Dynamic Analysis of Program Concepts in Java

Jeremy Singer
School of Computer Science
University of Manchester, UK
jsinger@cs.man.ac.uk

Chris Kirkham
School of Computer Science
University of Manchester, UK
chris@cs.man.ac.uk

ABSTRACT

Concept assignment identifies units of source code that are functionally related, even if this is not apparent from a syntactic point of view. Until now, the results of concept assignment have only been used for static analysis, mostly of program source code. This paper investigates the possibility of using concept information as part of dynamic analysis of programs. There are two case studies involving (i) a small Java program used in a previous research study; (ii) a large Java virtual machine (the popular Jikes RVM system). These studies demonstrate the usefulness of concept information for dynamic approaches to profiling, debugging and comprehension. This paper also introduces the new idea of feedback-directed concept assignment.

1. INTRODUCTION

This paper fuses together ideas from program *comprehension* (concepts and visualization) with program *compilation* (dynamic analysis). The aim is to provide techniques to visualize Java program execution traces in a user-friendly manner, at a higher level of abstraction than current tools support. These techniques should enable more effective program comprehension, profiling and debugging.

1.1 Concepts

Program concepts are a means of high-level program comprehension. Biggerstaff et al [4] define a concept as 'an expression of computational intent in human-oriented terms, involving a rich context of knowledge about the world.' They argue that a programmer must have some knowledge of program concepts (some informal intuition about the program's operation) in order to manipulate that program in any meaningful fashion. Concepts attempt to encapsulate original design intention, which may be obscured by the syntax of the programming language in which the system is implemented. Concept *selection* identifies how many orthogonal intentions the programmer has expressed in the program. Concept *assignment* infers the programmer's intentions from the program source code. As a simple example, concept assignment would relate the human-oriented concept `buyATrainTicket` with the low-level implementation-oriented artefacts:

```
{   queue();
    requestTicket(destination);
    pay(fare);
    takeTicket();
    sayThankYou();
}
```

Often, human-oriented concepts are expressed using UML diagrams or other high-level specification schemes, which are far removed from the typical programming language sphere of discourse. In contrast, implementation-oriented artefacts are expressed directly in terms of source code features, such as variables and method calls.

Concept assignment is a form of reverse engineering. In effect, it attempts to work backward from source code to recover the 'concepts' that the original programmers were thinking about as they wrote each part of the program. This conceptual pattern matching assists maintainers to search existing source code for program fragments that implement a concept from the application. This is useful for program comprehension, refactoring, and post-deployment extension.

Generally, each individual source code entity implements a single concept. The granularity of concepts may be as small as per-token or per-line; or as large as per-block, per-method or per-class. Often, concepts are visualized by colouring each source code entity with the colour associated with that particular concept. Concept assignment can be expressed mathematically. Given a set U of source code units $u_0, u_1, \ldots, u_n$ and a set C of concepts $c_0, c_1, \ldots, c_m$ then concept assignment is the construction of a mapping from U to C. Often the mapping itself is known as the concept assignment.

Note that there is some overlap between concepts and aspects. Both attempt to represent high-level information coupled with low-level program descriptions. The principal difference is that concepts are universal. Every source code entity implements some concept. In contrast, only some of the source code implements aspects. *Aspects* encapsulate implementation-oriented cross-cutting concerns, whereas *concepts* encapsulate human-oriented concerns which may or may not be cross-cutting.

Throughout this paper, we make no assumptions about how concept selection or assignment takes place. In fact, all the concepts are selected and assigned manually in our two case studies. This paper concentrates on how the concept information is applied, which is entirely independent of how it is constructed. However we note that automatic concept selection and assignment is a non-trivial artificial intelligence problem.

1.2 Dynamic Analysis with Concepts

To date, concept information has only been used for static analysis of program source code or higher-level program descriptions [4, 10, 11]. This work focuses on *dynamic analysis* using concept information, for Java programs. Such dynamic analysis relies on embedded concept information within dynamic execution traces of programs.

1.3 Contributions

This paper makes three major contributions:

1. Section 2 discusses how to represent concepts practically in Java source code and JVM dynamic execution traces.

2. Sections 3.2 and 3.3 outline two different ways of visualizing dynamic concept information.

3. Sections 3 and 4 report on two case studies of systems investigated by dynamic analysis of concepts.

2. CONCEPTS IN JAVA

This section considers several possible approaches for embedding concept information into Java programs. The information needs to be apparent at the source code level (for static analysis of concepts) and also in the execution trace of the bytecode program (for dynamic analysis of concepts).

