Dynamic Analysis of Java Program Concepts for Visualization and Profiling

Jeremy Singer *, Chris Kirkham

School of Computer Science, University of Manchester, 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 within a framework for dynamic analysis of programs. The paper presents two case studies involving a small Java program used in a previous research exercise, and a large Java virtual machine (the popular Jikes RVM system). These studies investigate two applications of dynamic concept information: visualization and profiling. The paper demonstrates two different styles of concept visualization, which show the proportion of overall time spent in each concept and the sequence of concept execution, respectively. The profiling study concerns the interaction between runtime compilation and garbage collection in Jikes RVM. For some benchmark cases, we are able to obtain a significant reduction in garbage collection time. We discuss how this phenomenon might be harnessed to optimize the scheduling of garbage collection in Jikes RVM.

Key words: Concept assignment, Dynamic analysis, Jikes RVM

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

* Corresponding author.
Email addresses: `jsinger@cs.man.ac.uk` (Jeremy Singer), `chris@cs.man.ac.uk` (Chris Kirkham).

techniques should enable more effective program comprehension, profiling and debugging. The overall objective is an improvement in software engineering practice.

1.1 Concepts

Program concepts are a means of high-level program comprehension. Biggerstaff et al. [1] 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.

Every source code entity is part of the implementation of some 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 belongs to 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. Section 2.4 develops this relationship.

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. For instance, Biggerstaff et al. describe a semi-automated design recovery system called DESIRE [1]. This uses a pre-computed domain model and a connectionist inference engine to perform the assignment. Gold and Bennett describe a hypothesis-based system [2]. This applies information from a human-generated knowledge base to source code using self-organizing maps to assign concepts.

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 [1,3,4]. This work focuses on *dynamic analysis* of Java programs using concept information. Such dynamic analysis relies on embedded concept information within source code and dynamic execution traces of programs. This paper discusses various techniques for encoding, extracting and applying this concept information.

1.3 Contributions

This paper makes four major contributions:

(1) Section 2 discusses how to represent concepts practically in Java source code and 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.
(4) Section 5 describes how concepts are used to profile garbage collection behaviour within a virtual machine.

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 small test cases. Nonetheless it has to be simple enough for straightforward automation.
- Granularity of concept annotations. Ideally we would like to place concept boundaries at arbitrary syntactic positions in the source code.
- 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 [5], part of the Valgrind toolset [6].

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

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 [1] [7].

[1] The latest versions of Jikes RVM (post 2.4.5) have added support for custom annotations. We plan to look into how this allows us to extend our approach.

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

Fig. 1. Example Java source code that uses annotations to represent concepts

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

128 Figure 1 shows a program fragment that uses annotations to represent concepts
129 in source code. It would be straightforward to construct and mark up concepts
130 using this mechanism, whether by hand or with an automated source code
131 processing tool.

132 Many systems use annotations to pass information from the static compiler
133 to the runtime system. An early example is the AJIT system from Azevedo et
134 al. [8]. Brown and Horspool present a more recent set of techniques [9].

135 One potential difficulty with an annotation-based concept system is that it
136 would be necessary to modify the JVM, so that it would dump concept in-
137 formation out to a trace file whenever it encounters a concept annotation at
138 runtime.

139 *2.2 Syntax Abuse*

140 Since the annotations are only markers, and do not convey any information
141 other than the particular concept name (which may be embedded in the an-
142 notation name) then it is not actually necessary to use the full power of an-
143 notations. Instead, we can use *marker* interfaces and exceptions, which are
144 supported by all versions of Java. The Jikes RVM system [7] employs this
145 technique to convey information to the JIT compiler, such as inlining infor-
146 mation and specific calling conventions.

147 This information can only be attached to classes (which reference marker
148 interfaces in their `implements` clauses) and methods (which reference marker
149 exceptions in their `throws` clauses). No finer level of granularity is possible in
150 this model. Again, these syntactic annotations are easy to insert into source
151 code. Figure 2 shows a program fragment that uses syntax abuse to represent
152 concepts in source code. However a major disadvantage is the need to modify

```
public class Concept1 extends Exception {
}

public class Test {
    public void f() throws Concept1 { ... }
    ...
}
```

Fig. 2. Example Java source code that uses syntax abuse to represent concepts

```
public class Test {

    public static final int CONCEPT1 = ...;

    public void f(int concept) { ... }
    ...
}
```

Fig. 3. Example Java source code that uses metadata to represent concepts

153 the JVM to dump concept information when it encounters a marker during
154 program execution.

155 *2.3 Custom Metadata*

156 Concept information can be embedded directly into class and method names.
157 Alternatively each class can have a special concept field, which would allow
158 us to take advantage of the class inheritance mechanism. Each method can
159 have a special concept parameter. However this system is thoroughly intrusive.
160 Consider inserting concept information after the Java source code has been
161 written. The concept information will cause wide-ranging changes to the source
162 code, even affecting the actual API. Figure 3 shows a program fragment that
163 uses metadata to represent concepts in source code. This is an unacceptably
164 invasive transformation. Now consider using such custom metadata at runtime.
165 Again, the metadata will only be useful on a specially instrumented JVM that
166 can dump appropriate concept information as it encounters the metadata.

