Search strategies for Java bottleneck location by dynamic instrumentation

D.J. Brear, T. Weise, T. Wiffen, K.C. Yeung, S.A.M. Bennett and P.H.J. Kelly

Abstract: The authors have developed a prototype tool that supports instrumentation of distributed Java applications by on-the-fly deployment of interposition code at user-selectable program points. The paper explores the idea, originated in the Paradyn Performance Consultant, of systematically searching for performance bottlenecks by progressive refinement. They present the callgraph search algorithm in detail, and discuss a number of shortcomings with the approach, some of which can be addressed by improving the search strategy. They support their conclusions with two application examples. This is a report of work in progress, aimed at stimulating further investigation of this interesting approach.

1 Introduction

The idea of dynamically patching instrumentation code into a program while it is running immediately leads to the idea of deploying instrumentation in response to earlier measurements. For diagnosing performance problems, the objective in doing this is to direct the programmer towards opportunities for improving performance. The idea was pioneered in the Paradyn Performance Consultant [1], which uses a callgraph-based bottleneck search strategy [2].

This paper presents our progress in exploring this approach in the Java context. We address the following issues:

- *Implementing dynamic instrumentation within a Java Virtual Machine:* Several options are available; our implementation uses our Veneer Virtual JVM, which offers the prospect of automatically optimising the bottleneck code once it has been identified.
- *The callgraph search strategy:* We give a detailed exposition of the callgraph search strategy, which clarifies some of the difficulties of the approach.
- *Search strategy enhancements*: To address some of these difficulties, 'deep start' enhancements to the callgraph strategy have been proposed [3]. Random call-stack sampling has been shown to be effective but for Java it is difficult to implement; we study a call count alternative.

We demonstrate the usefulness of our prototype tools, JUDI (Java utility for dynamic instrumentation) and JBOLT (Java bottleneck locator toolkit) using two nontrivial applications.

2 Background

The Performance Consultant (PC), in Paradyn [1], structures the search for performance 'bottle-necks' in terms of experiments; each experiment tests a hypothesis. A hypothesis is an assertion that the application spends a substantial amount of time behaving in some pathological way which might be addressed by the performance engineer. Each experiment is targeted on a 'focus'. For each experiment, the PC uses dynamic instrumentation to collect data to evaluate whether the experiment's hypothesis holds true for this focus.

If an experiment results in support for the hypothesis at the specified focus, a further experiment can be formulated to identify the problem more precisely. The experiment can be refined in two ways–by refining the hypothesis (for example, by distinguishing different kinds of synchronisation problem), or by refining the focus. A natural focus refinement is to test whether the problem is within a function or within one of its callees. Other focus refinements might be to distinguish particular threads or transaction types.

An alternative to using dynamic instrumentation might be to instrument the binary to generate data on all possible hypotheses at all foci. The PC aims to get essentially the same result, with much lower overhead. The approach relies on sampling, and works on long-running applications provided that behaviour is statistically fairly stable (we return to this issue shortly). The PC monitors the interference caused by the instrumentation it inserts, and throttles the number of ongoing experiments to keep the interference within specified limits. Refining the focus by tracing the application's callgraph confines the search to code which is actually executed (explored in [2]).

For finding CPU bottlenecks, this approach is not clearly better than conventional approaches, such as sampling the program counter at random intervals. The real potential for the idea lies in bottlenecks which are harder to characterise–where instrumenting for all hypotheses would lead to excessive interference.

DynInst [4] is an example of a portable library for dynamic instrumentation, derived from the dynamic instrumentation technology [5] developed as part of Paradyn. In DynInst, a 'point' is a location in a program where instrumentation can be inserted. A 'snippet' is an abstract syntax tree representing some executable code which is to be inserted into a program at a point. Snippets can include conditionals, functions calls and loops. Snippets are translated to binary instructions then copied into an array

IEE Proceedings online no. 20030807

doi: 10.1049/ip-sen:20030807

Paper received 25th April 2003

The authors are with the Department of Computing, Imperial College, London SW7 2AZ, UK

in the application's address space. The application then has to be modified to branch into the snippet code, using code they term a 'trampoline'.

