistent with the observations in Table 5.1. Therefore it is unclear exactly what is causing this particular performance anomaly.

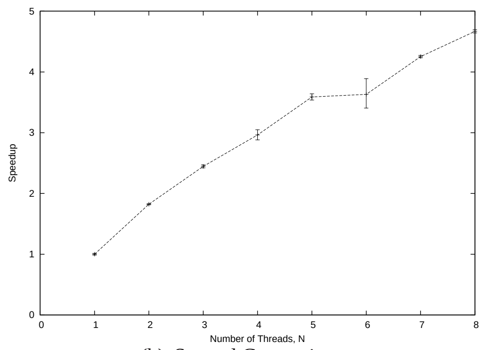
	Time (in seconds)		
Generation	**Mining**	**Fetching**	**Pruning/Joining**
1	338	19	0
2	2302	18	0.2
3	1394	20	0.1
4	1476	18	0.1
5	793	18	0
6	209	18	0
7	130	18	0

Table 5.3: Breakdown of execution time with 8 worker threads.

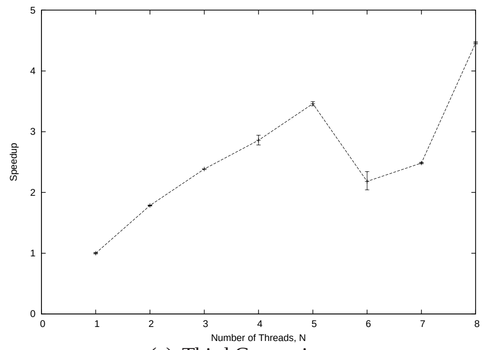
Table 5.3 breaks down the time spent mining each generation into three sections: time spent mining, time spent fetching data from the database, and time spent pruning and creating the next generation for 8 threads. The amount of time spent fetching data from the database is uniform accross all iterations of the algorithm. There is also a noticible increase in the amount of time spent



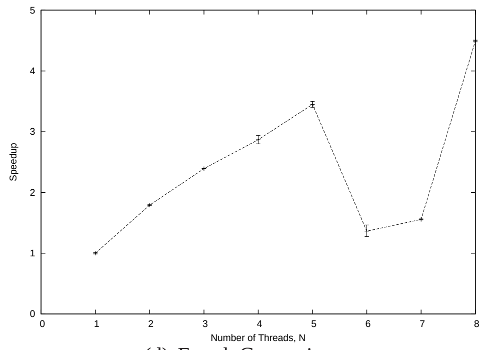
(a) First Generation



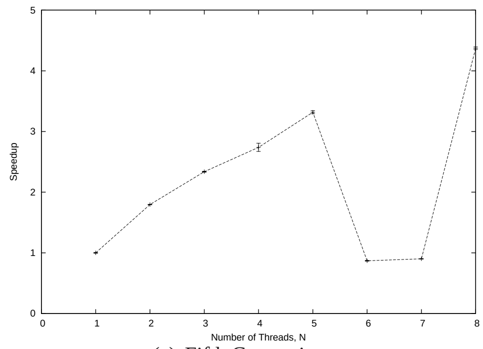
(b) Second Generation



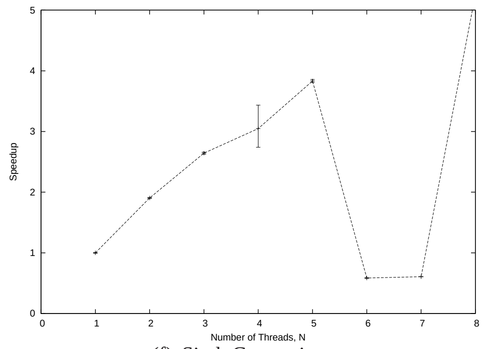
(c) Third Generation



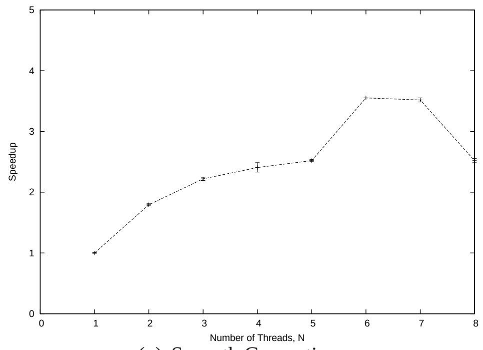
(d) Fourth Generation



(e) Fifth Generation



(f) Sixth Generation



(g) Seventh Generation

Figure 5.2: Speedup for the first through seventh generations as a function of the number of threads N.