167 *2.4 Aspects*

168 Aspect-oriented programming (AOP) [10] provides new constructs to han-
169 dle cross-cutting concerns in programs. Such concerns cannot be localized
170 within single entities in conventional programming languages. In AOP, cross-
171 cutting concerns are encapsulated using *aspects*. Aspects are encoded in sepa-

```
aspect Concept1 {
    OutputStream conceptStream = System.out;

    pointcut boundary():
        call (void f())
        /* may have other methods here */
        ;

     before(): boundary() {
        conceptStream.println(``concept1 entry'');
    }

     after(): boundary() {
        conceptStream.println(``concept1 exit'');
    }
}
```

Fig. 4. Example AspectJ source code that uses aspects to represent concepts

rate source code units, distinct from the rest of the program source code. The code contained in an aspect is known as *aspect advice*. A point in the program source code where a cross-cutting concern occurs is known as a *join point*. At some stage during the compilation process, the appropriate aspect advice is *woven* into the main program source code at the relevant join points. The set of all join points for a particular aspect is known as a *point cut*. Event logging is the canonical aspect. Note that this is similar to generating a dynamic execution trace of concepts. The rest of this section uses AspectJ [11], which is the standard aspect-oriented extension of Java.

Each individual concept can be represented by a single aspect. Point cuts specify appropriate method entry and exit events for concept boundaries. Aspect advice outputs concept logging information to a dynamic execution trace stream. Figure 4 shows a program fragment that uses aspects to represent concepts in source code.

One problem with aspects is that the join points (program points at which concept boundaries may be located) are restricted. They are more general than simply method entry and exit points, but there are still some constraints. The AspectJ documentation [11] gives full details. Another disadvantage is that aspects are not integrated into the program until compilation (or possibly even execution [12]) time. Thus when a programmer inspects the original source code, it is not apparent where concept boundaries lie. It is necessary to consider both aspect advice and program code in parallel to explore concepts at the source code level.

2.5 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. For instance, the Javadoc system supports custom tags in comments. This approach enables the insertion of concept information at arbitrary program points. A Javadoc style preprocessor (properly called a *doclet system* in Java) can perform appropriate 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. Figure 5 shows a program fragment that uses custom comments to represent concepts in source code.

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 are more efficient than `println` statements, but entirely nonportable. 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 logging 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). This disparity is due to the relative differences in concept granularity in the two studies. All the experiments in this paper are based on *exhaustive tracing* of concept information. Note that a *statistical sampling* approach would require less overhead

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

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

Fig. 5. Example Java source code that uses comments to represent concepts

234 than exhaustive tracing. A concept sampler could be incorporated with the
235 timer-based method profiler used in most adaptive JVM systems to identify
236 frequently executed regions of code.

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

240 3 Dynamic Analysis for Small Java Program

241 The first case study involves a small Java program called `BasicPredictors`
242 which is around 500 lines in total. This program analyses streams of ASCII
243 characters encoding method return values. It computes how well these values
244 could be predicted using standard hardware mechanisms such as last value
245 prediction [13] and finite context method [14]. The program also computes
246 information theoretic quantities such as the entropy of the value stream. We
247 used this program to generate the results for an earlier study on method return
248 value predictability for Java programs [15].

249 *3.1 Concept Assignment*

250 The `BasicPredictors` code is an interesting subject for concept assignment
251 since it calculates values for different purposes in the same control flow struc-
252 tures (for instance, it is possible to re-use information for prediction mecha-
253 nisms to compute entropy).

254 We have identified four concepts in the source code.

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 2.5. 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 [5]. Figure 6 shows a screenshot of the Kcachegrind system, with data from the `BasicPredictors` program. It is clear to see that most of the time (62%) is spent in the `system` concept. It is also interesting to note that `predictor_context` (25%) is more expensive than `predictor_compute` (12%). This is a well-known fact in the value prediction literature [14].

3.3 Dynamic Analysis for Concept Phases

While this analysis is useful for determining the 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 [16] and method [17,18] levels of detail. It should be possible to visualize phases at the higher level of concepts also.

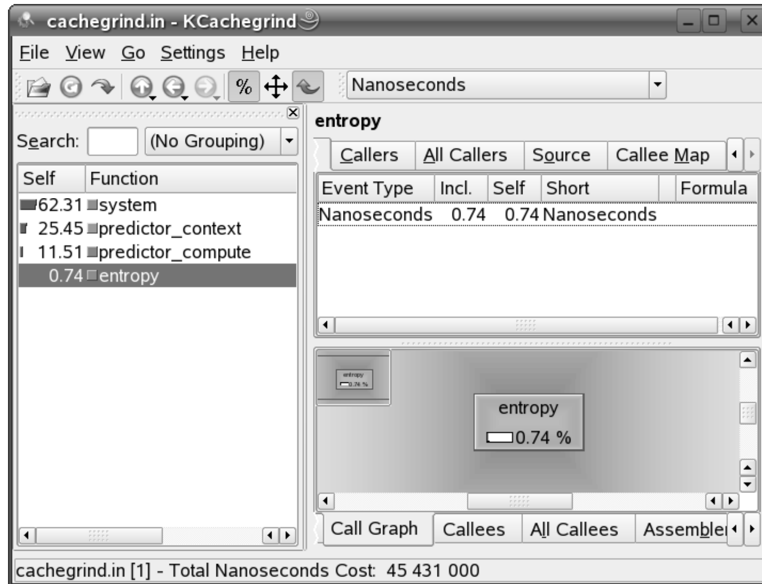

Fig. 6. Screenshot of Kcachegrind tool visualizing percentage of total program runtime spent in each concept

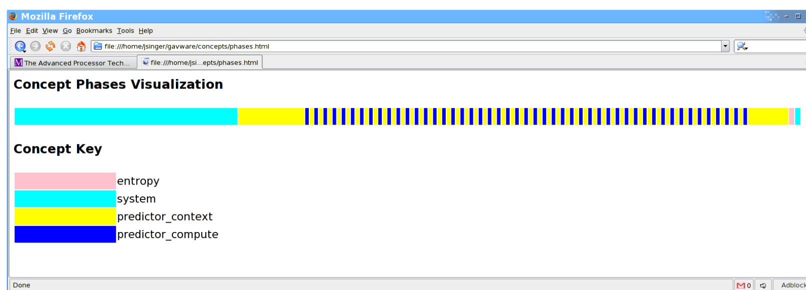

Fig. 7. Simple webpage visualizing phased behaviour of concept execution trace

So the visualization in Figure 7 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 program comprehension. They may also be useful tools for debugging (since concept anomalies often indicate bugs [19]) and profiling (since they show where most of the execution time is spent).