Dynamic instrumentation for Java cannot be implemented this way, without exposing low-level implementation details of the JVM (for example, profile-directed re-optimisation). There are several alternative approaches:

- *Re-define the class* using the Java Debug Interface (JDI) call `VirtualMachine.redefine Classes ()`, introduced in Sun's JDK 1.4. This approach is used in ProbeMeister [6]. The overhead to do this is reported to be around 20 ms for a small example, but increases with large classes since methods cannot be redefined individually, and JIT optimisation must be re-done. To reduce the overheads, Dmitriev [7] advocates refining the JDI with a call to redefine methods individually.
- *Run the JVM in debugging mode*, and set breakpoints to insert instrumentation. This is the approach taken by Popovici *et al.* [8]. Historically, JVMs have run substantially slower than normal in debugging mode, whether breakpoints are present or not; modern JVMs appear to make this approach more competitive.
- *Run the Java application in a virtual JVM*. This is the approach used in our JUDI tool [9]. We use the native JVM to execute application bytecode as much as possible, but have to intercept execution to retain control. The scheme suffers some overhead (see Section 5) on execution of all the application's code (apart from system libraries), but runs with JIT optimisation.

2.1 The Veneer Virtual Java Virtual Machine (vJVM)

When extending the Java platform, it is often desirable to implement new features directly into the Java Virtual Machine. However, such modifications are tied to a specific JVM, and the complexity and size of many JVM implementations can make this a difficult and time-consuming task. One way to overcome the portability problem might be to write a JVM in Java, which runs on top of an underlying JVM. This has been done before in projects such as JavaInJava [10]. However, such implementations have a tendency to be extremely slow, since they attempt to emulate all aspects of a JVM.

Our approach, similar to the Dynamo/Rio projects [11, 12], is to build a Java Virtual Machine using Java that uses the underlying JVM to directly execute as much of the program code as possible, only seizing control of the system when we wish the behaviour to deviate from that of the underlying JVM [13]. It allows us to run most of the application code directly (i.e. jumps to the corresponding bytecode), but the vJVM maintains control over execution by intercepting control flow. Veneer is much more powerful than is needed for dynamic instrumentation alone – our long-term goal is to use it to diagnose performance improvement opportunities automatically, then optimise dynamically.

The control flow is intercepted by 'fragmenting' each method. There are a number of different fragmentation policies: by basic block, at method level (used by the JBolt extension to JUDI, see Section 3) and at RMI invocations (used for our work on RMI optimization [13]). The method body is split into blocks, and the method entry is replaced by an 'executor loop' that walks the control flow graph, invoking each block in turn. A method's control flow graph can be updated 'on-the-fly' (i.e. as the application is running), allowing us to use this as a framework for dynamic instrumentation.

The fragmentation process (which is based on the SOOT framework [14]) includes use/def and liveness analyses. Each fragment carries this as dependence metadata, which can be used in a run-time optimiser.

2.2 Java utility for dynamic instrumentation (JUDI)

JUDI is a prototype dynamic instrumentation tool for Java [9]. It has a client graphical user interface (GUI) which connects to a set of remote vJVMs running fragmented code. The GUI allows the user to browse the remote systems' methods, and to upload 'instruments' to the remote systems, where they are patched into the running code. The instruments are simple Java objects that can be compiled and loaded on-the-fly. The tool consists of two components, which can run on separate hosts to avoid interference:

- JUDI-StartApp is responsible for starting the application and registering it with the RMI registry.
- JUDI-GUI allows the user to insert instruments dynamically into the application.

The instrumentation strategy used for the CPU Time bottleneck search (see Section 3) is designed to produce an inclusive timing of instrumented methods. Instruments are placed at the method entry point and at every return statement in the method. The instruments at the return statements are twinned with the single instrument at the entry, and when executed obtain the elapsed time on the HP Timer [Note 1] of the entry point instrument.

JUDI's unit of instrumentation deployment is an 'instrumentation strategy component' (ISC). This consists of:

- a set of *instruments* – subclasses of a generic instrument plan block. Instruments typically start, stop and log timers, or generate a log entry recording control flow, or data values.
- an *instrumentation strategy*: This is usually just whether the instrument is to be executed 'before', 'after' or 'around' its target block, and whether it applies to the whole method, or every basic block in the method.
- *instrumentation targets*: This is the set of program objects (methods, classes) to which the instrumentation strategy should be applied. If not the entire program, this is selected explicitly through the GUI.
- *instrumentation data class*: Instruments generate data, usually either a log or some kind of histogram.
- *instrumentation analyser*: This is a GUI component for viewing the results from the experiment.