There are obvious advantages and disadvantages with each approach. The main concerns are:

- ease of marking up concepts, presumably in source code. We hope to be able to do this manually, at least for simple test cases. Nonetheless it has to be simple enough to automate properly.

- ease of gathering dynamic information about concept execution at or after runtime. We hope to be able to use simple dump files of traces of concepts. These should be easy to postprocess with perl scripts or similar.

- ease of analysis of information. We would like to use visual tools to aid comprehension. We hope to be able to interface to the popular Linux profiling tool Kcachegrind [1], part of the Valgrind toolset [16].

The rest of this section considers different possibilities for embedded concept information and discusses how each approach copes with the above concerns.

```
public @interface Concept1 { }
public @interface Concept2 { }
...
@Concept1 public class Test {
    @Concept2 public void f() { ... }
    ...
}
```

Figure 1: Annotation-based concepts in example Java source code

2.1 Annotations

Custom annotations have only been supported in Java since version 1.5. This restricts their applicability to the most recent JVMs, excluding many research tools such as Jikes RVM [2].

Annotations are declared as special `interface` types. They can appear in Java wherever a modifier can appear. Hence annotations can be associated with classes and fields within classes. They cannot be used for more fine-grained (statement-level) markup.

Figure 1 shows an example that uses annotations to support concepts. It would be straightforward to construct and mark up concepts using this mechanism, whether by hand or with an automated source code processing tool.

Many systems use annotations to pass information from the static compiler to the runtime system. An early example is the AJIT system from Azevedo et al [3]. Brown and Horspool present a more recent set of techniques [5].

One potential difficulty with an annotation-based concept system is that it would be necessary to modify the JVM, so that it would dump concept information out to a trace file whenever it encounters a concept annotation.

2.2 Syntax Abuse

Since the annotations are only markers, and do not convey any information other than the particular concept name (which may be embedded in the annotation name) then it is not actually necessary to use the full power of annotations. Instead, we can use *marker* interfaces and exceptions, which are supported by all versions of Java. The Jikes RVM system [2] employs this technique to convey information to the JIT compiler, such as inlining information and specific calling conventions.

This information can only be attached to classes (which reference marker interfaces in their `implements` clauses) and methods (which reference marker exceptions in their `throws` clauses). No finer level of granularity is possible in this model. Again, these syntactic annotations are easy to insert into source code. However a major disadvantage is the need to modify the JVM to dump concept information when it encounters a marker during program execution.

2.3 Custom Metadata

Concept information can be embedded directly into class and method names. Alternatively each class can have a

special concept field, which would allow us to take advantage of the class inheritance mechanism. Each method can have a special concept parameter. However this system is thoroughly intrusive. Consider inserting concept information after the Java source code has been written. The concept information will cause wide-ranging changes to the source code, even affecting the actual API! This is an unacceptably invasive transformation. Now consider using such custom metadata at runtime. Again, the metadata will only be useful on a specially instrumented JVM that can dump appropriate concept information as it encounters the metadata.

2.4 Custom Comments

A key disadvantage of the above approaches is that concepts can only be embedded at certain points in the program, for specific granularities (classes and methods). In contrast, comments can occur at arbitrary program points. It would be possible to insert concept information in special comments, that could be recognised by some kind of preprocessor and transformed into something more useful. The Javadoc system supports custom tags in comments. This would allow us to insert concept information at arbitrary program points! Then we use a Javadoc style preprocessor (properly called a *doclet system* in Java) to perform source-to-source transformation.

We eventually adopted this method for supporting concepts in our Java source code, due to its simplicity of concept creation, markup and compilation.

The custom comments can be transformed to suitable statements that will be executed at runtime as the flow of execution crosses the marked concept boundaries. Such a statement would need to record the name of the concept, the boundary type (entry or exit) and some form of timestamp.

In our first system (see Section 3) the custom comments are replaced by simple `println` statements and timestamps are computed using the `System.nanoTime()` Java 1.5 API routine, thus there is no need for a specially instrumented JVM.

In our second system (see Section 4) the custom comments are replaced by Jikes RVM specific logging statements, which more efficient than `println` statements, but entirely non-portable. Timestamps are computed using the IA32 `TSC` register, via a new 'magic' method. Again this should be more efficient than using the `System.nanoTime()` routine.