This simple one-dimensional visualization of dynamic concept execution sequences can be extended easily. 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 [7] which is a reasonably large Java system, around 300,000 lines of code. It is a production-quality adaptive JVM written in Java. It has become a significant vehicle for virtual machine (VM) research, particularly into adaptive compilation mechanisms and garbage collection. All the tests reported in this section use Jikes RVM version 2.4.4, development configuration, Linux/IA-32 build and single pthread VM runtime.

Like all high-performance VMs, Jikes RVM comprises a number of adaptive runtime subsystems, which are invoked on-demand as user code executes. These include just-in-time compilation, garbage collection and thread scheduling. A common complaint from new users of Jikes RVM is that it is hard to understand how the different subsystems operate and interact. The *programmer* is not aware of how and when they will occur, unless he explicitly requests their services by sending messages like `System.GC()`, but this is rare. Similarly, the *user* is not aware of when these subsystems are operating, as the code executes. They are effectively invisible, from both a static and a dynamic point of view. So this case study selects some high-level concepts from the adaptive infrastructure, thus enabling visualization of runtime behaviour.

330 *4.1 Visualized Subsystems*

331 After some navigation of the Jikes RVM source code, we inserted concept
332 tags around a few key points that encapsulate the adaptive mechanisms of
333 (i) garbage collection and (ii) method compilation. These are the dominant
334 VM subsystems in terms of execution time. Other VM subsystems, such as
335 the thread scheduler, have negligible execution times so we do not profile
336 them. Note that all code not in an explicit concept (both Jikes RVM code and
337 user application code) is in the default `unmarked` concept. Figure 8 gives the
338 different concepts and their colours. Figures 9–11 show different runs of the
339 `_201_compress` benchmark from the SPECjvm98 suite, and how the executed
340 concepts vary over time.

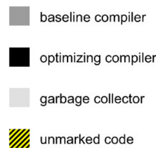

Fig. 8. Key for concept visualizations

341 The *garbage collection* VM subsystem (GC) manages memory. It is invoked
342 when the heap is becoming full, and it detects unreachable (*dead*) objects and
343 deletes them, thus freeing up heap space for new objects. All our experiments
344 use the default Jikes RVM generational mark-sweep garbage collection algo-
345 rithm. This algorithm's distinct behaviour is clear to see from the concept
346 visualizations. There are two generations: nursery and mature. The nursery
347 generation is cheap to collect, so generally a small amount of GC time is spent
348 here. On the other hand, the mature generation is more expensive to collect
349 but collections are less frequent, so occasionally a longer GC time occurs. This
350 is most apparent in Figure 9. Most GCs are short but there is one much longer
351 GC around 45% of the way through the execution.

352 The *compilation* VM subsystem is divided between two concepts. The *base-*
353 *line* compiler is a simple bytecode macro-expansion scheme. It runs quickly
354 but generates inefficient code. On the other hand the *optimizing* compiler (opt-
355 comp) is a sophisticated program analysis system. It runs slowly but generates
356 highly optimized code. Generally all methods are initially compiled with the
357 baseline compiler, but then frequently executed (*hot*) methods are recompiled
358 with the optimizing compiler. The time difference is clear in the visualiza-
359 tions. For instance, the bottom trace in Figure 10 has many short baseline
360 compilations and a few much longer optimizing compilations.

361 *4.2 Subsystem Properties*

362 This section presents five VM subsystem properties that our concept visual-
363 izations clearly demonstrate. This enables us to gain a better understanding
364 of VM behaviour in general.

365 *4.2.1 Pervasive*

366 VM subsystems are active throughout the entire program lifetime. Figure 9
367 illustrates this point. It is noticeable that the density of VM code in relation
368 to user code changes over time. It appears that VM code is more frequent
369 near the beginning of execution. This is because the compiler compiles every
370 method immediately before it is executed for the first time. Later compilation
371 activity generally involves selective optimizing recompilation of hot methods.

372 *4.2.2 Significant runtime*

373 Visualizations like Figure 9 show that VM code occupies a significant pro-
374 portion of total execution time. The total execution time is shared between
375 application code and VM code. For the long-running benchmark program used
376 in this study, around 90% of time is spent in user code and 10% in VM code.
377 The VM time should be more significant for shorter programs. It is also inter-
378 esting to discover how the VM execution time is divided between the various
379 subsystems. For the benchmark used in this study, the VM spends at least
380 twice as long in compilation as in garbage collection.