3 The basic bottleneck search algorithm

Automatic bottleneck search is implemented as a JUDI ISC called JBolt, the Java bottleneck locator toolkit. Fig. 1 shows, in outline form, the automatic bottleneck search algorithm (described informally in [2]). The algorithm starts with an application ready to run:

- It installs instrumentation at the root of the call graph (line 10), then allows execution to proceed.

Note 1: The timer class used to obtain inclusive time measurements, *HPTimer*, was developed by Kwok Yeung [13]. It offers nanosecond resolution.

```
 1  // Initially no bottlenecks, no frontier, & focus is just application's Main method
 2  Set<MethodId> bottleneckSet, focus; focus.add(mainId);
 3  Set<Instrument> frontier;
 4
 5  // Frontier tells us which methods have been instrumented and maintains a list
 6  // of each method's instrument objects.  Initialise frontier to contain just
 7  // the methods called by Main, and add instrumentation to this initial frontier
 8
 9  Set<MethodId> mainCallees = mainId.calleeSet();
10  frontier.add(TimerInstrumentFactory(mainCallees));
11
12  // Now run application. As it runs it will encounter instruments we put in place.
13  app.start();
14
15  while (!app.finished()) {
16    // wait for application to execute an instrument; resulting callback
17    // enqueues instrument object that has been activated
18
19    Instrument m = app.getNextActivatedInstrument();
20
21    // Consult profile database to determine whether we have enough
22    // information to conclude that this method is a bottleneck
23
24    switch (profileDatabase.isBottleneck(m)) {
25    case YES:
26      // This method turns out to be a bottleneck.  Add this method to list of
27      // known bottlenecks, remove it from search frontier, remove its instrument-
28      // ation, & instead add its callees to the search frontier and instrument them
29
30      bottleneckSet.add(m.methodId, m.measurement);
31      frontier.remove(m);
32      focus.add(m.methodId);
33      frontier.add(TimerInstrumentFactory(m.methodId.calleeSet()));
34      break;
35
36    case NO:
37      // This method turns out not to be a bottleneck.  Remove it from
38      // search frontier, remove its instrumentation.
39
40      frontier.remove(m);
41      break;
42
43    case MAYBE:
44      break;    // leave instrumentation as it is for a while
45    }
46    // Update profile database for future reference
47    profileDatabase.update(m.methodId, m.measurement);
48
49    // Remove parent method from focus if none of its callees remains in the frontier
50    if (frontier.isIn(methodId.parent().calleeSet())) focus.remove(parent);
51  }
```

Fig. 1 *Pseudocode outline of callgraph-based bottleneck search algorithm*

- When an application thread executes a timer instrument, the application thread blocks and the active instrument object is passed to the search algorithm (line 19).
- The algorithm maintains a profile database which records instrumentation data accumulated so far. At line 24 the algorithm determines whether the new measurement allows us to classify this candidate program point as a bottleneck.
- When a bottleneck method is identified (line 25), we add it to the output set (line 30), then the instrumentation is refined to determine which, if any, of the method's callees is responsible (line 33).
- We may instead conclude that we have enough data to decide that this method is not a bottleneck (line 36). The instrument is therefore removed.
- Finally, we may decide to leave the instrument in place to accumulate further profile data.

The output consists of a list of bottleneck methods, prioritised by severity and specificity. This is conveniently presented to the user by colouring the nodes of the call graph, as shown in Fig. 3.

The algorithm in Fig. 1 is incomplete in two important ways:

- It fails to find some bottlenecks - where a bottleneck method is called from many different points, but none of its callers is itself a bottleneck (for example, see Fig. 2). We return to this issue in Section 4.
- The algorithm assumes that we know each method's callees.

3.1 Finding the callee set

The algorithm maintains two key data structures:

- *Focus*: This is the set of method which have been identified as bottlenecks, and whose callees are being instrumented to determine whether a refined bottleneck hypothesis holds – i.e. whether the problem lies in one of the callees. Focus methods are not instrumented.
- *Frontier*: This is the set of methods currently being instrumented. A method is in the frontier if its caller is in the focus (we ignore recursion for simplicity of presentation).