Gap Size	Window Size	Execution Time (s)
0	0	6,726
0	1	14,068
1	0	12,748
1	1	22,314

Table 5.4: Influence of changing gap and window sizes on overall program performance.

pruning and joining the next generation of candidates, but this accounts for a small portion of total execution time.

The amount of time spent fetching data from the database is measured by starting and stopping a timer around each JDBC call, summing the time for all such calls from the same thread, and then averaging the totals over all worker threads. The amount of time spent pruning/joining is measured by a similar timer in the master thread. The time spent pruning/joining also includes the time spent fetching results from the worker threads. The time reported as mining time is the difference between the time for the mining task and the sum of the fetching and pruning/joining time.

The runs which produced the data in Tables 5.1 and 5.2 were obtained with a gap size of zero and a window size of zero. In many datasets a larger gap or window size may be required in order to discover interesting sequences. Table 5.4 shows the running time of FlowGSP on a Websphere Application Server profile when the gap and window sizes are increased. For this experiment FlowGSP was again run with 8 threads for ten iterations with a support threshold of 1%.

Increasing either the gap or the window size results in a significant increase in execution time. This increase can be explained by a general increase in the support values of candidate sequences and consequently more candidates meeting the support threshold. More candidates meeting the support threshold results in more candidates being generated at each iteration. Based on this data, it may be prudent to increase the support threshold when running FlowGSP with non-zero gap and window sizes in order to control the execution time. However, while increasing the support threshold is a good idea for Application Server profiles, this decision is ultimately dependant on the characteristics of the data being mined.

5.3 Distributed Implementation

This section describes a socket-based version of the algorithm for use in distributed memory machines. The scalability of the threaded implementation is limited by the number of cores on a single machine[1]. In order for the distributed implementation to be of use, it must scale beyond the capabilities of the threaded implementation. The distributed implementation described in this section does

[1]Most standard Java Virtual Machines (JVMs) do not allow for the distribution of threads between multiple machines.

not scale, therefore there is little reason to deploy this implementation of FlowGSP on a large cluster as opposed to single multi-core workstations.

5.3.1 Work Division

Similar to the threaded implementation, work is divided by a single master. Each worker is assigned an equal-sized subset of the candidate sequences in the current generation. Each worker then searches for instances of its subset of candidates over all of the methods in the profile. The decision to divide work in this manner was made after an initial attempt that divided work in the same manner as the threaded version. Information that needs to be transmitted from the master to the workers incurs a communication overhead because each worker has its own private address space. The amount of overhead for early generations is small because there are few candidates, but in later generations copying all candidates to each client is prohibitively expensive. Collecting results from workers is even more expensive: not only do the candidates need to be copied from each worker but the supports found by each worker need to be merged. Therefore, the method of work division was changed to reduce both the communication between workers and the computation needed to combine results from clients.

The only other difference between the socketed and threaded implementations is that the workers in the socketed implementation prune candidates with inadequate support prior to communicating the results of the mining to the master worker. This optimization was not implemented in the threaded version because all workers search for instances of all candidate sequences, therefore any pruning must wait until all workers have finished. This limitation is not present given this work distribution scheme and therefore it was convenient to implement this advanced pruning. It is unlikely, however, that this optimization resulted in any significant performance improvement.

5.3.2 Performance Analysis

The socketed implementation demonstrates extremely poor performance. In fact, runs with more than 2 workers usually resulted in a significant slowdown compared to the sequential case (in this instance, a run of the socketed implementation with only one worker apart from the master).

