Mining Opportunities for Code Improvement in a Just-In-Time Compiler

Adam Jocksch[1], Marcel Mitran[2], Joran Siu[2],
Nikola Grcevski[2], and José Nelson Amaral[1]

[1] Department of Computing Science
University of Alberta, Edmonton, Canada
{ajocksch,amaral}@cs.ualberta.ca
[2] IBM Toronto Software Laboratory, Toronto, Canada

Abstract. The productivity of a compiler development team depends on its ability not only to the design effective solutions to known code generation problems, but also to uncover potential code improvement opportunities. This paper describes a data mining tool that can be used to identify such opportunities based on a combination of hardware-profiling data and on compiler-generated counters. This data is combined into an Execution Flow Graph (EFG) and then FlowGSP, a new data mining algorithm, finds sequences of attributes associated with subpaths of the EFG. Many examples of important opportunities for code improvement in the IBM® Testarossa compiler are described to illustrate the usefulness of this data mining technique. This mining tool is specially useful for programs whose execution is not dominated by a small set of frequently executed loops. Information about the amount of space and time required to run the mining tool are also provided. In comparison with manual search through the data, the mining tool saved a significant amount of compiler development time and effort.

1 Introduction

Compiler developers continue to face the challenges of accelerated time-to-market and significantly reduced release cycles for both hardware and software. Micro-architectures continue to grow in numbers, complexity, and diversity. In this evolving technological environment, commercial-compiler developing teams must discover and rank the next set of opportunities for code transformations that will provide the highest performance improvement per development cost ratio.

The discovery of opportunities for profitable code transformations in large enterprise applications presents additional challenges. Traditionally, compiler developers have relied on the intuition that the code that is relevant for performance improvement is located in easily identifiable, frequently executed, regions of the code — often called *hot loops*. However, many enterprise applications do not exhibit discernible regions of frequently executed code. Rather, these applications exhibit a *flat profile*: thousands of methods are invoked along an execution path, and no single method accounts for a significant portion of the

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 10–25, 2010.

execution time — even though a typical transaction executes millions of instructions. Thus, focusing development effort on any single method provides negligible overall performance improvement. However, these applications may display code patterns that appear repeatedly throughout the code base. Even though no single instance of such a pattern is executed frequently, the aggregated run time of the pattern may be significant. Applications with flat profiles are becoming increasingly important for commercial compilers that are used to generate code for middleware and enterprise information-technology (IT) infrastructure.

Thus, a challenge when developing a compiler for applications with flat profiles is to discover code patterns whose aggregated execution time is significant so that development efforts can be focused into improving the code generation for such patterns. This paper describes a data mining infrastructure, based on the recently developed FlowGSP algorithm [13], which can be used for automatic analysis of code compiled by the IBM Testarossa Just-in-Time (JIT) Compiler [8]. This infrastructure was used to discover patterns in the code generated for applications running in the IBM® WebSphere® Application Server and for SPECjvm2008 [20] running under Linux® for System Z® [22,19].

WebSphere Application Server is a fully compliant Java™ Enterprise Edition (JEE) application server written in Java code [11]. This paper uses the DayTrader Benchmark in the WebSphere Application Server[7]. This benchmark produces a typical WebSphere Application Server profile reporting the compilation of thousands of methods, with no method representing more than 2% of the total execution time. For instance, cache misses represent 12% of the overall run time in one run of a certain application in application server. But, to account for 75% of the misses requires the aggregation of misses from 750 different methods [8].

SPECjvm2008 exemplifies the growing variety of industry standards that are quickly expanding the scope of benchmarks. The SPECjvm2008 suite comprises more than double the number of benchmarks that were in its predecessor, SPECjvm98 [18]. Some of the benchmarks in the newer suite have flat profiles, making the analysis and identification of opportunities for code improvement more difficult, more tedious and more indeterminate.

The IBM Testarossa JIT compiler ships as part of the IBM Developer Kit for Java which powers thousands of mission-critical applications on everything from embedded devices, to desktops, to high-end servers. The IBM Testarossa JIT is a state-of-the-art commercial compiler that offers a very complete set of traditional OO-based and Java-based optimizations. As a dynamic compiler, Testarossa is also equipped with a sophisticated compilation control system for online feedback-directed re-compilation [21].

The analysis presented in this paper was performed on Linux for System z. System z10™ is the latest and most powerful incarnation of IBM's mainframe family, which continues to provide the foundation for IT centers for many of the world's largest institutions. The System z10 processor has a 4.4 GHz dual core super-scalar pipeline, executes instructions in order, and can be characterized as an address-generation-interlocked pipeline. This processor is a complex

instruction set computer with a rich set of register-to-register, register-to-storage, storage-to-storage, and complex branching operations, in addition to hardware co-processors for cryptography, decimal-floating-point, and Lempel-Ziv compression [22]. The System z10 processor also provides an extensive set of performance-monitoring counters that can be used to examine the state of the processor as it executes the program.