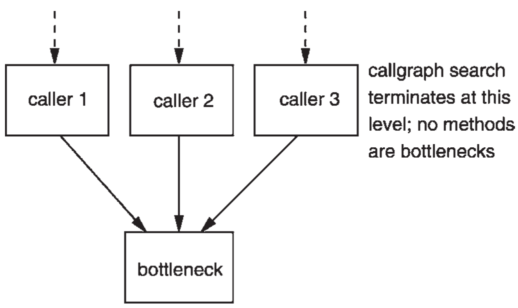

Fig. 2 *Bottleneck hidden from the callgraph search*

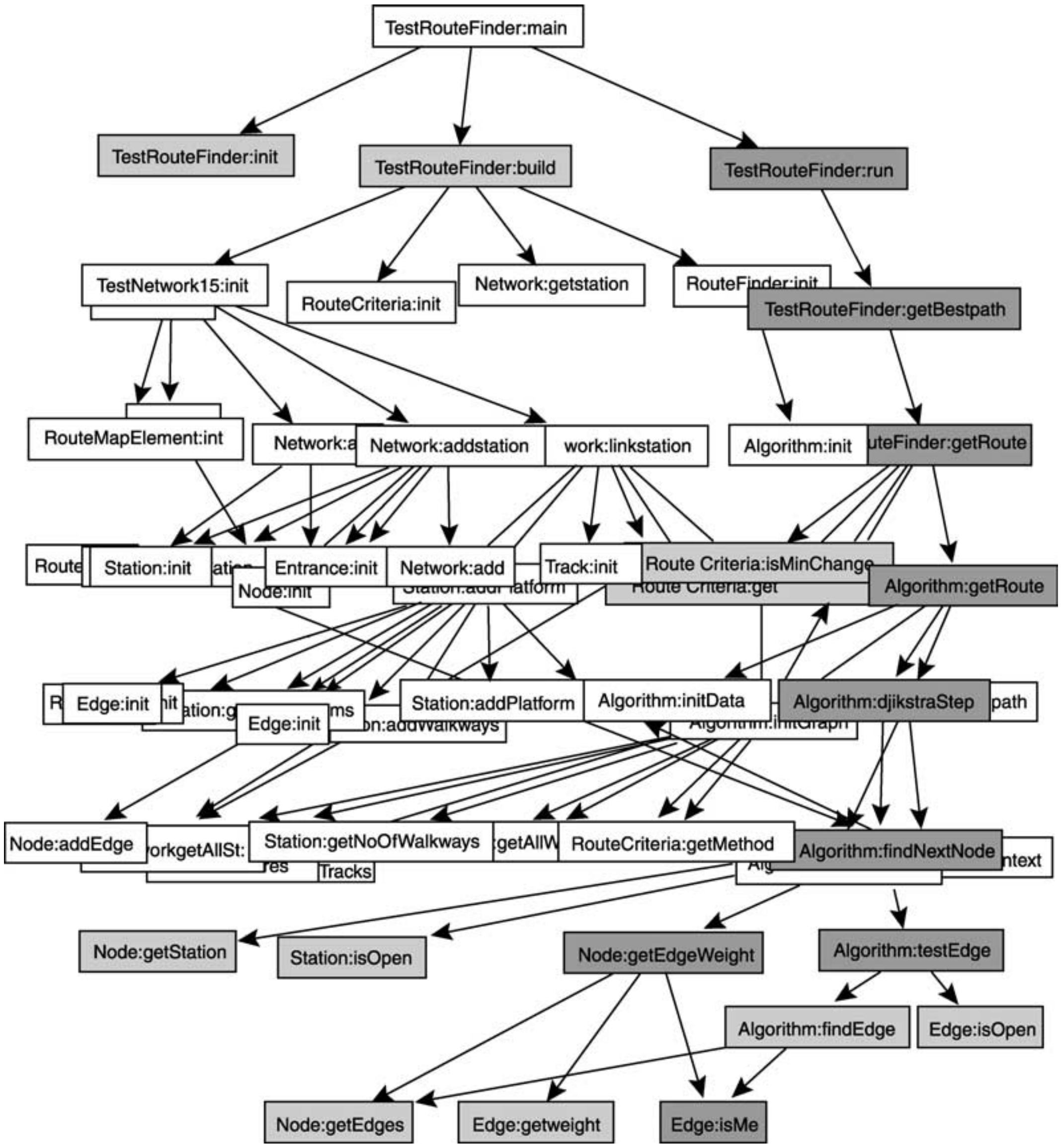

Fig. 3 *RouteFinder–bottlenecks identified by JBolt*
The path down the right-hand side (displayed in red on the JUDI GUI) identifies the method Edge:isMe as the main bottleneck in this application