Table 5.5 gives the execution times for the distributed implementation for 1,2,4,8, and 16 workers. The data in Table 5.5 is from a single run of the distributed implementation and therefore may contain a significant amount of error. However, given that the distributed implementation fails to even achieve a speedup of two with sixteen workers, no further runs were performed and a 95%

	Execution Time (in seconds)					
N	**Gen. 1**	**Gen. 2**	**Gen. 3**	**Gen, 4**	**Gen. 5**	**Total**
1	697	1,434	2,240	6,666	10,437	21,478
2	455	975	1,673	4,565	7,345	15,018
4	370	625	1,313	4,986	9,088	16,388
8	506	556	1,016	5,199	10,696	17,978
16	684	647	816	3,974	11,178	17,304

Table 5.5: Execution time for the distributed implementation of FlowGSP with N workers.

confidence interval was not calculated. It is unlikely that the amount of error represents more than a small portion of total execution time.

In order to understand the behavior of the distributed implementation, the amount of time spent fetching data from the database was recorded. The time spent fetching data was recorded in the same manner as the threaded implementation. The data shows that, for 16 workers, 1,688 seconds were spent fetching data from the database. It is unlikely that the fetching of data from the database is the bottleneck for the distributed implementation because the amount of time spent fetching accounts for only 9.7% of total execution time.

The amount of time spent retrieving results from workers, pruning candidates with insufficient support, and creating the next generation of candidates was also recorded. The amount of time spent on these activities accounts for twelve seconds. Twelve seconds is a relatively insignificant portion of the total execution time listed in Table 5.5.

It seems likely that poor granularity is the reason the distributed implementation experiences poor performance. The current method of work division was implemented only after the work-division scheme used in the threaded implementation was used unsuccessfully. Neither of these methods of work division result in good parallel performance. Therefore further work is required to determine an appropriate method of dividing the mining problem among the workers. Given the relatively good performance of the threaded implementation, the discovery of such a method is left to future work.

Final Remarks

A threaded and distributed (*via* sockets) implementation of FlowGSP has been discussed. The threaded version achieved a 78% decrease in total execution time. The threaded implementation of FlowGSP reduces execution time to the point where multiple runs can be easily performed in a day, a critical feature if FlowGSP is to be used in a production compiler environment. It may be possible with further investigation to further increase the performance of the threaded implementation.

The distributed version, unfortunately, achieves little to no speedup because the database server is unable to cope with the volume or frequency of requests. It is possible that with more resources the performance of both implementations of FlowGSP could see significant improvement. The socketed implementation is of no use because it is outperformed by the threaded implementation.

Chapter 6

Mining WebSphere Application Server Profiles with FlowGSP

The experimental results presented in this Chapter demonstrate that FlowGSP works in the context of mining WebSphere Application Server profiles, a large enterprise application. The execution paths in this server consist of millions of assembly instructions per transaction and a plethora of hardware events per instruction. These results establish FlowGSP as a practical and effective solution for the mining of large flow graphs.

6.1 Discovery of Previously Known Patterns

Before the development of FlowGSP, compiler developers were faced with the difficult challenge of identifying patterns in the execution paths of large enterprise applications using nothing but intuition and observation. This approach to discovery is not only tedious and time consuming, but is also fraught with limitations of capturing the scope and respective support for improvement opportunities. The intuitive approach may lead to large investments in compiler development effort that may not necessarily pay off. Countless person-hours of effort have been invested in such discovery processes.

With the implementation of FlowGSP, an interesting acid test of the automatic approach is to discover patterns that had already been identified manually by compiler developers. FlowGSP passed this acid test because it was able to identify all the patterns that were known to the developers. Some of these patterns include:

- $\langle(Icachemiss, TLBmiss)\rangle, S_M = 0.529$ indicates a high correlation between instruction-cache misses and TLB misses on the host architecture.

- $\langle(Prologue, Icachemiss)\rangle, S_M = 0.1175$ indicates a high occurrence of instruction-cache misses in the prologues of methods. This is significant considering that the sequence $\langle(Prologue)\rangle$ has $S_M = 0.120$.

- $\langle(JITtarget, Icachemiss)\rangle, S_M = 0.0935$ corresponds to a significant number of instruction-cache misses on the JIT target instructions. The JIT target instruction is the first assembly instruction to be called when the method is called from natively compiled code. In general, a different first instruction is executed when the method is called from interpreted code. The level of support for this attribute pair is even more significant because the sequence $\langle(JITtarget)\rangle$ has $S_M = 0.0935$.