The data mining infrastructure was applied to a large set of compiler attributes and hardware counters. The attributes and hardware data are organized in a directed graph representing program flow. Edge frequencies are used to represent the probabilistic flow between basic blocks. The FlowGSP algorithm is general and can mine any flow graph. A vertex in this flow graph may represent any single-entry-single-exit region such as an instruction, a basic block, a bytecode, or a method. Attributes are associated with each vertex, and the algorithm mines for sequences of attributes along a path.

The main contributions of this paper are:

- An introduction of the problem of identifying important code patterns that occur in applications with flat profiles, such as enterprise applications.
- A description of a new data mining framework that can be used to discover important opportunities for code generation improvement in a commercial dynamic compiler environment.
- A demonstration of the effectiveness of the data mining tool through the narrative of several discoveries in the code generated for the System z architecture by the IBM Testarossa compiler.
- Statistics on space and time requirements for the usage of the mining tool in this environment. This information should be relevant for other compiler groups that wish to implement a similar tool, as well as for researchers that wish to improve on our design.

Section 2 explains the need for the mining tool through the description of one of the important discoveries in a very common segment of code. The mining tool is described in Section 3. Several additional improvement opportunities discovered by the tool are described in Section 4. Experimental data describing the time and space requirements for the usage of the tool in the Testarossa environment is presented in Section 5. Section 6 discusses previous work related to the development of similar analysis tools.

2 Motivating Case Study

This section outlines the motivation for the use of data mining to discover patterns that account for significant execution time by describing one such pattern discovered by FlowGSP. The data mined by FlowGSP to discover this pattern includes, instruction type, execution time, cache misses, pipeline interlock, *etc* [13]. This pattern is part of the array-copy code generated by Testarossa for the System z10 platform. FlowGSP identified that, in some benchmarks, more than 5% of the execution time was due to a single instruction called `execute` (EX).

This finding is surprising because the IBM Testarossa JIT compiler uses this instruction in only one scenario – to implement the tail-end of an array copy.[1] More specifically, a variable-length array copy is implemented with a loop that executes an MVC (Move Characters) instruction. The MVC instruction is very efficient at copying up to 256 bytes. The 256-byte copy length is encoded as a literal value of the instruction. Figure 1 shows the code generated for array copying. Any residual of the copy after the repeated execution of MVCs is handled by using the EX instruction. The EX instruction executes a target instruction out of order. Before executing the target, EX replaces an 8-bit literal value specified in the target with an 8-bit field from a register specified in EX. The overloading is done through an OR of the two bit fields. For the residual array-copy code generated by Testarossa, the register specified in EX contains the length of the residual array and the target instruction is a MVC instruction.

```
            Rsrc = Address of source array;
            Rtrgt = Address of target array;
            while(Rlength >= 256)
                  MVC  Rsrc, Rtrgt, 256
                   Rsrc = Rsrc + 256;
                   Rtrgt = Rtgt + 256;
                   Rlength = Rlength - 256;
                  }
            EX ResLabel, Rlength, mvcLabel;
            ...
ResLabel: MVC Rsrc, Rtrgt, 0
```

Fig. 1. Pseudo-assembly code for array copy

After the data mining tool identified that 5% of the time was spent in EX, we examined the profiling data more carefully to find out that the 5% of time spent in EX is spread over several methods. Therefore, the time spent in the EX instruction would not be apparent from a study of individual methods. Moreover, part of that time is spent in the MVC instruction. Nonetheless, the EX instruction incurs significantly more misses in the data-cache and the translation-look-aside-buffer (TLB) misses than expected. There are two potential reasons for this:

1. The length of many array copies is less than 256 byte long. In this case, data cache misses would occur while fetching the source/target operands of MVC.
2. The EX instruction misses the cache upon fetching the overloaded MVC. This miss occurs because the targeted MVC instruction is located next to other insructions used by the program, and hence resides in the instruction cache. On a z10, the EX instruction needs the targeted MVC in the data

[1] Array copies use 256-byte copy instructions, the tail-end is any final portion of the copy that is smaller than 256 bytes.

cache. Moving the targeted MVC from the instruction cache to the data cache incurs an extra cost that was not apparent to the compiler designers.

This discovery started an important review of the array-copy code generated by the compiler. A suitable strategy must be designed to isolate the targeted MVC from the other data values that are located around it. This strategy must take into consideration the long lines in the architecture.