The problem is that this definition is retrospective: by the time we find out that a method is a bottleneck, it is too late to instrument its callees. This is a fundamental problem, but there are several measures which help:

1. We could analyse each method to find call sites where the target method is statically known. This technique is used in Paradyn, but does not handle Java's prevalent virtual methods well. Callee methods identified this way can, in principle, be added to the frontier as soon as the caller reaches the bottleneck threshold. A method can be included in the callee set but not actually called, at the cost of some redundant instrumentation.
2. We could instrument each bottleneck method, with code to record its callees and add them to the frontier. This is how Paradyn deals with virtual and indirect method calls. This does lead to some interference: the callee logging overheads are included in the caller method measurement.
3. We add a simple instrument to *all* methods, which checks whether the caller is in the focus. If so, the callee is added to the frontier. Our current prototype uses this technique, but we plan to investigate alternatives.
4. Some bottleneck methods are executed only once in a run of the application. To instrument their callees, we have to re-run the application. The need for this is alleviated to some extent by augmenting the callgraph search as described in Section 4.
5. We could instrument the application to construct its call graph, then re-run the application and use this call graph to guide bottleneck search. The first run would be slowed down due to heavy instrumentation, but the second run would suffer minimum interference. This scheme relies on the call graph being very similar on both runs. This is an attractive alternative strategy for our prototype.

3.2 Bottleneck identification criterion

The objective is to find and prioritise bottlenecks. The criterion for including a candidate bottleneck would ideally be the proportion of the application's total run-time attributable to that candidate. However, to direct the bottleneck search, we need to classify candidates before the program has finished. For the experiments reported here we used a simple threshold of 10% of the execution time so far. We found that this strategy led to some problems:

- Object constructors often evaluate as bottlenecks when the application is starting up. When a constructor is called from the 'main' method of the application, it is instrumented since the 'main' method is initially in the search focus. Later on, when seen in the context of the entire program run, it will probably represent only a small fraction of the total CPU time.

- Small methods that are called very frequently often do not initially appear to be bottlenecks. When instrumented and a time obtained, the method's relatively short execution time does not make it appear to be a bottleneck. However, later on in the program, once that method has been called many times, the combination of high frequency calls and short but non-negligible execution time may mean it is a bottleneck.

In either case, the root cause of the initial misinterpretation of the metric can be attributed to a lack of context; the method can really only be properly evaluated as a bottleneck in the context of the whole program. For this reason, JBolt periodically re-evaluates (currently at the end of each run through the application) all instrument data, in order to get a balanced view for each method.

4 Searching upwards through the call graph

Figure 2 shows how the callgraph-based bottleneck search algorithm fails to find some bottlenecks. The callees of a method are included in the search only if the caller's execution time indicates that a bottleneck is present. If a method is called by several non-bottleneck methods, it could still account for a large proportion of the run time.

The idea proposed by Roth and Miller [3] is to augment the search using additional information, and use this to target the search on 'deep starters'. Roth and Miller choose deep starters, using call stack sampling, from information which their implementation already collects. In our vJVM implementation, it is possible to capture stack samples but rather expensive. Instead we use a simple instrument to count method executions (the same instrument builds the call graph in order to provide callee information). In our implementation, deep starters are methods whose execution frequency exceeds our chosen threshold (10%).

We use a deep-starter to target the callgraph search, by finding all the paths through the call graph that connect a focus node to the deep starter. All methods on these paths can be added to the frontier, and thus be instrumented. When executed, these instruments generate method timings (collected at line 19 in Fig. 1). If a method exceeds the bottleneck threshold, it is added to the frontier.

The timing instrument is an 'around advice' [15]: the timer is started on entry, and logged on exit. However, the deep-starter scheme above adds methods from the call paths to the instrumentation frontier before those methods have returned. If we use only 'around' instruments, we will not get any measurements until these methods are re-entered. Methods which are called just once will have to wait until the application is restarted. To improve this situation we experimented with 'late instruments' - if a method is already on the call stack, we add an 'after' instrument. This is used to measure the time between adding the instrument and method exit. The actual method execution time is sure to be more, but if this lower bound exceeds the threshold we can add the method to the focus immediately. We found that late instruments speed up the search substantially, at the expense of less reliable quantitative results (see Section 5.3).