6.2 Discovery of New Characteristics

In addition to re-discovering known patterns, FlowGSP was effective in identifying an opportunity to improve WebSphere Application Server performance by enabling large pages. By discovering a high incidence and correlation of instruction cache and TLB misses, FlowGSP helped identify the use of large pages as a performance opportunity. As such, enabling large pages resulted in a 3-4% decrease in instruction-cache misses and an overall throughput improvement of 2%.

FlowGSP was also successful at discovering many expected patterns. For instance, it is known that the instruction-cache-miss counter exhibits a long tail because the counter continues to register for multiple instructions following the instruction that incurred the actual delinquent cache fetch. FlowGSP identified good support for the sequence $\langle(Icachemiss), (Icachemiss)\rangle, SM = 0.112$. A result that confirms this *a priori* knowledge. FlowGSP also identified a high occurance of sequences which contained data cache misses and TLB misses, a result that is both typical and expected.

The sequence $\langle(Icachemiss, Branchmispredict)\rangle, S_M = 0.240$ shows that there is a correlation between branch mispredictions and instruction-cache misses. This observation suggests that it may be possible to improve the hardware's branch predictor to improve WebSphere Application Server performance.

Final Remarks

This chapter demonstrated that FlowGSP is able to identify known patterns in an EFG constructed from a WebSphere Application Server profile. These patterns were identified with a level of support proportional to their known significance. This chapter also showed that FlowGSP is capable of identifying previously unknown sequences in an EFG, and that these sequences can be used to achieve performance improvements in the target application.

Chapter 7

Related Work

There has been a significant body of work recently investigating the possible applications of machine learning techniques to compiler research. The application of machine learning could potentially relieve many person-hours of intensive, skilled labor because compilers rely heavily on hand-tuned heuristics. Section 7.1 discusses such applications.

Data mining techniques have been extended to mine data in a large variety of formats. While, to the best of our knowledge, no algorithm exists that can mine EFGs, there are a number of algorithms that mine similar types of data. These algorithms are discussed in Section 7.2.

There have also been a number of efforts to improve compiler performance through traditional means based on hardware profiling information. Section 7.3 discusses some of these approaches.

Enterprise applications, such as the WebSphere Application Server, usually carry a very large load in terms of responsibility for many businesses. Therefore, interest in improving their performance is common. Other approaches that attempt to improve the performance of large Java applications are covered in Section 7.4.

While all of these approaches have some elements in common with FlowGSP, none of them are uniquely equipped to handle the challenges faced when mining for frequent sequences in hardware profile data.

7.1 Machine Learning and Compilers

A common problem with trying to achieve optimal performance with modern optimizing compilers is that it is extremely difficult to predict the ideal optimization parameters for any given program. Moreover, the number of possible parameter values for most optimizing compilers is extremely large. Manually searching through them all, *i.e.* through iterative optimization, is prohibitively expensive. Compiler researchers have begun to turn to the field of machine learning in order to more efficiently find good optimization configurations.

7.1.1 Supervised Learning

There have been a number of efforts into using predictive models to improve the performance of optimizing compilers.

Cavazos *et. al.* present a method for automatically selecting good compiler optimizations via a model constructed using performance-counter data [6]. They evaluate a variety of optimization configurations on a training set of programs and use the resulting speedups relative to a baseline optimization level to construct a predictive model. Given a novel program, they compile it once using the baseline compiler options. The performance-counter data from this run is fed back into the model to produce a set of "best" optimizations. They test their system using cross-validation on the PathScale EKOPath compiler and the Standard Performance Evaluation Corporation (SPEC) 1995 benchmark suite. N-fold Cross-validation involves splitting the data set into N sections, then training on N-1 sections and testing on the remaining section. Cavazos *et. al.* are able to achieve a 17% performance increase over the fastest built-in optimization setting (`-Ofast`). While they do produce excellent results, they are only exploring the space of all pre-existing optimization settings. FlowGSP allows the discovery of new opportunities for code transformations that may not be within the scope of existing compiler flags. It would not be possible for their system to discover any of these opportunities.

Stephenson *et al.* use supervised learning techniques to determine the ideal loop unrolling factor for a program [33]. They use both nearest-neighbor and Support Vector Machine (SVM) classifiers [9] in their research. Using these methods they are able to improve loop unrolling performance by 5% on the SPEC 2000 benchmark suite [8]. While an adapted version of their tool could be used to discover good parameter values for existing optimization, their method is unable to discover opportunities for new code transformations, or even if the current set of parameters to existing options is wanting.