In order to change the runtime behaviour at concept boundaries, all that is required is to change the few lines in the concept doclet that specify the code to be executed at the boundaries. One could imagine that more complicated code is possible, such as data transfer via a network socket in a distributed system. However note the following efficiency concern: One aim of this logging is that it should be unobtrusive. The execution overhead of concept logging should be no more than noise, otherwise any profiling will be inaccurate! In the studies described in this paper, the mean execution time overhead for running concept-annotated code is 35% for the small Java program (Section 3) but only 2% for the large Java program (Section 4).

```
// @concept_begin Concept1
public class Test {
    public void f() {
        ....

        while (...)
            // @concept_end Concept1
            // @concept_begin Concept2
    }
    ...
}
// @concept_end Concept2
```

Figure 2: Comments-based concepts in example Java source code

Figure 2 shows some example Java source code with concepts marked up as custom comments.

There are certainly other approaches for supporting concepts, but the four presented above seemed the most intuitive and the final one seemed the most effective.

3. DYNAMIC ANALYSIS FOR SMALL JAVA PROGRAM

The first case study involves a small Java program called `BasicPredictors` which is around 500 lines in total. This program analyses textual streams of values and computes how well these values could be predicted using standard hardware prediction mechanisms. It also computes information theoretic quantities such as the entropy of the stream. The program was used to generate the results for an earlier study on method return value predictability for Java programs [23].

3.1 Concept Assignment

The `BasicPredictors` code is an interesting subject for concept assignment since it calculates values for different purposes in the same control flow structures (for instance, it is possible to re-use information for prediction mechanisms to compute entropy).

We have identified four concepts in the source code, shown below.

system: the default concept. Prints output to stdout, reads in input file. Reads arguments. allocates memory.

predictor_compute: performs accuracy calculation for several *computational* value prediction mechanisms.

predictor_context: performs accuracy calculation for *context-based* value prediction mechanism (table lookup).

entropy: performs calculation to determine information theoretic entropy of entire stream of values.

The concepts are marked up manually using custom Javadoc tags, as described in Section 2.4. This code is transformed using the custom doclet, so the comments have been replaced by `println` statements that dump out concept information at execution time. After we have executed the

instrumented program and obtained the dynamic execution trace which includes concept information, we are now in a position to perform some dynamic analysis.

3.2 Dynamic Analysis for Concept Proportions

The first analysis simply processes the dynamic concept trace and calculates the overall amount of time spent in each concept. (At this stage we do not permit nesting of concepts, so code can only belong to a single concept at any point in execution time.) This analysis is similar to standard function profiling, except that it is now based on specification-level features of programs, rather than low-level syntactic features such as function calls.

The tool outputs its data in a format suitable for use with the Kcachegrind profiling and visualization toolkit [1]. Figure 3 shows a screenshot of the Kcachegrind system, with data from the `BasicPredictors` program. It is clear to see that most of the time is spent in the `system` concept. It is also interesting to note that `predictor_context` is far more expensive than `predictor_compute`. This is a well-known fact in the value prediction literature [19].

3.3 Dynamic Analysis for Concept Phases

While the analysis above is useful for determining overall time spent in each concept, it gives no indication of the temporal relationship between concepts.

It is commonly acknowledged that programs go through different phases of execution which may be visible at the microarchitectural [7] and method [9, 15] levels of detail. It should be possible to visualize phases at the higher level of concepts also.

So the visualization in Figure 4 attempts to plot concepts against execution time. The different concepts are highlighted in different colours, with time running horizontally from left-to-right. Again, this information is extracted from the dynamic concept trace using a simple perl script, this time visualized as HTML within any standard web browser.

There are many algorithms to perform phase detection but even just by observation, it is possible to see three phases in this program. The startup phase has long periods of `system` (opening and reading files) and `predictor_context` (setting up initial table) concept execution. This is followed by a periodic phase of prediction concepts, alternately `predictor_context` and `predictor_compute`. Finally there is a result report and shutdown phase.

3.4 Applying this Information

How can these visualizations be used? They are ideal for visualization and program comprehension. They may also be useful tools for debugging (since concept anomalies often indicate bugs [22]) and profiling (since they show where most of the execution time is spent).