5 Experimental evaluation

This Section presents two examples of using JBolt to detect performance bottlenecks, and validates the results against Sun's *hprof* profiler. RouteFinder is a railway route planning tool based on Dijkstra's shortest path algorithm. It is single-threaded and consists 3823 lines of code (55 classes, 74 methods). SpecJVM98 _208_db (data management) is taken from the SpecJVM98 benchmark suite [16]. The program performs a variety of database operations on a memory-resident database of name, address and phone number records. It is also single-threaded, and consists of 8541 lines of code (24 classes, 40 methods).

We used the Sun Java 2 platform, standard edition version 1.4_02, running on SuSE Linux 7.2. Most of the experiments were carried out on a system with a single 1400 MHz AMD Athlon processor, with separate 64KB L1 data and instruction caches, unified 256KB L2 cache and 512MB memory. *Hprof* [17] samples at a constant rate, so the results are more accurate on a slower system. To compare measured method timings we used a slower machine, with a 450 MHz Pentium III processor, with separate L1 data and instruction caches of 16kbytes, L2 unified cache 512kbytes and 256Mbytes memory.

Table 1 shows the impact of JBolt and *hprof* on the two applications' execution time. For _209_db, the slowdown is fairly small, and JBolt does better than *hprof*. However, for RouteFinder, the slowdown is very severe with both profilers, with JBolt somewhat worse. We believe the reason is that RouteFinder's bottleneck method is executed many times (7 million), while _209_db spends most of its time in a method which is called a small number of times. In both profilers, method entry is the main source of overhead, but *hprof* incurs performance overheads on primitive Java classes, which Veneer runs at full speed.

5.1 Routefinder results

Figure 3 shows the view displayed at the end of JBolt's search for bottlenecks in RouteFinder. There is only one search strand in the application, branching at *Algorithm.-findNextNode*, and joining again at *Edge.isMe*. Both *hprof* and JBolt agree on the bottleneck. As a result, the method *Edge.isMe* was modified. Running the new version of the program (with a larger rail network) gave a time of 12.07 s, as opposed to 25.13 s before optimisation (each averaged over five runs), i.e. a speedup of just over two.

Figure 4 illustrates the efficiency of the hybrid search (top-down + deep-start strategies) in comparison with the

Table 1: Benchmark overheads - the JBOLT runs were done with hybrid search (callgraph plus deep start), with no limit on the number of deep start instruments

	Route Finder		SpecJVM98_209_db	
	Time, s	Slowdown factor	Time, s	Slowdown factor
Unfragmented application	3.73	1.00	22.02	1.00
Fragmented application	28.13	7.54	24.30	1.10
Profiled with JBolt	105.14	28.19	26.96	1.22
Profiled with *hprof*	71.74	19.23	51.30	2.33

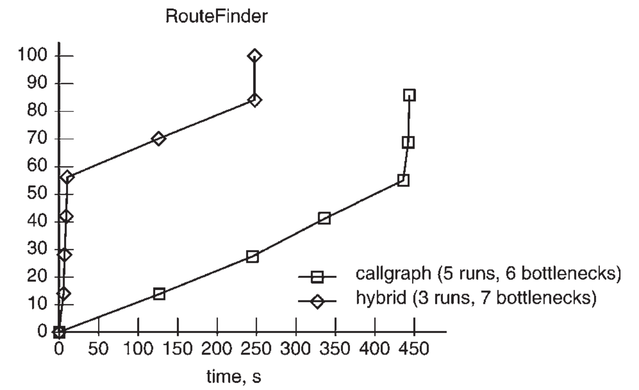

Fig. 4 *RouteFinder–the hybrid search finds more bottlenecks in approximately 55% of the time taken by the callgraph-only search*

These hybrid runs were done with a limit of one deep starter instrumented at a time

callgraph (just top-down) search. The hybrid search locates all the bottlenecks (including a hidden bottleneck) in ~55% of the time taken by the callgraph search on its own.

5.2 SpecJVM98 _209_db results

Again, *hprof* and JBolt agree on the main bottleneck for this application; the method *DataBase.shell_sort*. JBolt overheads are low because *DataBase.shell_sort* is called a small number of times.

For the *DataBase.read_db* method, JBolt only attributes 4.43% of total CPU time on average to this method, compared to *hprofs* 9.8%. Although JBolt and *hprof* agree on where the bottlenecks are in both applications, their attribution of time to methods varies substantially. In general, compared to *hprof*, JBolt underestimates time spent in short-running methods and overestimates time spent in longer-running methods.