7.1.2 Unsupervised Learning

Stephenson *et al.* (2003) use Genetic Programming to determine ideal parameters for hyperblock formation, data prefetching, and register allocation [34]. Genetic Programming uses the principles of genetics to “evolve” more suitable candidates over multiple generations. However there is some doubt as to the usefulness of genetic algorithms in cases such as these, especially when their performance is compared to that of pure random search [4].

7.2 Data Mining

Mining for frequent sequences in structured data has its roots in algorithms such as GSP [2], PrefixSpan [13], and WINEPI/MINEPI [20]. These algorithms all search for sequences or partial sequences within a totally ordered dataset. While this assumption of a total ordering is sufficient for some applications, such as market-basket analysis, there are many examples of real-world data, such as hardware profiles, that cannot be expressed using a total ordering.

One of the most commonly used data structures in both computing science and mathematics is a graph. It should come as no surprise that there has been a significant amount of work devoted toward developing mining algorithms that operate on topologically ordered data structures such as graphs. AGM [16], gSpan [39], Origami [14], and Gaston [23] are all algorithms that search for frequent substructures in graph-based data.

While mining for common substructures is a very similar goal to that accomplished by FlowGSP, there are some crucial differences. First, FlowGSP is interested in discovering frequent sequences of attributes on the vertices of a flow graph. The mining performed by FlowGSP is more than just frequent substructure discovery because a frequent connected substructure need not be a sequence of vertices. Also, none of the cited works handle multiple labels or attributes on vertices.

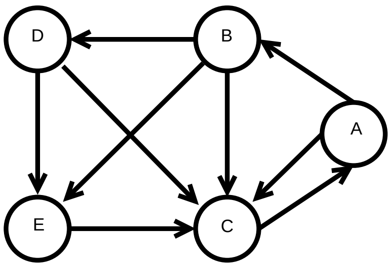
A traversal of a graph is an ordered sequence of connected vertices in a graph. There are a number of real-world situations for which finding frequent sequences in a collection of graph traversals may prove useful, such as analyzing the behavior of visitors to a web site.

Lee *et al.*develop an algorithm for discovering frequent patterns in traversals of a weighted, directed graph [19]. A traversal of a graph is an ordered sequence of connected vertices in the underlying graph. They evaluate their algorithm on a number of randomly generated graphs.

Geng *et al.* propose another algorithm for mining frequent patterns in traversals of graphs with weighted edges and vertices [11]. They used a generate-and-test approach based on the Apriori algorithm [30] to grow their candidate patterns.

The key difference between these traversal-mining algorithms and FlowGSP is the form of the data being mined. Both Lee *et al.* and Geng *et al.* mine a collection of traversals over a graph. This difference is illustrated by Figure 7.1. Subfigure (a) is an example of a directed graph G; subfigure

(b) gives a list of weighted traversals. Each traversal consists of a sequence of nodes from G (under the traversal column) and a series of weights assigned to each edge of the traversal (under the weight column). This figure is adapted from Figure 1 in [19]. Traversals such as these are not the same as the EFGs mined by FlowGSP: FlowGSP mines for frequent possible subpaths in a graph.



(a) G

ID	Traversal	Weight
1	$< A, B, C >$	$< 2.2, 2.0 >$
2	$< B, D, E, C, A >$	$< 3.0, 4.3, 3.5, 3.1 >$
3	$< C, A, B, D >$	$< 2.9, 2.0, 4.0 >$
4	$< D, C, A >$	$< 4.0, 3.0 >$
5	$< B, C, A >$	$< 2.2, 2.9 >$
6	$< A, B, E, C >$	$< 1.4, 3.9, 4.4, 3.2 >$

(b) Traversals on G

Figure 7.1: Example of a directed graph and a weighted traversal.

Hwang *et al.* perform data mining on program call graphs in order to identify recurring patterns in method call sequences [15]. They successfully identify two commonly occurring “control patterns” that occur frequently together in a collection of Java programs. Hwang *et al.* only provide the mechanism for identifying these control patterns and do not provide insights into how this information may be used to increase program performance. Also, their work relies on obtaining program traces rather than profiles. The cost of obtaining a full trace for an application as large as the WebSphere Application Server would be prohibitively expensive. Their method also only uses the sequence of method calls and does not incorporate low-level or performance-counter information.