An important question is why there is the need for a data mining tool to discover such an opportunity. Could simple inspection of the hardware and compiler profiling data reveal this opportunity? Even if a developer were to spot the cache miss caused by the EX instruction, she would have no way to know that the aggregation of occurrences of EX in many infrequently executed methods is amount to significant performance loss that needs to be addressed. Even though profile logs of code generated by this commercial compiler had been inspected by hand for many years, the issue with the use of EX and MVC for array copy had never been regarded as worthy of attention from the team. Once the mining tool reported it, one of the developers remarked: "Now we can see!".

3 The Mining Tool

The mining tool design is based on a new data mining algorithm called FlowGSP. FlowGSP mines for subpaths in an execution flow graph (EFG). Jocksch formally defines a an EFG as a directed flow graph possibly containing cycles [13]. Each EFG vertex is annotated with a normalized weight and has an associated list of attributes. Each EFG edge is annotated with a normalized execution frequency. A subpath is of interest if either its frequency of execution, called *frequency support*, or vertex weights, called *weight support*, is above a set threshold. A subpath is also of interest if the difference between its frequency and weight support is higher than a *difference support*. FlowGSP reports sequences of attributes whose aggregated support over the entire EFG is higher than the specified supports.

FlowGSP is an extension of the Generalized Sequential Pattern (GSP) algorithm, originally introduced by Agrawal *et al.* [1]. The main difference between FlowGSP and GSP is that GSP was designed to mine for sequences of attributes in a list of totally ordered transactions while FlowGSP enables the mining for sequences of attributes in subpaths of a flow graph, thus allowing a partial order between the transactions (vertices in the EFG). Similar to GSP, FlowGSP allows for windows and gaps. A window allows attributes that occur in distinct vertices that are close in a subpath — within the specified window — to be regarded as occurring in the same vertex. A gap is a maximum number of vertices in the subpath that do not contain attributes in the sequence.

3.1 Preparation of Data for Mining

The overall architecture and flow in a system that uses FlowGSP for mining is shown in Figure 2. Performance-counter data generated by the hardware [12]

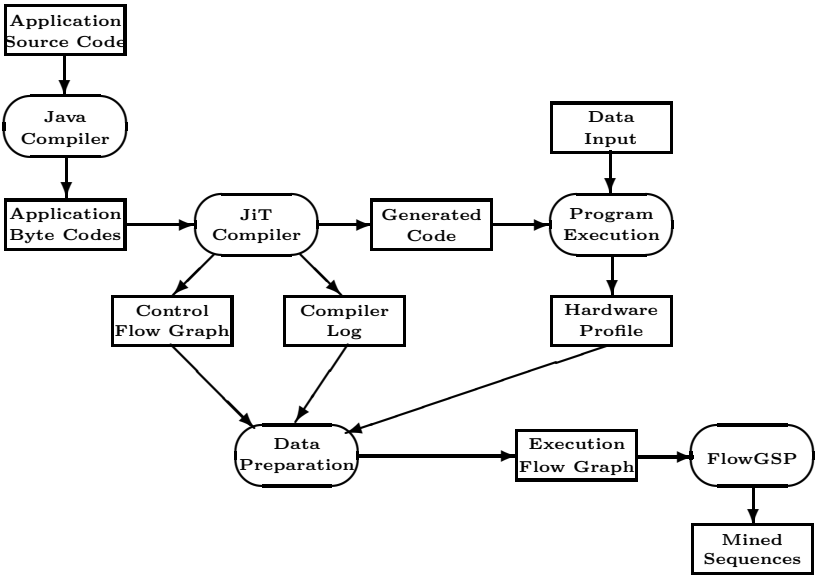

Fig. 2. Overall architecture and flow in system that uses FlowGSP for mining

is added to the control-flow-graph representation of the program created by the compiler to produce the input for the mining tool. The Testarossa compiler comes equipped with a rich set of logging features, including the ability to report all generated machine instructions. The only modification to the compiler was to annotate each instruction with a corresponding basic block so that the log can then be transformed into an EFG. In the implementation of the mining tool, the hardware performance counter information and the control-flow-graph data from the compiler are stored in IBM DB2® Version 9.1 Express Edition for Linux, a relational database. A relational database was chosen because the amount of input data is quite large (some applications running in the WebSphere Application Server contain over 4000 methods). A flat representation of this data could result in a very large input file with very poor random-access performance. Moreover, a relational database allows concurrent access to the data, which enables the use of a parallel implementation of FlowGSP.

For the use of the mining tool reported in this paper, each vertex in the EFG represents an instruction. The weight of each instruction represents the amount of total execution time spent on that instruction. The System Z operating system uses an event-based sampling mechanism: active events and the instruction under execution are recorded when the sample takes place. Instructions that occupy more cycles will be sampled more frequently, and the number of sampling hits or "ticks" is recorded on each instruction. The vertex weights are calculated by counting the number of sampling ticks on each instruction. The edge frequencies in the EFG are a measure of how many times each edge was taken during program

execution. In the case of edges that lie between basic blocks, this value can be read directly from the control flow graph in the compiler logs. For intra-basic-block edges, edge weights are assigned the frequency of the basic block in which they reside. Both edge and basic block frequencies in the control flow graph are obtained by the compiler through counters inserted in the JVM interpreter.