5.3 Discussion

Our experiments show that JBolt is able to identify the same CPU bottlenecks as a conventional profiler. However, applications can run very slowly under JBolt. This is largely the overhead of fragmentation. We are implementing a number of improvements to Veneer's basic mechanisms. Another strategy which could help would be to switch between unfragmented and fragmented method variants as instrumentation is added and removed.

JBolt profiles the fragmented version of the application, not the real application. Comparing the method timings given by JBolt and *hprof* suggests that there are differences, so it is unclear how much of the variation in method timings and bottleneck ranking are due to differences in the performance characteristics of fragmented code, and how much is due to inaccuracies in the data sampling and analysis. Fragmentation appears to distort results for short lived methods.

Short-running applications like the benchmarks chosen here have to be re-executed several times (up to four) for JBolt to finish its search for bottlenecks. The main reason for this is to account for methods which are called only once per run.

The callgraph strategy has the desirable property that instrumentation overheads are never included in measurements. However, the upward search schemes introduce 'late instruments' which are added while measurements higher in the callgraph are in place. We heuristically alleviated this problem by allowing only one late instrument to be in place at a time; we maintain a late instrument queue, and always select the deepest late instrument in the call graph. Relaxing this constraint leads to serious measurement errors, but enormously faster convergence: most applications saw all the bottlenecks identified within the first run. A more sophisticated instrument placement strategy should help here.

6 Conclusions and further work

We have reported on our exploration of search strategies for using dynamic instrumentation to locate and characterise performance bottlenecks. This work forms part of our longer-term objective to explore automatic profile-driven optimisation, and it is constructed on top of the Veneer framework which we built for this purpose.

Our results are not entirely positive. The main purpose of dynamic instrumentation is to avoid the performance impact of static approaches. We expect to reduce the performance

- **Excessive CPU time**
  - ○ Due to **excessive use of String libraries**
  - ○ Due to **excessive garbage collection**
  - ○ Due to **thread contention**
  - ○ Due to **synchronised methods**
    - ○ Due to **lock contention**
  - ○ Due to **poor CPI**
    - ○ Due to **poor branch prediction**
    - ○ Due to **high cache miss rate**
      - ○ Due to **aliasing/associativity conflicts**
      - ○ Due to **excessive working set**
      - ○ Due to **poor spatial locality**
  - ○ Due to **JDBC**
    - * Due to **Statement processing overheads**
    - * Due to **SELECT metadata overheads**
    - * Due to **locking conflicts**
    - * Due to **large query results**

Fig. 5 *An example 'Why' bottleneck hierarchy*

Poor spatial locality implies cache misses, which imply CPU time. Conversely, if synchronised methods are not a bottleneck, then neither is lock contention. Contention for structured resources also leads to a hierarchy of bottlenecks. A similar process of refinement applies to source code (file, module, class, method), location (server, process, CPU, thread, transaction) and time of day impact of Veneer dramatically with further development. There are also serious concerns about the statistical significance of JBolt's results. This seems inherent in the approach: sampling is driven by earlier measurements, so is not very random. For very long-running applications, or applications with a known repetitive structure, this can, perhaps, be overcome.

We are currently testing a new version of JUDI (DYJIT), which runs with either the Veneer vJVM or the JDI (Java Debug Interface), and uses Aspect-**J** syntax [15] to specify instruments. We plan to do comparative benchmarking of the bottleneck instrument on the two underlying platforms. In addition, we are investigating amending Veneer to only use fragmented methods when they are instrumented (i.e. to run the original byte code where possible).

Perhaps the most promising prospect lies in searching for more subtle performance bottlenecks. Miller *et al.* [1] observe that the search for a bottleneck can involve refinement in three dimensions; they call this the W^3 model:

- *When*: Is the performance problem confined to a particular phase of the computation? A particular time of day?
- *Where*: At what class, method, module, server, component or line of code does the problem occur? The callgraph (for example, as shown in Fig. 2) shows a natural example. Others are possible: Which threads? Which users? Which transaction types?
- *Why*: What is the reason for the performance problem? Bottlenecks naturally form a hierarchical structure, as shown in Fig. 5.

In each dimension (when, where, why), the hierarchy provides a way to structure the search, leading to a successively more refined characterisation of the problem. This should allow us to target expensive instrumentation on just the parts of the code and the phases of the computation where subtle performance problems are likely to occur.

7 Acknowledgments

This work was funded in part by the UK Engineering and Physical Sciences Research Council through a PhD studentship and the DESORMI project (GR/R15566).