Pawlak develops a method for mining association rules from labelled, weighted, flow graphs [24]. They derive these rules by “fusing” the graph being mined and combining the normalized edge weights. Through fusing they are able to obtain association rules between the graph’s sources and sinks by fusing out all the intermediary nodes. Such an approach does not map well to mining control-flow-graph data because each method has only a single source and single sink node. Also, patterns that occur within the internal structure of a flow graph are of higher interest.

Moseley *et al.* develop Optiscope, a tool for comparing multiple hardware profiles [21]. Their goal is to allow easy comparison of multiple executions of the same program under various compiler configurations. Loops in each profile are matched, and the differences in the profiling information examined via a web interface. The profiling data from each loop or method is aggregated and compared. The local focus of Optiscope is in sharp contrast to the goal of finding global patterns that may yield performance opportunities. Optiscope is built around basic slicing operations. That is to say that optiscope only performs basic aggregations; no search for frequent patterns or sequences is performed.

7.3 Performance Counters

Choi *et al.* use performance counter data to enhance the performance of the Intel compiler on the Itanium 2 platform [7]. They use performance counter data to supplement the information usually obtained during static FDO. However, as is usual for FDO, their process is restricted to improving the performance of existing code transformations and cannot aid in the discovery of new code transformations.

There have been a growing body of research directed towards allowing JIT compilers access to hardware profiling data at run-time. Schneider *et al.* collects information about instruction cache misses on the Intel Pentium 4 platform in order to increase performance via object co-allocation [27]. Cuthbertson *et al.* also use I-cache information from hardware event counters on the Intel Itanium platform to improve co-allocation, as well as to influence global scheduling [10]. Shye *et al.* collect branching information from the Itanium Branch Transition Buffer (BTB) and use this information to build partial path profiles of the executing program [28]. All of these works only operate on a very small number of performance counters. While a small number of counters is enough data for the code transformations they investigate, by discarding some performance counters, information about the program is lost. It may not be possible to simply increase the amount of counter information passed into the JVM, because it is not clear how well the strategies presented will scale as the number of counters retrieved increases.

Buytaert *et al.* use hardware-performance counters (in particular sampling ticks) to increase the accuracy and decrease the cost of the instrumentation used to detect hot methods in a Java JIT compiler [5]. No new code transformations are introduced because their work is focused on improving the performance of current techniques.

7.4 Enterprise Application Performance

Xu *et al.* investigate object copying as a symptom of bloat in large Java applications [38]. By profiling object copy behavior they are able to hand-tune their applications and significantly increase performance. They are able to decrease the execution time of two DaCapo benchmarks by 65% and 9.3% respectively. While Xu *et al.* also investigate large scale Java application performance their approach is focused on hard tuning the resulting application instead of searching for opportunities for new compiler code optimizations. While application tuning is important for overall performance, the role of the compiler is equally important. FlowGSP addresses the needs of the compiler developer when it comes to improving the performance of enterprise applications.

Conclusion

This thesis presented the EFG, a data structure that represents the information contained in hardware profiles. This data structure was designed by supplementing the profile data with the CFG taken from the compiler logs. This thesis defined the notions of frequency and weight support of a subpath in an EFG. It also presented FlowGSP, an algorithm to mine for sequences in an EFG. FlowGSP ranks sequences by both their frequency in the graph and by the cost of the subpaths in which they appear.

EFGs were constructed using profiling information obtained from WebSphere Application Server runs on the IBM z10 architecture. A list of attributes was created to represent the characterstics of z10 profiling data. This thesis presented an implementation of FlowGSP in Java to mine these EFGs.

Two parallel implementations of FlowGSP were also introduced. The first implementation uses Java threads and is targeted at the rapidly growing domain of multi-core workstations. The threaded implementation of FlowGSP reduces the time required to mine a profile of the WebSphere Application Server by roughly 75%. This improved performance allows FlowGSP to mine such a profile many times in a day, a necessary requirement for deployment in a production compiler development environment. This thesis also presented a distributed implementation of FlowGSP targeted at cluster-based computing. However, the distributed implementation failed to show the performance necessary to merit deployment. Future work may yield a better work-division strategy for this implementation.