381 *4.2.3 Dependence on VM configuration*

382 Modern adaptive runtimes are highly configurable. It is possible to specify
383 policies and parameters for all subsystems. These have a major impact on
384 system performance. Previously it was not possible to see exactly how vary-
385 ing configurations changed overall behaviour. Now our concept visualizations
386 make this task straightforward. Figure 10 shows two runs of the same program,
387 but with different VM configurations. The top trace uses the default VM com-
388 pilation policy, which at first compiles all methods using the baseline compiler
389 then recompiles hot methods with the more expensive optimizing compiler.
390 The bottom trace uses a modified compilation policy, which initially uses the
391 optimizing compiler rather than the baseline compiler, as much as possible.
392 Note that Jikes RVM requires that some methods must be compiled at baseline
393 level.

Fig. 9. Pervasive nature of VM subsystem execution

Fig. 10. Varying nature of VM subsystem execution, illustrated by two runs of the same benchmark program

Fig. 11. Periodic nature of VM subsystem execution

394 *4.2.4 Interactivity*

395 VM subsystems are not entirely independent. They affect one another in subtle
396 ways. For instance in Figure 10, it is clear to see that greater use of the opti-
397 mizing compiler causes increased garbage collection activity. This is because
398 the optimizing compiler generates many intermediate program representations
399 and temporary data structures as it compiles methods, thus filling up heap
400 space. Note that there is a single heap shared between VM code and user code.
401 Section 5 explores this interaction in more detail.

402 *4.2.5 Periodicity*

403 Programs go through different phases and exhibit periodic patterns, as Section
404 3.3 mentions. Figure 11 demonstrates that VM code may be periodic too. In
405 this execution trace, the program is memory-intensive and the heap has been
406 restricted to a small size. Thus the garbage collector has to run frequently,
407 and if the memory load is constant over time, then the garbage collector will
408 run periodically.

409 **5 Profiling Garbage Collection**

410 Section 4.2 noted that optimizing compilation (optcomp) frequently triggers
411 GC. We assume this is because optcomp creates many intermediate data struc-
412 tures such as static single assignment form when it analyses a method. This
413 takes up space in the heap. (VM and user code share the same heap in Jikes
414 RVM system.) However most of these intermediate compilation objects die as

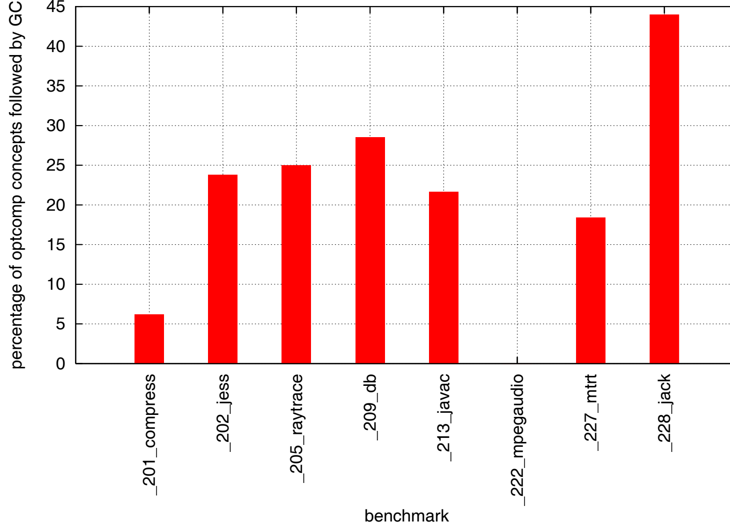

Fig. 12. Percentage of optcomp concepts that are followed by GC, for each benchmark program

415 soon as the compilation completes. A recent study [20] shows that it is most
416 efficient to do GC when the proportion of dead to live data on the heap is max-
417 imal, even if the heap is not entirely full. Processing live data wastes GC time.
418 Live data must be scanned and perhaps copied. On the other hand, dead data
419 may be immediately discarded. This insight leads to our idea that it may be
420 good to perform GC *immediately after optcomp*. Thus we consider modifying
421 the VM to force GC automatically after every optimizing compilation.

422 We query a set of dynamic execution traces to determine how often GC follows
423 optcomp, in a standard Jikes RVM system setup. We use the default GC strat-
424 egy (generational mark-sweep) with standard heap sizes (50MB start, 100MB
425 maximum). We gather concept data from the entire SPECjvm98 benchmark
426 suite. For each benchmark, we measure the percentage of optcomp concepts
427 that are followed by GC (i.e. GC is the next VM concept after optcomp,
428 possibly with some intervening unmarked concept code). Figure 12 shows the
429 results. For some programs, around 25% of optcomp concepts are followed
430 by GC. However the proportion is much lower for others. This suggests that
431 any optimization should be program-specific. Presumably since some methods
432 have larger and more complex methods, optcomp has to do more work and
433 uses more memory.

434 Now we modify the optimizing compiler so that it forces a GC immediately
435 after it has completed an optcomp concept (eager GC-after-optcomp). We
436 hope to target the heap when a large proportion of objects have just become
437 dead. We use the Jikes RVM memory management toolkit harness code to
438 measure the amount of time spent in GC throughout the entire execution
439 of each benchmark. In order to stress the GC subsystem, we decide to use
440 relatively small heap sizes for each benchmark. We determine the minimum