8 References

1 Miller, B.P., Callaghan, M.D., Cargille, J.M., *et al.* 'The Paradyn parallel performance measurement tool', *Computer*, 1995, **28**, (11), pp. 37–46
2 Cain, H., Wylie, B., and Miller, B.P.: 'A callgraph based search strategy for automated performance diagnosis', in Arndt Bode *et al.*, (Ed.): '*Euro-Par 2000 - Munich*', *Lect. Notes Comput. Sci.*, 2000, **1900**, pp. 108–122
3 Roth, P.C., and Miller, B.P.: 'Deep start: a hybrid strategy for automated performance problem searches', in Burkhard Monien and Rainer Feldmann, (Eds): 'Euro-Par 2002 - Paderborn, Germany', *Lect. Notes Comput. Sci.*, 2002, **2400**, pp. 86–96
4 Buck, B., and Hollingsworth, J.K.: 'An API for runtime code patching', *Int. J. High Perform. Comput. Appl.*, 2000, **14**, (4), pp. 317–329
5 Hollingsworth, J.K., Miller, B.P., and Cargille, J.: 'Dynamic program instrumentation for scalable performance tools'. Presented at Scalable high-performance computing Conf., Knoxville, Tennessee, May 1994. ftp://ftp.cs.wisc.edu/paradyn/technical_papers/dyninst.ps.Z
6 Pazandak, P., and Wells, D.: 'ProbeMeister: distributed runtime software instrumentation'. Presented at 1st Int. Workshop on Unanticipated software evolution (USE), Malaga, Spain, June 2002. http: //www.joint.org/use2002/sub/pazandak-ProbeMeister.pdf
7 Dmitriev, M.: 'Application of the HotSwap technology to advanced profiling'. Presented at 1st Int. Workshop on Unanticipated software evolution (USE), Malaga, Spain, June 2002. http://www.joint.org/use2002/sub/dmitriev-hotswapprof.pdf
8 Popovici, A., Gross, T., and Alonso, G.: 'Dynamic weaving for aspect oriented programming'. Presented at 1st Int. Conf. on Aspect-oriented software development (AOSD), Enschede, The Netherlands, 22–26 April 2002. http://ikplab11.inf.ethz.ch:9000/prose/webthings/aosd02.ps
9 Yeung, K., Kelly, P.H.J., and Bennett, S. 'Dynamic instrumentation for Java using a virtual JVM' in Getov, V., *et al.* (Eds): Performance analysis and grid computing (Kluwer, The Netherlands, 2003)
10 Taivalsaari, A.: 'Implementing a Java virtual machine in the Java programming language'. Technical Report TR-98-64, Sun Microsystems, 1998. http://research.sun.com/techrep/1998/abstract-64.html
11 Bala, V., Duesterwald, E., and Banerjia, S.: 'Dynamo: a transparent dynamic optimization system', *ACM SIGPLAN Not.*, 2000, **35**, (5), pp. 1–12
12 Bruening, D., Duesterwald, E., and Amarasinghe, S.: 'Design and implementation at of a dynamic optimization framework for windows'. Presented at 4th ACM Workshop on Feedback-directed and dynamic optimization (FDDO-4), December 2001
13 Yeung, K.C., and Kelly, P.H.J.: 'Optimising Java RMI programs by communication restructuring', in D. Schmidt and M. Endler, (Eds): 'Middleware 2003 - Rio de Janeiro, Brazil', *Lect. Notes Comput. Sci.*, **2672**, pp. 324–343, 2003
14 Vallée-Rai, R., Gagnon, E., Hendren, L.J., *et al.* 'Optimizing Java bytecode using the Soot framework: Is it feasible?', in David A. Watt (Ed.): 'Compiler Construction (CC2000), Berlin, Germany', *Lect. Notes Comput. Sci.*, 2000, **1781** 18–34
15 Kiczales, G., Hilsdale, E., Hugunin, J., *et al.*: 'An overview of Aspect J.', *Lect. Notes Comput. Sci.*, 2001, **2072**, pp. 327–355
16 Standard Performance Evaluation Corporation (SPEC) JVM98 Suite, 1998. Available from http://www.spec.org
17 Liang, S., and Viswanathan, D.: 'Comprehensive profiling support in the Java Virtual Machine'. Proc. 5th USENIX Conference on Object-oriented technologies and systems (COOTS), San Diego, CA, May 1999, pp. 229–240