Extensions are possible. At the moment we only draw a single bar. It would be necessary to move to something resembling a Gantt chart if we allow nested concepts (so a source code entity can belong to more than one concept at once) or if we have multiple threads of execution (so more than one concept is being executed at once).

4. DYNAMIC ANALYSIS FOR LARGE JAVA PROGRAM

The second case study uses Jikes RVM [2] which is a reasonably large Java system. It is a production-quality adaptive JVM written in Java. It has become a significant vehicle for JVM research, particularly into adaptive compilation mechanisms and garbage collection. All the tests reported in this section use Jikes RVM version 2.4.4 on IA32 Linux.

A common complaint from new users of Jikes RVM is that it is hard to understand how the different adaptive runtime mechanisms operate and interact. So this case study selects some high-level concepts from the adaptive infrastructure, thus enabling visualization of runtime behaviour.

After some navigation of the Jikes RVM source code, we inserted concepts tags around a few key points that encapsulate adaptive mechanisms like garbage collection and method compilation. Note that all code not in such a concept (both Jikes RVM code and user application code) is in the default `system` concept.

4.1 Garbage Collection

Figure 5 shows concept visualization of two runs of the `_201_compress` benchmark from SPEC JVM98. The top run has an initial and maximum heap size of 20MB (`-Xms20M -Xmx20M`) whereas the bottom run has an initial and maximum heap size of 200MB. It is clear to see that garbage collection occurs far more frequently in the smaller heap, as might be expected. In the top run, the garbage collector executes frequently and periodically, doing a non-trivial amount of work as it compacts the heap. In the bottom run, there is a single trivial call to `System.gc()` as part of the initial benchmark harness code. After this, garbage collection is never required so we assume that the heap size is larger than the memory footprint of the benchmark.

Many other garbage collection investigations are possible. So far we have only considered varying the heap configuration. It is also possible to change the garbage collection algorithms in Jikes RVM, and determine from concept visualizations what effect this has on runtime performance.

4.2 Runtime Compilation

In the investigation above, the `compilation` concept only captures optimizing compiler behaviour. However since Jikes RVM is an adaptive compilation system, it has several levels of compilation. The cheapest compiler to run (but one that generates least efficient code) is known as the *baseline* compiler. This is a simple macro-expansion routine from Java bytecode instructions to IA32 assembler. Higher levels of code efficiency (and corresponding compilation expense!) are provided by the sophisticated *optimizing* compiler, which can operate at different levels since it has many flags to enable or disable various optimization strategies. In general, Jikes RVM initially compiles application methods using the baseline compiler. The adaptive monitoring system identifies 'hot' methods that are frequently executed, and these are candidates for optimizing compilation. The hottest methods should be the most heavily optimized methods.

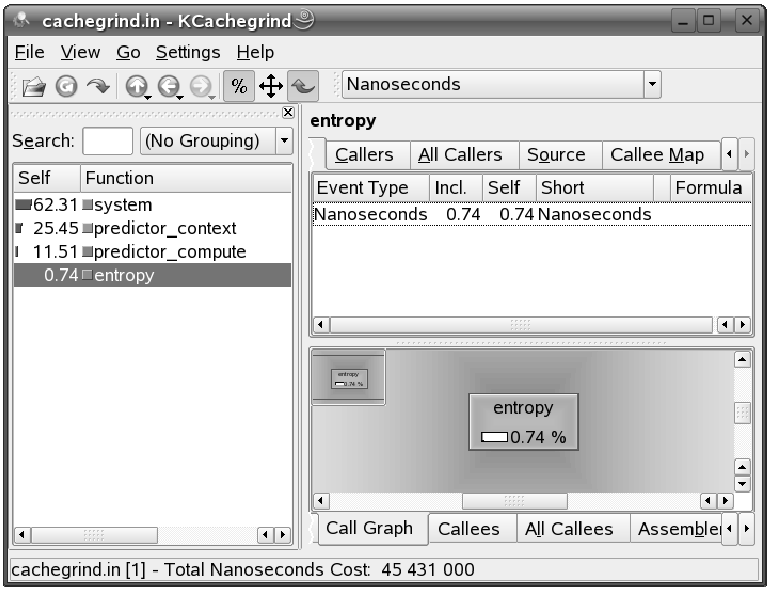

Figure 3: Screenshot of Kcachegrind tool visualizing percentage of total program runtime spent in each concept