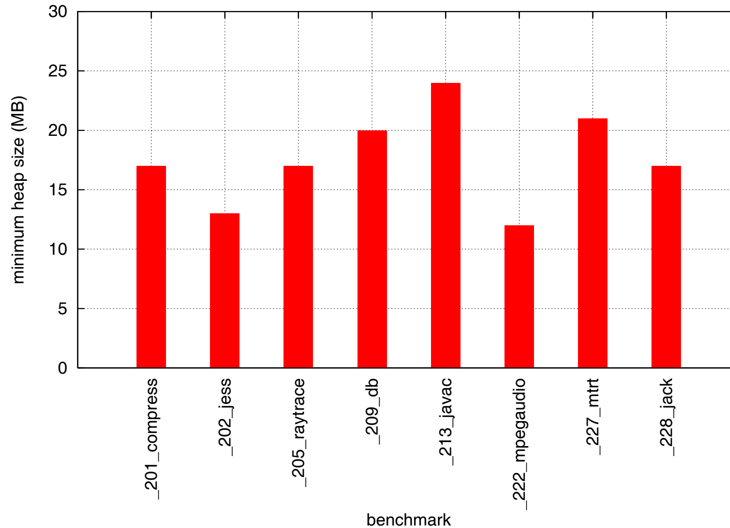

Fig. 13. Minimum possible Jikes RVM heap sizes for each benchmark program

fixed heap size (specified using the `-Xms -Xmx` options for Jikes RVM) in which each benchmark will run without throwing any out-of-memory exceptions. Note that within the fixed heap, the nursery generation may expand to fill the available free space. Figure 13 shows these minimum heap sizes for each benchmark. We conducted three sets of experiments, using 1, 1.5 and 2 times the minimum heap size for each benchmark. All timing figures are taken as the median score of up to five runs.

In our preliminary experiments, we modified the Jikes RVM GC policy to force a collection immediately after each optcomp. However, we noticed that this actually causes a performance degradation. We changed the GC policy so that the VM checks to see if the heap usage has exceeded a certain threshold, immediately after each optcomp. If the threshold is exceeded, we force a collection. All the experiments below use this threshold-based eager GC-after-optcomp policy on modifed VMs. We arbitrarily chose to set the threshold to 0.9. A more detailed profiling study would assess various threshold values to determine an optimal heuristic.

Figure 14 shows the GC times for each benchmark. These initial experiments are run on the unmodified VM, so garbage collection only occurs when the standard VM heap usage monitoring code detects that the heap is nearly full. Then Figure 15 shows the relative difference in GC times between the unmodified and modified VMs. A negative score indicates a speedup in the modified VM, whereas a positive score indicates a slow-down. There is a clear variation in performance, with the most obvious improvements occuring for the minimum heap size, in general.

Finally we investigate how this eager GC-after-optcomp strategy affects the overall runtime of the programs. Reduction in GC time has a direct impact on

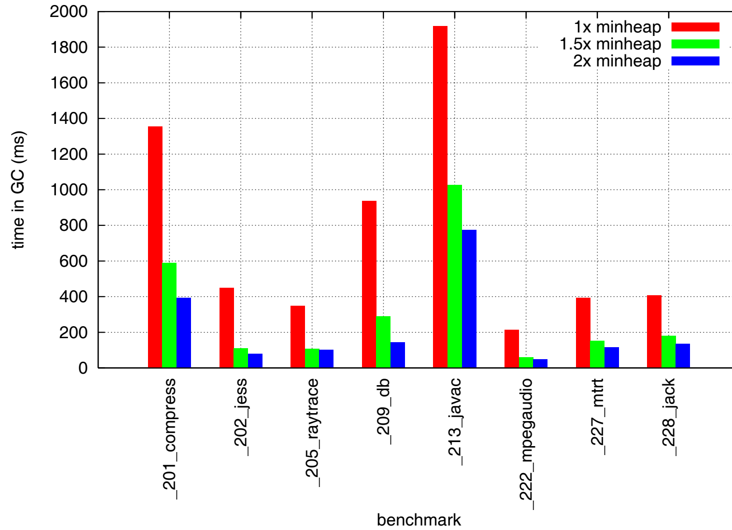

Fig. 14. GC times for different benchmarks before VM modification

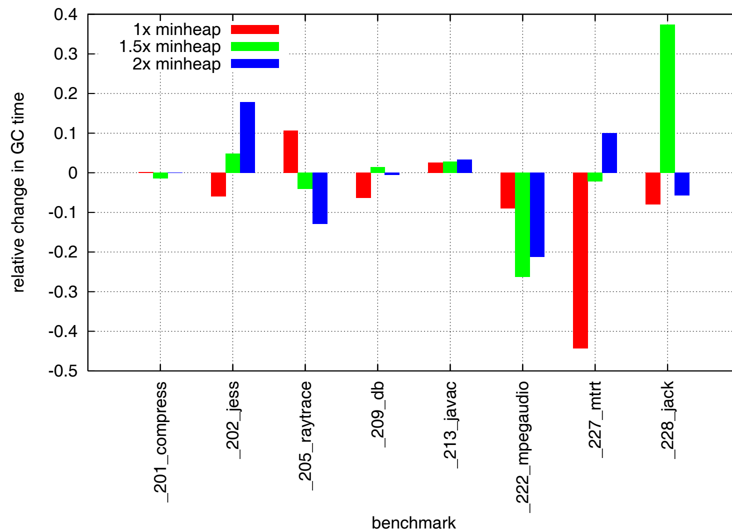

Fig. 15. Relative difference in GC times after VM modification, to force GC after optcomp

467 overall execution time, since GC time is included in the overall time. However,
468 there is also an indirect impact caused by improved GC. The execution time
469 of the benchmark code itself may be reduced due to secondary GC effects like
470 improved cache locality.

471 Figure 16 shows the overall execution times for each benchmark. These exper-
472 iments are run on the unmodified VM. From a comparison between Figures
473 14 and 16, it is clear to see that GC time is a small proportion of overall time.
474 Figure 17 shows the relative difference in overall times between the unmodified
475 and modified VMs. A negative score indicates a speedup in the modified VM,
476 whereas a positive score indicates a slow-down. There is a clear variation in
477 performance, with four significant improvements at the minimum heap size.