Each vertex is assigned attributes based on the corresponding instruction's characteristics or events observed on the instruction in the hardware profile data. Examples of attributes include: opcode, whether an instruction-cache miss was observed, and whether the instruction caused a TLB miss.

In this application FlowGSP is mining for sequences of attributes that occur in subpaths of the EFG, but this search is based on edge frequency collected by the compiler. Precise path execution frequency cannot be derived from edge frequencies [2]. Therefore, the results produced by the mining tool are an approximation. The support reported for a sequence of attributes represents the maximal possible execution of that path that could have occurred based on the edge-frequency information available [13].

FlowGSP is a general flow mining algorithm that can be applied to any flow graph. For instance, each vertex of the EFG could represent any single-entry/single-exit region, including a Java bytecode, a basic block, or an entire method. The vertex weights and edge frequencies would have to be computed accordingly.

3.2 Operation of the Mining Algorithm

When the tool is run, it first recreates the control flow graph from the information taken from the compiler logs. Then, it inserts each instruction from the hardware profile into the correct basic block using the instruction's annotations. The tool constructs and mines only a single method at a time in order to match the level of granularity of the compiler; the Testarossa JIT compiles each individual method in isolation. As a consequence, FlowGSP does not discover patterns that cross method boundaries. However, this restriction is a design decision of the tool, not a limitation of the algorithm.

To mine graphs containing cycles, FlowGSP does not allow a vertex that is the start vertex of a current candidate sequence to start a new sequence. Therefore a vertex within a cycle can only start a sequence the first time that it is visited. FlowGSP can detect frequent subpaths that occur over cycles but avoids looping indefinitely because the lenght of a sequence is bounded by an specified constant. Jocksch provides a detailed description of FlowGSP [13].

FlowGSP is an iterative generate-and-test algorithm. Each iteration creates a set of candidate sequences from the survivors of the previous generation, and then calculates their supports and tests them against the provided thresholds (discussed in Section 3.3). Each iteration discovers longer sequences in the data. Execution terminates when either a specified number of iterations have completed or no new candidate sequences meet the minimum support thresholds.

3.3 Support Thresholds for Mining

FlowGSP accepts a number of parameters that can adjust the type and quantity of sequences that are discovered. FlowGSP takes a maximal support threshold and a differential support threshold. If the support of a sequence does not meet either of these thresholds, then the sequence is excluded from further mining. FlowGSP also accepts a maximum allowable gap size and window size. The maximum gap size determines how much space is allowed between each part of a sequence, and the maximum window size determines how many vertices to consider when searching for one part of a sequence.

Table 1 lists the parameters used in the experimental evaluation for both the SPECjvm2008 benchmarks and the DayTrader 2.0 benchmark in the WebSphere Application Server. The support values for the application server are lower than the corresponding values for the SPECjvm2008 benchmarks because the application server is orders of magnitude larger than any of the SPECjvm2008 benchmarks and has an extremely flat profile. The System z10 instructions are grouped into pairs for execution. Therefore, events that occur on one instruction of a pair can sometimes also appear on the other instruction. A window size of one is used to group paired instructions together so that more accurate patterns can be discovered.

Table 1. FlowGSP parameters used during this study

Parameter	crypto	compiler	sunflow	montecarlo	xml	serial	WebSphere
Maximal support	1%	7%	7%	7%	15%	7%	1%
Diff. support	1%	7%	7%	7%	15%	7%	1%
Gap size	1	0	0	0	0	0	0
Window size	1	0	1	1	1	1	1
Iterations	5	5	5	5	5	5	5

4 Opportunities Discovered

Before the development of the data-mining framework, significant development resources had been invested on the search for performance improvement opportunities in applications running in the WebSphere Application Server. This investment resulted in many observations about potential opportunities for performance improvement. Therefore, a first effort to test the FlowGSP algorithm, and to build confidence in the compiler development team about the efficacy of the framework, was a set of *acid tests* to find out if data mining could discover the opportunities for code improvement that were already known to the team. FlowGSP performed extremely well in these tests: it identified all the patterns that were listed by the developers. Examples of these patterns include:

1. A high correlation between data cache misses, TLB misses, and instruction cache misses. Consultation with hardware experts led to the observation that the page table is loaded through the instruction cache, which explained the

unusual correlation. After FlowGSP confirmed and quantified this correlation, large pages (1 MB instead of 4 KB) were used to reduce the number of TLB misses, resulting in a performance improvement of 3% on applications running in the WebSphere Application Server.
2. A high incidence for instruction-cache misses on entry to JIT code methods. These are cold cache misses for which effective prefetching is a challenge because of dynamic method dispatching. This observation led to additional efforts for inlining and code-cache organization by the compiler team, as well as to discussions on how to mitigate the cache misses in future hardware releases.
3. A high correlation between branch misprediction and instruction cache misses on indirect branches with a higher-than-expected occurrence of these events. A large volume of indirect branches overflows the branch-table buffers. The compiler team implemented code transformations to transform indirect branches into direct branches through versioning. Moreover, the hardware team was engaged to look for solutions to mitigate this issue in future hardware.