Figure 4: Simple webpage visualizing phased behaviour of concept execution trace

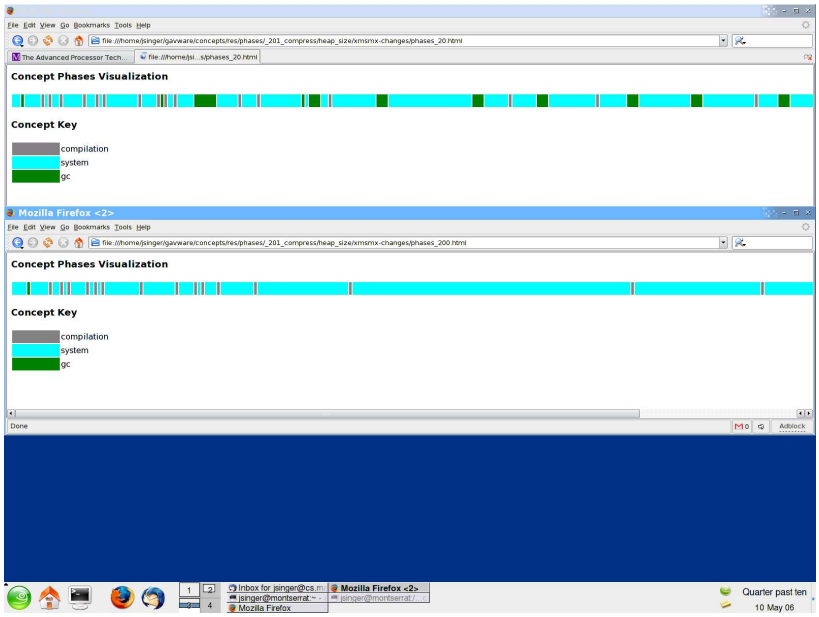

Figure 5: Investigation of garbage collection activity for different heap sizes

We use the `_201_compress` benchmark from SPEC JVM98 again. Figure 6 shows the start of benchmark execution using the default (adaptive) compilation settings (top) and specifying that the optimizing compiler should be used by default (bottom, `-X:aos:initial_compiler=opt`). In the top execution, the baseline compiler is invoked frequently for a small amount of time each invocation. On the other hand, in the bottom execution, the optimizing compiler is invoked more frequently. It takes a long time on some methods (since it employs expensive analysis techniques). However note that even with the optimizing compiler as default, it is still the case that there are some baseline compiled methods. This is not necessarily intuitive, but it is clear to see from the visualization!

Figure 7 shows the same execution profiles, only further on in execution time. The top visualization (default compilation settings) shows that many methods are now (re)compiled using the optimizing compiler. As methods get hot at different times, optimizing compiler execution is scattered across runtime. In the bottom visualization, once all the methods have been compiled with the optimizing compiler, there is generally no need for recompilation.

Note that both Figures 6 and 7 demonstrate that the optimizing compiler causes more garbage collection! The compilation system uses the same heap as user applications, and there is intensive memory usage for some optimizing compiler analyses.

There are plenty of other investigations to be performed with the Jikes RVM compilation system. In addition, we hope to identify other interesting concepts in Jikes RVM.

5. RELATED WORK

Hauswirth et al [13] introduce the discipline of *vertical profiling* which involves monitoring events at all levels of abstraction (from hardware counters through virtual machine state to user-defined application-specific debugging statistics). Their system is built around Jikes RVM. It is able to correlate events at different abstraction levels in dynamic execution traces. They present some interesting case studies to explain performance anomalies in standard benchmarks. Our work focuses on user-defined high-level concepts, and how source code and dynamic execution traces are partitioned by concepts. Their work relies more on event-based counters at all levels of abstraction in dynamic execution traces.

GCspy [18] is an elegant visualization tool also incorporated with Jikes RVM. It is an extremely flexible tool for visualizing heaps and garbage collection behaviour. Our work examines processor utilization by source code concepts, rather than heap utilization by source code mutators.

Sefika et al [20] introduce *architecture-oriented visualization*. They recognise that classes and methods are the base units of instrumentation and visualization, but they state that higher-level aggregates (which we term concepts!) are more likely to be useful. They instrument methods in the memory management system of an experimental operating system. The methods are grouped into architectural units (concepts) and instrumentation is enabled or disabled for each concept. This allows efficient partial instrumentation on a per-concept basis, with a corresponding reduction in the dynamic trace data size. Our instrumentation is better in that it can operate at a finer granularity than method-level. However our instrumentation cannot be selectively disabled, other than by re-assigning concepts to reduce the number of concept boundaries.