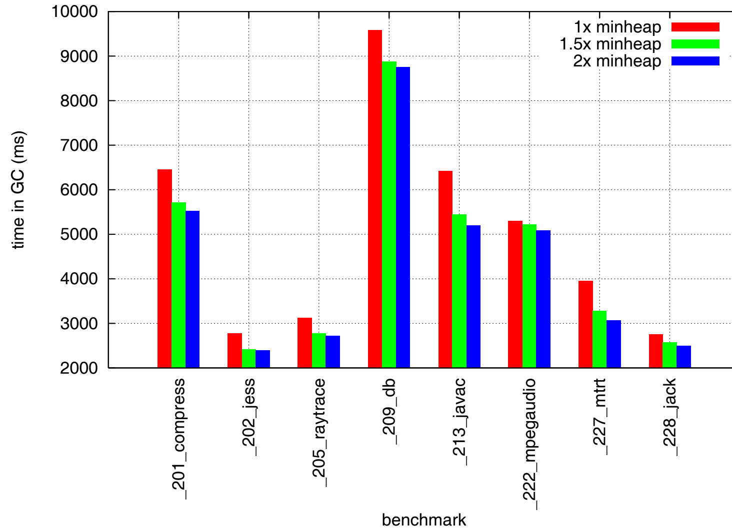

Fig. 16. Overall execution times for different benchmarks before VM modification

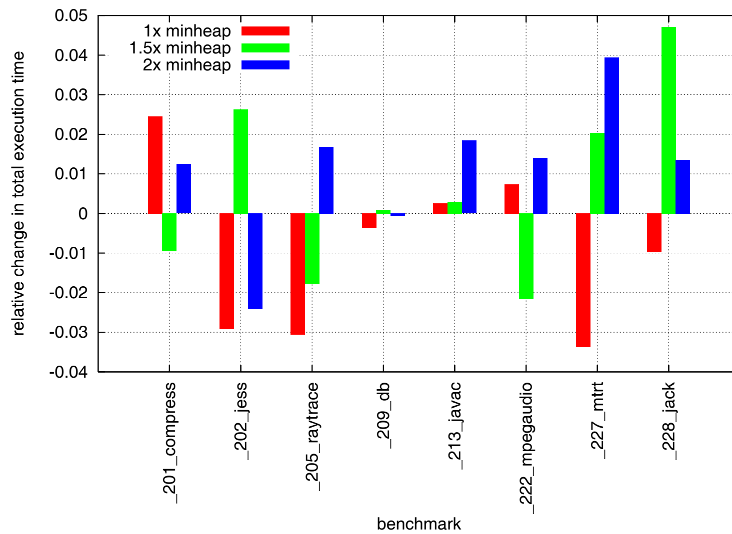

Fig. 17. Relative difference in overall execution times after VM modification, to force GC after optcomp

From this small study, we can see that it is sometimes advantageous to employ the eager GC-after-optcomp policy, although sometimes it does not improve performance. Perhaps this strategy should be an adaptive VM option rather than a hardwired choice, since it seems to depend on particular program characteristics. It should also depend on heap size configuration, growth policy and GC algorithm.

484 ## 6 Related Work

485 This paper is an extended version of previous research [21]. The current paper
486 improves on our earlier work in two ways:

487 (1) It provides a fuller treatment of the relationship between concepts and
488 aspects (Sections 1.1 and 2.4).
489 (2) It uses concept-based profiling to investigate scheduling policies for garbage
490 collection and optimizing compilation in Jikes RVM (Section 5).

491 ### *6.1 Visualization Systems*

492 Hauswirth et al. [22] introduce the discipline of *vertical profiling* which involves
493 monitoring events at all levels of abstraction (from hardware counters through
494 virtual machine state to user-defined application-specific debugging statistics).
495 Their system is built around Jikes RVM. It is able to correlate events at differ-
496 ent abstraction levels in dynamic execution traces. They present some interest-
497 ing case studies to explain performance anomalies in standard benchmarks.
498 Our work focuses on user-defined high-level concepts, and how source code
499 and dynamic execution traces are partitioned by concepts. Their work relies
500 more on event-based counters at all levels of abstraction in dynamic execution
501 traces.

502 GCspy [23] is an elegant visualization tool also incorporated with Jikes RVM.
503 It is an extremely flexible tool for visualizing heaps and garbage collection
504 behaviour. Our work examines processor utilization by source code concepts,
505 rather than heap utilization by source code mutators.

506 Sefika et al. [24] introduce *architecture-oriented visualization.* They recognise
507 that classes and methods are the base units of instrumentation and visualiza-
508 tion, but they state that higher-level aggregates (which we term concepts) are
509 more likely to be useful. They instrument methods in the memory management
510 system of an experimental operating system. The methods are grouped into
511 architectural units (concepts) and instrumentation is enabled or disabled for
512 each concept. This allows efficient partial instrumentation on a per-concept
513 basis, with a corresponding reduction in the dynamic trace data size. Our
514 instrumentation is better in that it can operate at a finer granularity than
515 method-level. However our instrumentation cannot be selectively disabled,
516 other than by re-assigning concepts to reduce the number of concept bound-
517 aries.