The discovery of these issues through manual inspection of performance-monitor data by analysts required orders of magnitude more time and effort than the analysis with the data-mining tool based on FlowGSP. Moreover, the manual approach is not easy to reproduce for a new data set and is less deterministic.

Once the development team was confident about the results produced by the mining tool, they started examining the output of the tool to find new opportunities for code improvement. The time spent in the EX instruction in array copies described in Section 2 is one such opportunity. The team discovered most of the new opportunities when applying the tool to profiling data collected from newer benchmarks, such as the SPECjvm2008. While extensive development effort has been dedicated to discover opportunities in applications running in the WebSphere Application Server over many years, these newer benchmarks have received relatively less attention from the compiler development team. Some of the new discoveries are listed here:

- Stores account for a majority of data cache directory misses [14] in all SPECjvm2008 benchmarks. This is unexpected because the load-to-store ratio in programs is typically on the order of 5:1. Moreover, intuition would indicate that a program writes to locations from which it has read recently. Discussions and analysis are still under way to better understand this ratio. The `serial` benchmark spends three times more time servicing directory lookups for stores than for loads. This benchmark is highly parallel in nature, which, on the surface, would lead developers to dismiss cache contention as a concern. The trends presented by FlowGPS, which would have remained unobserved under manual inspection, have been instrumental in forcing developers to reconsider cache contention as a possible concern.
- Address-generation interlock (AGI) accounts for more than 10% of the execution time in some benchmarks. In the System z architecture, an AGI occurs when the computation of the address required by a memory access

instruction has not completed by the time that the instruction needs it [22]. In some cases, such as in a small pointer-chasing loop, AGIs are difficult to avoid. The mining tool's finding is helping to focus analysis in this benchmark, and the team is planning a review of the instruction scheduling in the compiler to reduce the impact of AGIs on execution time.

– Branch misses account for 9% of execution time in `montecarlo`, a benchmark from the SPECjvm2008 suite. This is unexpected because the execution of this benchmark is dominated by a single method with several hot loops and the benchmark has very good instruction locality. This result led to further analysis that uncovered a limitation in the hardware's instruction fetch unit: the unit stops predicting branches when it cannot detect any previously taken branches within a given window further down the instruction stream. A consequence of this limitation is that when the compiler unrolls a loop, it needs to take into account the size of this window to ensure that the loop backedge is predicted correctly. The compiler team is currently re-examining the loop unrolling strategy to take into account the penalty for branch misses.

Experienced compiler developers will understand the value of the observations above to provide direction to a compiler development team. These observations focus on the z/architecture®, the Testarossa compiler, and are based on mining data from the SPECjvm2008 benchmark suite. A similar approach can be used to most combinations of compiler/architecture/application. Moreover, the mining tool can be used to discover opportunities that might be specific to important applications.

5 Experimental Data on the Usage of the Mining Tool

This section presents statistics on the usage of storage and on the time required to mine several benchmarks. The goal of this section is to provide developers with an idea of the resources needed to deploy such a tool, and to encourage researchers to come up with improvements on our tool design. Information reported here include size of input data, overall running time, number of sequences generated, and the format of the rules output by the tool.

5.1 Profiling and Storage Requirements

This experimental evaluation uses the DayTrader 2.0 benchmark in the WebSphere Application Server 7.0 and programs from the SPECjvm2008 benchmark suite. All programs are run using the IBM Testarossa JIT compiler. The WebSphere Application Server workload is DayTrader 2.0 and the server is run for 5 minutes once a stable throughput has been achieved. This delay is necessary to ensure that the Testarossa JIT has compiled the majority of the methods in the application server to native code. The throughput of the application server increases as methods are compiled to native code. Therefore, stabilization of throughput is an indication that the majority of the code being executed has

been natively compiled. A hardware profile of 5 minutes of execution of the WebSphere Application Server results in roughly 37 MB of compressed data. The same run produces a 5.9 GB uncompressed, plain-text compiler log.[2] At the time of this writing, the Testarossa JIT does not have an option to output logs in a compressed format. Compressing the compiler-generated log using gzip reduces its size to around 700 MB.

