

Visualized Adaptive Runtime Subsystems

Jeremy Singer*
University of Manchester, UK

Chris Kirkham†
University of Manchester, UK

Abstract

Virtual execution platforms contain many runtime subsystems that are invoked on-demand as user code executes. We have instrumented an open-source JVM to dump out information on these runtime subsystems, in order to examine how they are used. The monitored components are known as *visualized adaptive runtime subsystems*. The visualizations in this poster demonstrate five interesting properties of such runtime subsystems.

1 Introduction

Virtual execution platforms such as the Java virtual machine (JVM) are now ubiquitous. They are deployed throughout the spectrum of computing devices, from high-end servers to mobile phones.

State-of-the-art JVM implementations, like Sun's HotSpot and IBM's Jikes RVM [Alpern et al. 2005], feature *adaptive runtime subsystems*. Various runtime services are invoked on-demand as the user code executes. These include (i) just-in-time (JIT) compilation; (ii) garbage collection; and (iii) thread scheduling. The research presented in this poster focuses on JIT compilation and garbage collection.

These adaptive runtime subsystems execute 'under-the-hood'. 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 services are occurring, even as the code executes. They are effectively invisible, from both a static and a dynamic point of view. *Visualized* adaptive runtime subsystems (VARS) are no longer invisible. This research project aims to enable both programmers and users to see the effects that the adaptive runtime subsystems have on their code. Other Java visualization systems such as JOVE [Reiss and Renieris 2005] do not make a distinction between user and VM code. Basically, we are only interested in VM behaviour, and how to optimize it.

We study VARS behaviour by execution of Java code within a specially instrumented JVM (IBM's Jikes RVM). This VM is adapted so that it dumps out profiling information for the amount of time spent in the different adaptive runtime subsystems. The instrumentation required to produce runtime profiling information can be inserted using techniques like static aspect weaving and concept assignment [Singer and Kirkham 2006]. After an execution run with the instrumented VM, we postprocess the trace file to produce the visualization of VARS behaviour. This work concentrates on compilation and garbage collection VARS. Note that compilation and

garbage collection are the dominant VARS in terms of execution time. Compilation divides into subcategories of baseline and optimizing. The baseline compiler is a simple macro-expansion system that is quick to run but generates inefficient code. On the other hand the optimizing compiler is a sophisticated SSA-based program analysis system. It is very slow to run but generates highly optimized code. Generally the optimizing compiler should be used for 'hot' methods that are frequently executed. In the visualization examples, baseline compilation is marked as red and optimizing as blue. Garbage collection is green. All other code is yellow. This is mostly user application code, although it does include some other runtime overhead which we have not measured. Figure 1 gives a legend for the subsequent visualization examples.

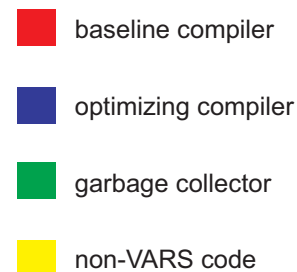


Figure 1: Key for VARS types

All the visualizations in this poster are runs from the `_201_compress` program from the SPEC JVM98 benchmark suite.

In each visualization, time increases horizontally. The coloured blocks indicate which kind of code is executed at each time stage. All our case studies are conducted for single threaded programs and VM. When multiple threads become involved, some style of Gantt chart visualization will be necessary. Note that all presented traces are necessarily truncated for space reasons.

2 VARS Properties

This section presents five properties of VARS that our visualizations clearly demonstrate. The visualizations have enabled us to gain a better understanding of the nature of VARS.

2.1 They are pervasive

VARS code is invoked throughout the entire program lifetime. Figure 2 illustrates this point. It is noticeable that the density of VARS code in relation to user code changes over time. It appears that VARS code is more frequent near the beginning of execution. This is caused by the JIT compiler, which compiles every method immediately before it is executed for the first time. Later compilation generally involves selective optimizing recompilation of 'hot' methods.

*e-mail: jsinger@cs.man.ac.uk

†e-mail: chris@cs.man.ac.uk



Figure 2: Pervasive nature of VARS execution



Figure 3: Varying nature of VARS execution



Figure 4: Periodic nature of VARS execution

2.2 They occupy significant runtime

Visualizations like Figure 2 show that VARS code occupies a significant proportion of total execution time. Effectively a VM's execution time is shared between application code and VARS code. For the long-running benchmark program used in this study, around 90% of execution time is spent in user code and 10% in VARS code. The VARS time should be more significant for shorter programs. It is also interesting to discover how the VARS execution time is shared between the various VARS components. For the benchmark used in this study, the VM spends at least twice as long in compilation as in garbage collection.

2.3 They vary with VM configuration

Modern adaptive runtimes are highly configurable. It is possible to specify policies and parameters for all VARS. These have a major impact on system performance. Previously it was not possible to see exactly how varying configurations changed VARS behaviour. Now our VARS visualizations make this task straightforward. Figure 3 shows two runs of the same program, but with different VM configurations. The top line uses the default VM compilation policy, which at first compiles all methods using the baseline compiler then recompiles 'hot' methods with the more expensive optimizing compiler. The bottom line uses a modified compilation policy, which initially uses the optimizing compiler rather than the baseline compiler, as much as possible. Note that Jikes RVM insists that some methods must be compiled at baseline level.

2.4 They interact with each other

The VARS are not entirely independent. They affect one another in subtle ways. For instance in Figure 3 above, it is clear to see that greater use of the optimizing compiler causes increased garbage collection activity. This is because the optimizing compiler generates many program representations and temporary data structures as it compiles methods, thus filling up heap space. Note that there is a shared heap between VARS code and user code.

2.5 They exhibit periodic behaviour

It is commonly acknowledged that programs go through different phases and exhibit periodic patterns in various architectural and high-level metrics. Figure 4 demonstrates that VARS code may

be periodic too. In this execution trace, the program is memory-intensive and the VM heap has been restricted to a small size. Thus the garbage collector has to run frequently, and if the memory load is constant over time, then the garbage collector will run periodically.

3 Conclusions

VARS are interesting. They should be handled separately from user code since they are VM-specific and dependent. VARS should be optimizable directly without reference to user code. In general, VARS are not well investigated or understood. This visualization project attempts to study and understand the nature and behaviour of VARS. The properties learnt should prove interesting by themselves, but may also have practical applications for VM engineering. For instance, the VARS periodicity information in Section 2.5 could be useful to schedule collection slightly ahead-of-time at a more convenient point in execution.

Future work involves adding instrumentation support for other VARS such as thread scheduling. Our more long-term goal is to refactor Jikes RVM source code to set up a unified adaptive runtime management system that effectively provides a single programmatic interface for adaptive runtime subsystems. Of course, such an interface would make instrumentation for profiling and visualization much simpler too.

References

- ALPERN, B., AUGART, S., BLACKBURN, S. M., BUTRICO, M., COCCHI, A., CHENG, P., DOLBY, J., FINK, S., GROVE, D., HIND, M., MCKINLEY, K. S., MERGEN, M., MOSS, J. E. B., NGO, T., SARKAR, V., AND TRAPP, M. 2005. The Jikes Research Virtual Machine Project: Building an Open Source Research Community. *IBM Systems Journal* 44, 2, 1–19.
- REISS, S. P., AND RENIERIS, M. 2005. JOVE: Java as it Happens. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, 115–124.
- SINGER, J., AND KIRKHAM, C. 2006. Dynamic Analysis of Program Concepts in Java. In *Proceedings of the International Conference on Principles and Practices of Programming in Java*. To appear.