518 Sevitsky et al. [25] describe a tool for analysing performance of Java programs
519 using *execution slices.* An execution slice is a set of program elements that

520 a user specifies to belong to the same category—again, this is a disguised
521 concept. Their tool builds on the work of Jinsight [26] which creates a database
522 for a Java program execution trace. Whereas Jinsight only operates on typical
523 object-oriented structures like classes and methods, the tool by Sevitsky et al.
524 handles compound execution slices composed of multiple classes and methods.
525 They allow these execution slices to be selected manually or automatically.
526 The automatic selection process is based on ranges of attribute values—for
527 instance, method invocations may be characterized as slow, medium or fast
528 based on their execution times.

529 Eng [27] presents a system for representing static and dynamic analysis in-
530 formation in an XML document framework. All Java source code entities are
531 represented, and may be tagged with analysis results. This could be used
532 for static representation of concept information, but it is not clear how the
533 information could be extracted at runtime for the dynamic execution trace.

534 There are some Java visualization systems (for example, [28,29]) that instru-
535 ment user code at each method entry and exit point to provide extremely
536 detailed views of dynamic application behaviour. However these systems gen-
537 erate too much information to be useful for high-level comprehension purposes.
538 In addition, they do not capture JVM activity.

539 Other Java visualization research projects (for example, [30,31]) instrument
540 JVMs to dump out low-level dynamic execution information. However they
541 have no facility for dealing with higher-level concept information. In principle
542 it would be possible to reconstruct concept information from the lower-level
543 traces in a postprocessing stage, but this would cause unnecessarily complica-
544 tion, inefficiency and potential inaccuracy.

545 *6.2 Eager Garbage Collection Strategies*

546 Buytaert et al. [20] give a good overview of forced GC at potentially optimal
547 points. They have profiling runs to determine optimal GC points based on
548 heap usage statistics. They use the results of profiling to generate *hints* for the
549 GC subsystem regarding when to initiate a collection. Wilson and Moher [32]
550 append GC onto long computational program phases, to minimise GC pause
551 time in interactive programs. This is similar in our eager GC-after-optcomp
552 approach. Both optcomp and GC reduce interactivity, so it is beneficial to
553 combine these pauses whenever possible. Ding et al. [33] also exploit phase
554 behaviour. They force GC at the beginning of certain program phases, gain-
555 ing 40% execution time improvement. Their high-level phases are similar to
556 our notion of concepts. They assume that most heap-allocated data is dead at
557 phase transitions, and this assumption seems to be true for the single bench-

mark program they investigate. The behaviour of Jikes RVM is more variable and merits further investigation.

7 Concluding Remarks

This paper has explored the dynamic analysis of concept information. This is a promising research area that has received little previous attention. We have outlined different techniques for embedding concept information in Java source code and dynamic execution traces. We have presented case studies of concept visualization and profiling. This high-level presentation of concept information seems to be appealingly intuitive. We have demonstrated the utility of this approach by harnessing the interaction between runtime compilation and garbage collection in the Jikes RVM adaptive runtime environment.

Until now, concepts have been a compile-time feature. They have been used for static analysis and program comprehension. The current work drives concept information through the compilation process from source code to dynamic execution trace, and makes 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 [34].

During the course of this research project, we conceived a novel process which we term *feedback-directed concept assignment*. This involves: (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 fact, this is how we reached the decision to examine both baseline and optimizing compilers separately in Section 4.1 rather than having a single compilation concept. We noticed that the single compilation concept (incorporating the activities of both baseline and optimizing compilers) was large, and did not correlate as well with the garbage collection concept. Once we split this concept into two, we observed that garbage collection follows optimizing compilation rather than baseline.

The process of feedback-directed compilation could be partially automated, given sufficient tool support. We envisage a system that allows users to specify the granularity of concepts in terms of source code (average number of lines per concept) or execution profile (average execution time percentage per concept) or both. The tool would process an initial concept assignment, execute the concept-annotated program and determine whether the user-specified requirements are met. If so, the tool indicates success. If not, the tool suggests a possible splitting of concepts, which the user has to approve or modify, then the tool reassesses the concept assignment.

With regard to future work, we should incorporate the analyses and visualizations presented in this paper 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.

8 Acknowledgements

The first author is employed as a research associate on the Jamaica project, which is funded by the EPSRC Portfolio Award GR/S61270/01.

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 *SCP* and *PPPJ '06* referees, together with many *PPPJ '06* conference attendees, for their thoughtful and helpful feedback on various iterations of this paper.

References

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

[2] N. Gold, K. Bennett, Hypothesis-based concept assignment in software maintenance, IEE Software 149 (4) (2002) 103–110.

[3] N. Gold, Hypothesis-based concept assignment to support software maintenance, in: Proceedings of the IEEE International Conference on Software Maintenance, 2001, pp. 545–548.

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

[5] J. Weidendorfer, Kcachegrind profiling visualization, see kcachegrind.sourceforge.net for details (2005).

628 [6] N. Nethercote, J. Seward, Valgrind: A program supervision framework,
629 Electronic Notes in Theoretical Computer Science 89 (2) (2003) 1–23.

630 [7] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby,
631 S. Fink, D. Grove, M. Hind, et al., The Jikes research virtual machine project:
632 building an open-source research community., IBM Systems Journal 44 (2)
633 (2005) 399–417.

634 [8] A. Azevedo, A. Nicolau, J. Hummel, Java annotation-aware just-in-time (AJIT)
635 compilation system, in: Proceedings of the ACM Conference on Java Grande,
636 1999, pp. 142–151.

637 [9] R. H. F. Brown, R. N. Horspool, Object-specific redundancy elimination
638 techniques, in: Proceedings of Workshop on Implementation, Compilation,
639 Optimization of Object-Oriented Languages, Programs and Systems, 2006.