Table 2. SPECjvm2008 benchmarks studied

Benchmark	# of Methods to Account for 50% of time	# of Methods Compilations	# Unique Methods Invoked
`compiler.compiler`	60	3659	7113
`compiler.sunflow`	55	4009	6946
`crypto.signverify`	2	1219	4654
`scimark.montecarlo`	1	703	4077
`serial`	8	2967	7645
`xml.transform`	25	5374	12430

The SPECjvm2008 benchmarks are profiled for a period of 4 minutes after a 1-minute warm-up time. Only a minute is required until the most of the benchmark code is being executed natively because the SPECjvm2008 benchmarks used in this study are significantly smaller than applications running in the WebSphere Application Server. The 6 SPECjvm2008 benchmarks examined in this study are listed in Table 2. The data in this table provides an indication of how flat the execution profile of each benchmark is by listing the number of methods that need to be examined to account for 50% of the execution time.[3] The table also show the total number of method compilations and the total number of unique methods that are invoked when the benchmark is executed. These benchmarks were chosen because they form a representative sample of the SPECjvm2008 benchmark suite and they produce both flat and non-flat profiles. Running these benchmarks for 5 minutes results in 7 MB of hardware profiling data per benchmark on average, and an average uncompressed compiler log with 1.4 GB of data. The benchmark with largest hardware profile is `compiler.compiler` which produces 12 MB of data. largest compiler log has 3.3 GB of data and is produced by `xml.transform`. The benchmark `scimark.montecarlo` produces the smallest hardware profile (385 KB) and the smallest compiler log (97 MB).

5.2 Time Needed to Mine

The execution time of the tool depends on the size of the log of the program being mined and the parameters passed to the tool. FlowGSP is multi-threaded

[2] The compiler option required to output control flow graph data also outputs a large volume of information that was extraneous to the mining process.

[3] This measurement is an approximation because the number of sampling ticks in the performance monitor that is used to determine the number of methods shown in the table.

in order to exploit the resources available in multi-core architectures. FlowGSP was run with 8 threads on a machine equipped with two AMD 2350 quad-core CPUs and 8 GB of memory. All runs were performed with the parameters outlined in Section 3.

Table 3. Running times of FlowGSP, in seconds

Program	Execution Time
Websphere App. Server (DayTrader 2.0)	6399
`compiler.compiler`	815
`compiler.sunflow`	539
`scimark.montecarlo`	2
`xml.transform`	557
`serial`	215
`crypto.signverify`	177

Table 3 lists the running time of FlowGSP on both the DayTrader 2.0 benchmark in the WebSphere Application Server and SPECjvm2008 benchmark profiles with execution time in seconds. The `xml.transform`, `compiler.sunflow`, `serial`, and `scimark.montecarlo` benchmarks terminated when no more candidates with support greater than the minimum threshold remained. `Xml.transform` and `scimark.montecarlo` terminated after three iterations, `compiler.sunflow` and `serial` after four iterations. `Montecarlo` has one small method which occupies almost 100% of total execution time. Therefore the time to mine this benchmark is significantly lower. The times reported in Table 3 indicate that the mining tool based in FlowGSP can be used on a daily basis in the development of a production compiler.

5.3 Sequences Reported by Mining

FlowGSP outputs frequent sequences in the following format:

$$S = \langle s_1, \ldots, s_k \rangle$$

where each $s_i \in s_1, \ldots, s_k$ is a set of attributes:

$$s_i = (\alpha_1, \ldots, \alpha_k)$$

Each sequence is accompanied by four values, which indicate the sequence's weight, frequency, maximal, and differential support. In this use of the data-mining tool the vertices of the EFG are instructions. Examples of attributes include the instruction type, occurrence of cache misses, pipeline interlock, branch missprediction, the type of bytecode the originated the isntruction, *etc.* Results are output to a plain-text file. In the experiments reported here, the DayTrader 2.0 benchmark in WebSphere produced 1286 sequences while the SPECjvm2008 data produced, on average, 64,000 sequences. The SPECjvm2008 benchmarks exhibited a very wide range in terms of the number of sequences generated. The

most sequences were discovered in the `scimark.montecarlo` benchmark with roughly 291,000 sequences. On the other hand, the `xml.tranform` benchmark had the smallest number of sequences at around 1,900.

In general, support thresholds for the SPECjvm2008 benchmarks were set generously low because this is an initial exploration of the applications of data mining in the compiler development. These low thresholds ensure that no interesting sequences are overlooked. With experience the support threshold can be increased to allow only the most interesting sequences to be reported. It could be possible in future work to automate this process based on the number of surviving sequences.