Sevitsky et al [21] describe a tool for analysing performance of Java programs using *execution slices*. An execution slice is a set of program elements that a user specifies to belong

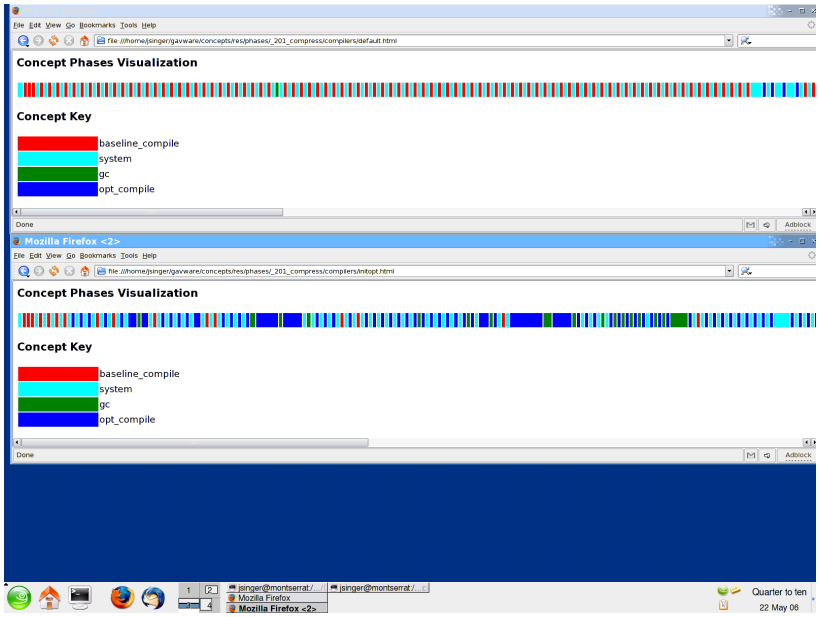

Figure 6: Investigation of initial runtime compilation activity for different adaptive configurations. Top graph is for default compilation strategy, i.e. use baseline before optimizing compiler. Bottom graph is for optimizing compilation strategy, i.e. use optimizing compiler whenever possible.

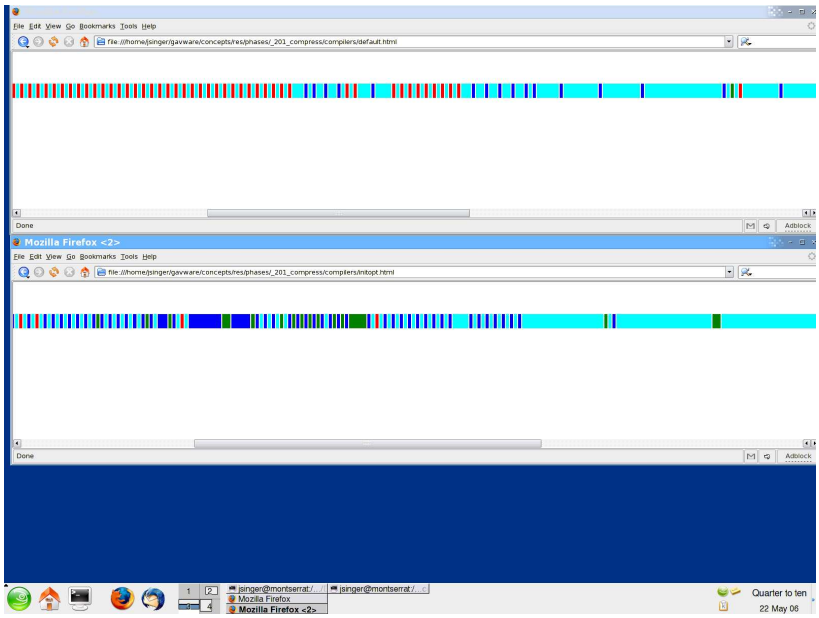

Figure 7: Investigation of later runtime compilation activity for different adaptive configurations. (Same configurations as in previous figure.)

to the same category—again, this is a disguised concept! Their tool builds on the work of Jinsight [17] which creates a database for a Java program execution trace. Whereas Jinsight only operates on typical object-oriented structures like classes and methods, the tool by Sevitsky et al handles compound execution slices. Again, our instrumentation is at a finer granularity. Our system also does not specify how concepts are assigned. They allow manual selection or automatic selection based on attribute values—for instance, method invocations may be characterized as slow, medium or fast based on their execution times.