640 [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier,
641 J. Irwin, Aspect-oriented programming., in: Proceedings of the 11th European
642 Conference on Object-Oriented Programming, 1997, pp. 220–242.
643 URL
644 `http://link.springer.de/link/service/series/0558/bibs/1241/12410220.htm`

645 [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold,
646 An overview of AspectJ, in: Proceedings of the 15th European Conference on
647 Object-Oriented Programming, 2001, pp. 327–353.
648 URL
649 `http://link.springer.de/link/service/series/0558/bibs/2072/20720327.htm`

650 [12] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented
651 programming, in: Proceedings of the 1st International Conference on Aspect-
652 Oriented Software Development, 2002, pp. 141–147.

653 [13] M. H. Lipasti, C. B. Wilkerson, J. P. Shen, Value locality and load value
654 prediction, in: Proceedings of the 7th International Conference on Architectural
655 Support for Programming Languages and Operating Systems, 1996, pp. 138–
656 147.

657 [14] Y. Sazeides, J. E. Smith, The predictability of data values, in: Proceedings of
658 the 30th ACM/IEEE International Symposium on Microarchitecture, 1997, pp.
659 248–258.

660 [15] J. Singer, G. Brown, Return value prediction meets information theory,
661 Electronic Notes in Theoretical Computer Science 164 (3) (2006) 137–151.

662 [16] E. Duesterwald, C. Cascaval, S. Dwarkadas, Characterizing and predicting
663 program behavior and its variability, in: Proceedings of the 12th International
664 Conference on Parallel Architectures and Compilation Techniques, 2003, pp.
665 220–231.

666 [17] A. Georges, D. Buytaert, L. Eeckhout, K. De Bosschere, Method-level
667 phase behavior in Java workloads, Proceedings of the 19th ACM SIGPLAN
668 Conference on Object-Oriented Programming, Systems, Languages, and
669 Applications (2004) 270–287.

670 [18] P. Nagpurkar, C. Krintz, Visualization and analysis of phased behavior in Java
671 programs, in: Proceedings of the 3rd International Symposium on Principles
672 and Practice of Programming in Java, 2004, pp. 27–33.

673 [19] J. Singer, Concept assignment as a debugging technique for code generators, in:
674 Proceedings of the 5th IEEE International Workshop on Source Code Analysis
675 and Manipulation, 2005, pp. 75–84.

676 [20] D. Buytaert, K. Venstermans, L. Eeckhout, K. D. Bosschere, GCH: Hints for
677 triggering garbage collection, Transactions on High-Performance Embedded
678 Architectures and Compilers 1 (1) (2006) 52–72.

679 [21] J. Singer, C. Kirkham, Dynamic analysis of program concepts in Java, in:
680 Proceedings of the 4th International Symposium on Principles and Practice
681 of Programming in Java, 2006, pp. 31–39.

682 [22] M. Hauswirth, P. F. Sweeney, A. Diwan, M. Hind, Vertical profiling:
683 understanding the behavior of object-oriented applications, in: Proceedings
684 of the 19th ACM SIGPLAN Conference on Object-Oriented Programming,
685 Systems, Languages, and Applications, 2004, pp. 251–269.

686 [23] T. Printezis, R. Jones, GCspy: an adaptable heap visualisation framework,
687 in: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented
688 Programming, Systems, Languages, and Applications, 2002, pp. 343–358.

689 [24] M. Sefika, A. Sane, R. H. Campbell, Architecture-oriented visualization, in:
690 Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented
691 Programming, Systems, Languages, and Applications, 1996, pp. 389–405.

692 [25] G. Sevitsky, W. D. Pauw, R. Konuru, An information exploration tool for
693 performance analysis of Java programs, in: Proceedings of TOOLS Europe
694 Conference, 2001.

695 [26] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang,
696 Visualizing the execution of Java programs, in: Software Visualization, Vol.
697 2269 of Lecture Notes in Computer Science, 2002, pp. 151–162.

698 [27] D. Eng, Combining static and dynamic data in code visualization, in:
699 Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program
700 Analysis for Software Tools and Engineering, 2002, pp. 43–50.

701 [28] S. P. Reiss, Visualizing Java in action, in: Proceedings of the IEEE Conference
702 on Software Visualization, pp. 123–132.

703 [29] S. P. Reiss, M. Renieris, JOVE: Java as it happens, in: Proceedings of the ACM
704 Symposium on Software Visualization, 2005, pp. 115–124.

705 [30] P. Dourish, J. Byttner, A visual virtual machine for Java programs: exploration
706 and early experiences, in: Proceedings of the ICDMS Workshop on Visual
707 Computing, 2002.

708 [31] M. Golm, C. Wawersich, J. Baumann, M. Felser, J. Kleinöder, Understanding
709 the performance of the Java operating system JX using visualization techniques,
710 in: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande,
711 2002, p. 230.

712 [32] P. R. Wilson, T. G. Moher, Design of the opportunistic garbage collector, in:
713 Conference Proceedings on Object-Oriented Programming Systems, Languages
714 and Applications, 1989, pp. 23–35.

715 [33] C. Ding, C. Zhang, X. Shen, M. Ogihara, Gated memory control for memory
716 monitoring, leak detection and garbage collection, in: Proceedings of the
717 Workshop on Memory System Performance, 2005, pp. 62–67.

718 [34] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly
719 language, ACM Transactions on Programming Languages and Systems 21 (3)
720 (1999) 527–568.