We implemented an user interface to display the results of mining. This interface allows sequences to be sorted lexicographically or by any of the support metrics. A maximum and minimum support value can be specified to reduce the number of sequences displayed. The tool can also selectively display sequences based on whether they do or do not contain specific attributes. This filtering is particularly effective at reducing the number of sequences that must be examined by a compiler developer. For instance, the `serial` benchmark contained 16,518 sequences, but only 2,880 involved pipeline stalls due to AGI interlocks. Ranking these resulting sequences by maximal or differential support allows quick identification of the most interesting patterns.

The tool also allows the developer to specify one rule as the baseline against which all other sequences are compared. This feature allows for easy comparison of sequences with respect to the baseline sequence.

6 Related Work

This is potentially the first attempt to use data mining to discover patterns of execution that occur frequently in an application but yet do not necessarily occur inside loops. Work that is related to this approach include performance analysis tools, the use of performance counters in JVMs, and the search for code bloat.

Optiscope is an "optimization microscope" developed to aid compiler developers in understanding low-level differences in the code generated by a compiler executing different code transformations, or between code generated by two different compilers for the same program [15]. Optiscope automatically matches up code in two hardware profiles that originated from the same region of source code. Optiscope focuses on loops. In contrast, FlowGSP focuses on finding interesting patterns within a single hardware profile and aims to discover common patterns that occur throughout the profile.

The design of most existing performance analysis tools, such as the popular Intel VTune for Intel® chipsets [5], focuses on locating small regions of code that are frequently executed to concentrate development efforts on these regions. Chen *et al.* try to capture the most execution time with the least amount of code [4]. Similarly, Schneider *et al.* use hardware performance monitors to "direct the compiler to those parts of the program that deserve its attention" [17]. Contrary to earlier work, the premise of this paper is that in some applications these parts are scattered through the code and not concentrated in smaller

regions. Hundt presents HP Caliper, a framework for developing performance analysis tools on the Intel Itanium® platform running HP-UX [10]. Similar to the approach presented here, Caliper integrates sampled hardware performance counters with compiler-generated dynamic instrumentation. Dynamic instrumentation involves changing program instructions on the fly to obtain more accurate program analysis. However, unlike our mining tool, HP Caliper does not attempt to mine the combined data for patterns.

Huck *et al.* present PerfExplorer, a parallel performance analysis tool [9]. PerfExplorer incorporates a number of automated data analysis techniques such as k-means and hierarchical clustering, coefficient of correlation analysis, and comparative analysis. PerfExplorer targets application developers seeking to understand bottlenecks in their code, not compiler developers. Also, PerfExplorer does not search for frequent sequences in the data.

Cuthbertson *et al.* incorporate performance counter information into a production JVM to improve program performance [6]. They use a custom library to retrieve instruction cache miss information on the Intel Itanium platform. This information is used to guide both object allocation and instruction scheduling in order to increase performance. They achieve an average performance increase of 2% on various Java benchmarks. Schneider *et al.* perform similar work using hardware counters on the Intel Itanium platform to guide object co-allocation [17]. However, these approaches can only improve the performance of *existing* code transformations whereas FlowGSP is aimed at discovering opportunities for new code transformations. Also, both approaches only look at a small fraction of all available program data. It is not clear how much increased overhead will result from increasing the amount of data being brought into the compiler.

Buytaert *et al.* use hardware-performance counters to both improve the accuracy and decrease the cost of hot method detection in a production JVM [3]. Their focus is purely on improving the efficiency and accuracy of the JVM and does not provide any insights into new opportunities for code transformations.

Xu *et al.* develop a method for profiling Java programs to identify areas of code bloat [23]. They evaluate the DaCapo benchmark suite, elements of the Java 1.5 standard library, and Eclipse 3.1, and are able to identify a number of specific opportunities to improve performance by decreasing bloat. Similarly, Novark *et al.* develop a tool called Hound to identify memory leaks and sources of bloat in C and C++ programs [16]. Hound was able to achieve a 14% performance increase in one of the studied benchmarks by identifying a single line of code that needed to be changed. While removing code bloat can significantly improve the performance of applications, it only addresses performance from the point of view of the application programmer. Proper use of code transformations by the compiler is equally as important in increasing program performance.

7 Conclusion

In compiler and computer-architecture development, as in Science in general, discovering the question to ask is often as difficult as finding the answer. Recent

developments in hardware performance-monitoring tools, and in leaner techniques to insert profiling counters in generated code, have provided developers with an unprecedented amount of data to examine the run-time behavior of a program. The combination of these techniques amounts to a very powerful scope. The mining tool presented in this paper is a mechanism to help focus this powerful scope on patterns that happen frequently enough to warrant the attention of compiler or hardware developers. This paper describes the methodology and the tool used for this mining task. It also presents several examples of discoveries that were done using the tool. Then, it presents statistics on the amount of space and time that is required to use the tool to mine the data produced by enterprise software in a high-end hardware platform with a mature compiler infrastructure. This data indicates that this methodology can be used routinely for the development of production compilers.