Eng [8] presents a system for representing static and dynamic analysis information in an XML document framework. All Java source code entities are represented, and may be tagged with analysis results. This could be used for static representation of concept information, but it is not clear how the information could be extracted at runtime for the dynamic execution trace.

Other Java visualization research projects (for example, [6, 12]) instrument JVMs to dump out low-level dynamic execution information. However they have no facility for dealing with higher-level concept information. In principle it would be possible to reconstruct concept information from the lower-level traces in a postprocessing stage, but this would cause unnecessarily complication, inefficiency and potential inaccuracy.

6. CONCLUDING REMARKS

Until now, concepts have been a compile-time feature. They have been used for static analysis and program comprehension. This work has driven concept information through the compilation process from source code to dynamic execution trace, and made use of the concept information in dynamic analyses. This follows the recent trend of retaining compile-time information until execution time. Consider typed assembly language, for instance [14].

Feedback-directed concept assignment is the process of (1) selecting concepts, (2) assigning concepts to source code, (3) running the program, (4) checking results from dynamic analysis of concepts and (5) using this information to repeat step (1)! This is similar to feedback-directed (or profile-guided) compilation. In effect, this is how we made the decision to examine both baseline and optimizing compilers in Section 4.2 rather than just optimizing compiler as in Section 4.1. The process could be entirely automated, with sufficient tool support.

With regard to future work, we should incorporate these analyses and visualizations into an integrated development environment such as Eclipse. Further experience reports would be helpful, as we conduct more investigations with these tools. The addition of timestamps information to the phases visualization (Section 3.3) would make the comparison of different runs easier. We need to formulate other dynamic analyses in addition to concept proportions and phases. One possibility is *concept hotness*, which would record how the execution profile changes over time, with more or less time being spent executing different concepts. This kind of information is readily available for method-level analysis in Jikes RVM, but no-one has extended it to higher-level abstractions.

7. ACKNOWLEDGEMENTS

The first author is employed as a research associate on the Jamaica project, which is funded by the EPSRC Portfolio Award GR/S61270.

The small Java program described in Section 3 was written by Gavin Brown. We thank him for allowing us to analyse his program and report on the results.

Finally we thank the conference referees for their thoughtful and helpful feedback on this paper.

8. REFERENCES

[1] 2005. Kcachegrind profiling visualization, Josef Weidendorfer. see kcachegrind.sourceforge.net for details.

[2] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, 1999.

[4] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[5] R. H. F. Brown and R. N. Horspool. Object-specific redundancy elimination techniques. In *Proceedings of Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2006.

[6] P. Dourish and J. Byttner. A visual virtual machine for Java programs: exploration and early experiences. In *Proceedings of the ICDMS Workshop on Visual Computing*, 2002.

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, 2003.

[8] D. Eng. Combining static and dynamic data in code visualization. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–50, 2002.

[9] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 270–287, 2004.

[10] N. Gold. Hypothesis-based concept assignment to support software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 545–548, 2001.

[11] N. Gold, M. Harman, D. Binkley, and R. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software—Practice & Experience*, 35(10):977–1006, 2005.

[12] M. Golm, C. Wawersich, J. Baumann, M. Felser, and J. Kleinöder. Understanding the performance of the Java operating system JX using visualization techniques. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, page 230, 2002.

[13] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 251–269, 2004.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[15] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 27–33, 2004.

[16] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, 2003.

[17] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 151–162, 2002.

[18] T. Printezis and R. Jones. GCspy: an adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 343–358, 2002.

[19] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, 1997.

[20] M. Sefika, A. Sane, and R. H. Campbell. Architecture-oriented visualization. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 389–405, 1996.

[21] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *Proceedings of TOOLS Europe Conference*, 2001.

[22] J. Singer. Concept assignment as a debugging technique for code generators. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, 2005.

[23] J. Singer and G. Brown. Return value prediction meets information theory. In *Proceedings of the 4th Workshop on Quantitative Aspects of Programming Languages*, 2006. To appear in *Electronic Notes in Theoretical Computer Science*.