FlowGSP was evaluated on WebSphere Application Server hardware profiles. This thesis presented a number of sequences discovered by FlowGSP which characterize known Application Server behavior. The discovery of previously unknown sequences was also discussed. These new sequences may yield new performance opportunities upon further study.

There are a number of areas for future development based on the work in this thesis. The parallel implementations of FlowGSP discussed in Chapter 5 can be greatly improved. The threaded implementation suffers from a database bottleneck. Adding additional database servers or implementing some caching protocol may alleviate this bottleneck. The distributed implementation exhibits poor granularity and does not scale to a large number of nodes. The development of an alternate method of work division which does not exhibit this poor granularity would increase the viability of this implementation.

There are a number of newer data mining algorithms, PrefixSpan [25] for example, that have improved performance compared to GSP. GSP was chosen as the basis for FlowGSP due to its simplicity and the fact that it is well studied. Other sequence-mining algorithms could be extended to achieve the same goals as FlowGSP, perhaps with greater efficiency. It may also be possible to extend a graph mining algorithm such as Gaston [23] or gSpan [39] to search for frequent sequences in an EFG. These algorithms have differing characteristics and it is possible that using an algorithm other than GSP as the base to mine EFGs for data may result in considerable performance improvements.

FlowGSP has been extensively tested on WebSphere Application Server profiles. However, the WebSphere Application Server is hardly representative of all programs. Other application profiles could yield patterns of interest to compiler developers with FlowGSP's help.

EFGs are not limited to hardware profiles. There are many other domains that could be characterized in a manner similar to performance-counter data. Web traffic pattern analysis, urban traffic planning, and purchasing pattern analysis are examples of areas that could benefit from the application of FlowGSP.

Bibliography

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

[2] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *International Conference on Data Engineering (ICDE)*, pages 3–14. IEEE Computer Society, March 1995.

[3] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Inf. Process. Lett.*, 82(1):7–13, 2002.

[4] José Nelson Amaral, Adalberto Tiexeira Castelo Neto, and Alessandro Valerio Dias. Genetic algorithms in optimization: Better than random search? In *1997 International Conference on Engineering and Informatics*, pages 320–326, April 1997.

[5] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using HPM-sampling to drive dynamic compilation. volume 42, pages 553–568. ACM, 2007.

[6] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization (CGO)*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson. Design and experience using the Intel Itanium 2 processor performance monitoring unit to implement feedback optimizations. *EPIC2 Workshop*, 2002.

[8] Standard Performance Evaluation Corporation. SPEC benchmark. http://www.spec.org.

[9] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *J. Mach. Learn. Res.*, 2:265–292, 2002.

[10] John Cuthbertson, Sandhya Viswanathan, Konstantin Bobrovsky, Alexander Astapchuk, and Eric Kaczmarek. Uma Srinivasan. A practical approach to hardware performance monitoring based dynamic optimizations in a production JVM. In *Code Generation and Optimization (CGO)*, pages 190–199, Seattle, WA, USA, 2009. IEEE Computer Society.

[11] Runian Geng, Xiangjun Dong, Xingye Zhang, and Wenbo Xu. Efficiently mining closed frequent patterns with weight constraint from directed graph traversals using weighted FP-tree approach. In *International Colloquium on Computing, Communication, Control, and Management*, pages 399–403, Guangzhou City, China, August 2008.

[12] M. Golden and T. Mudge. Comparison of two common pipeline structures. *Computers and Digital Techniques, IEE Proceedings*, 143(3):161–167, May 1996.

[13] J. Han, J. Pei, and Y. Yin. *Sequential Pattern Mining by Pattern-Growth: Principles and Extensions*, volume 180 of *Studies in Fuzziness and Soft Computing*, pages 183–220. Springer, 2005.

[14] Mohammad Al Hasan, Vineet Chaoji, Saeed Salem, Jeremy Besson, and Mohammed J. Zaki. Origami: Mining representative orthogonal graph patterns. In *International Conference on Data Mining (ICDM)*, pages 153–162, Omaha, NE, USA, 2007. IEEE Computer Society.