Acknowledgments

We are very thankful to Jane Bartik and John Rankin from the IBM Poughkeepsie campus for sharing their invaluable insight into the z/Architecture. This work was supported by an IBM Centre for Advanced Studies fellowship and by grants from the Natural Science and Engineering Research Council (NSERC) of Canada through its Collaborative Research and Development program.

Trademarks

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: International Conference on Data Engineering (ICDE), March 1995, pp. 3–14 (1995)
2. Ball, T., Mataga, P., Sagiv, M.: Edge profiling versus path profiling: the showdown. In: Symposium on Principles of Programming Languages (POPL), San Diego, CA, USA, pp. 134–148 (1998)
3. Buytaert, D., Georges, A., Hind, M., Arnold, M., Eeckhout, L., De Bosschere, K.: Using HPM-sampling to drive dynamic compilation. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Quebec, Canada, pp. 553–568 (2007)
4. Chen, H., Hsu, W.-C., Lu, J., Yew, P.-C., Chen, D.-Y.: Dynamic trace selection using performance monitoring hardware sampling. In: Code Generation and Optimization (CGO), San Francisco, CA, USA, pp. 79–90 (2003)
5. Intel Corporation. Intel v-Tune performance analyzer, `http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-white-papers/`

6. Cuthbertson, J., Viswanathan, S., Bobrovsky, K., Astapchuk, A., Kaczmarek, E., Srinivasan, U.: A practical approach to hardware performance monitoring based dynamic optimizations in a production JVM. In: Code Generation and Optimization (CGO), Seattle, WA, USA, pp. 190–199 (2009)
7. Geronimo, A.: Apache daytrader benchmark sample (October 2009), `http://cwiki.apache.org/GMOxDOC20/daytrader.html`
8. Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., Sundaresan, V.: Java just-in-time compiler and virtual machine improvements for server and middleware applications. In: Conference on Virtual Machine Research and Technology Symposium (VM), San Jose, CA, USA, pp. 12–12 (2004)
9. Huck, K.A., Malony, A.D.: PerfExplorer: A performance data mining framework for large-scale parallel computing. In: ACM/IEEE Conference on Supercomputing (SC), Seattle, WA, USA, p. 41 (2005)
10. Hundt, R.: HP Caliper: A framework for performance analysis tools. IEEE Concurrency 8(4), 64–71 (2000)
11. IBM Corporation. WebSphere Application Server (October 2009), `http://www-01.ibm.com/software/websphere/`
12. Jackson, K.M., Wisniewski, M.A., Schmidt, D., Hild, U., Heisig, S., Yeh, P.C., Gellerich, W.: Ibm system z10 performance improvements with software and hardware synergy. IBM J. of Res. and Development 53(1), Paper 16:1–8 (2009)
13. Jocksch, A.: Data mining flow graphs in a dynamic compiler. Master's thesis, University of Alberta, Edmonton, AB, Canada (October 2009)
14. Mak, P., Walters, C.R., Strait, G.E.: IBM system z10 processor cache subsystem microarchitecture. IBM J. of Res. and Development 53(1), Paper 2:1–12 (2009)
15. Moseley, T., Grunwald, D., Peri, R.V.: Optiscope: Performance accountability for optimizing compilers. In: Code Generation and Optimization (CGO), Seattle, WA, USA (2009)
16. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, pp. 397–407 (2009)
17. Schneider, F.T., Payer, M., Gross, T.R.: Online optimizations driven by hardware performance monitoring. In: Conference on Programming Language Design and Implementation (PLDI), pp. 373–382 (2007)
18. Shiv, K., Chow, K., Wang, Y., Petrochenko, D.: SPECjvm2008 performance characterization. In: SPEC Workshop on Computer Performance Evaluation and Benchmarking, Austin, TX, USA, pp. 17–35 (2009)
19. Shum, C.-L.K., Busaba, F., Dao-Trong, S., Gerwig, G., Jacobi, C., Koehler, T., Pfeffer, E., Prasky, B.R., Rell, J.G., Tsai, A.: Design and microarchitecture of the IBM system z10 microprocessor. IBM J. of Res. and Development 53(1), Paper 1:1–12 (2009)
20. Standard Performance Evaluation Corporation. SPEC: The standard performance evaluation corporation, `http://www.spec.org/`
21. Sundaresan, V., Maier, D., Ramarao, P., Stoodley, M.: Experiences with multithreading and dynamic class loading in a java just-in-time compiler. In: Code Generation and Optimization (CGO), New York, NY, USA, pp. 87–97 (2006)
22. Webb, C.F.: IBM z10: The next generation mainframe microprocessor. IEEE Micro 28(2), 19–29 (2008)
23. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: profiling copies to find runtime bloat. In: Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, pp. 419–430 (2009)