[15] Chung-Chien Hwang, Shih-Kun Huang, Deng-Jyi Chen, and D.T.K. Chen. Object-oriented program behavior analysis based on control patterns. In *Asia-Pacific Conference on Quality Software (APCQS)*, pages 81–87, Hong Kong, China, December 2001.

[16] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23, London, UK, 2000. Springer-Verlag.

[17] K.M. Jackson, M.A. Wisniewski, D. Schmidt, U. Hild, S. Heisig, P. C. Yeh, and W. Gellerich. IBM system z10 performance improvements with software and hardware synergy. *IBM J. Res. Dev.*, 53(1-16), May 2009.

[18] T. Kisuki, P.M.W. knijnenburg, Michael F. P. O'Boyle, Francois Bodin, and E. Rouhu. Iterative compilation in a non-linear optimization space. In *Parallel Architectures and Compilation Techniques (PACT)*, Paris, France, October 1998.

[19] Seong Dae Lee and Hyu Chan Park. Mining frequent patterns from weighted traversals on graph using confidence interval and pattern priority. *International Journal of Computer Science and Network Security (IJCSNS)*, 6(5A):136–141, May 2006.

[20] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering Frequent Episodes in Sequences. In U. M. Fayyad and R. Uthurusamy, editors, *Knowledge Discovery and Data Mining (KDD)*, Montreal, Canada, 1995. AAAI Press.

[21] Tipp Moseley, Dirk Grunwald, and Ramesh V. Peri. Optiscope: Performance accountability for optimizing compilers. In *Code Generation and Optimization (CGO)*, Seattle, WA, USA, 2009. IEEE Computer Society.

[22] Priya Nagpurkar, Harold W. Cain, Mauricio Serrano, Jong-Deok Choi, and Ra 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 (CAECW)*, Phoenix, AZ, USA, 2007.

[23] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Knowledge Discovery and Data Mining (KDD)*, pages 647–652, New York, NY, USA, 2004. ACM.

[24] Zdzislaw Pawlak. Flow graphs and data mining. In *Transactions on Rough Sets III*, volume 3400/2005 of *Lecture Notes in Computing Science*, pages 1–36. Springer, 2005.

[25] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In *International Conference on Data Engineering (ICDE)*, pages 215–226, Heidelburg, Germany, 2001.

[26] K.E. Plambeck, W. Eckert, R. R. Rogers, and C. F. Webb. Development and attributes of z/architecture. *IBM J. R. Dev.*, 46(4-5):367, 2002.

[27] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. In *Programming language design and implementation (PLDI)*, pages 373–382, New York, NY, USA, 2007. ACM.

[28] Alex Shye, Matthew Iyer, Tipp Moseley, David Hodgdon, Dan Fay, Vijay Janapa Reddi, and Daniel A. Connors. Analyis of path profiling information generated with performance monitoring hardware. In *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 34–43, Rome, Italy, 2005. IEEE Computer Society.

[29] T. J. Siegel, E. Pfeffer, and J. A. Magee. The IBM eServer z990 microprocessor. *IBM J. Res. Dev.*, 48(3-4):295–309, 2004.

[30] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *SIGMOD International Conference on Management of Data*, pages 1–12, New York, NY, USA, 1996. ACM.

[31] Ramakrishnan Srikant and Rakesh Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements*, pages 3–17. Advances in Database Technology. Springer, 1996.

[32] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generatlizations and performance improvements. Technical report, IBM Research Division, Almaden Research Center, San Jose, CA, USA, 1996.

[33] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Code Generation and Optimization (CGO)*, pages 123–134, San Jose, CA, USA, 2005. IEEE Computer Society.

[34] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Programming language design and implementation (PLDI)*, pages 77–90, New York, NY, USA, 2003. ACM.

[35] Robert Tibshirani and Jerome Friedman Trevor Hastie. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, 2001.

[36] E. Tzortzatos, J. Bartik, and P. Sutton. IBM system z10 support for large pages. *IBM J. Res. Dev.*, 53(1-17), May 2009.

[37] C. F. Webb. IBM z10: The next generation microprocessor. *IEEE Micro*, 28(2):19–29, March 2008.

[38] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 419–430, Dublin, Ireland, 2009. ACM.

[39] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *International Conference on Data Mining (ICDM)*, page 721, Maebashi City, Japan, 2002. IEEE Computer Society.