McGill University
School of Computer Science
Sable Research Group

Problems in Objectively Quantifying Benchmarks using Dynamic Metrics

Sable Technical Report No. 2003-6

Bruno Dufour, Laurie Hendren and Clark Verbrugge

October 22, 2003

Contents

Abstract

Basic problems encountered when trying to accurately and reasonably measure dynamic properties of a program are discussed. These include problems in determining and assessing specific, desirable metric qualities that may be perturbed by subtle and unexpected program behaviour, as well as technical limitations on data collection. Some interpreter or Java-specific problems are examined, and the general issue of how to present metrics is also discussed.

1 Introduction

Runtime system implementation, compiler optimization, software understanding, and many other areas of program analysis need accurate information as to how a program behaves. Measurements of pertinent quantities or qualities can then be used to guide implementation, understanding and further exploration. Such measurements are most easily acquired by statically examining program code; however, it is also possible to examine, and measure, the dynamic or runtime behaviour of a program for further insight.

In this paper we describe some of the basic problems encountered when trying to accurately and reasonably measure dynamic properties of a program. Some issues in dynamic analysis are of course well-known: the input to a program under analysis must naturally result in behaviour that is representative of the runtime behaviour under any projected input. Unfortunately, even given this caveat there are many problems associated with establishing useful measurements, both in design and implementation. These include (non-obvious) decision problems for ensuring the measured quality or quantity actually does correspond to whatever property one is trying to examine (particularly in the case of Java), but also problems and conflicts in ensuring comparability of metric values between programs or execution runs. We characterize these general issues as ones of ensuring *robustness*, that a metric does not change unless something relevant changes, and also ensuring that a metric is *discriminating*, and so does indeed vary when a relevant quality is changed. Problems of ensuring the measured values and measuring technique is "sensible" are often exacerbated by technical concerns that constrain what can feasibly be measured. In order to establish actual, general metrics that describe program behaviour these concerns must all be addressed. The metrics themselves are described in more details in [5].

1.1 Roadmap

In the remaining sections we describe some basic dynamic metric properties, and how various issues affect metric collection, determination, and analysis. Section 2 begins this process by a discussion of the technical issues that limit the possible metrics available. Problems related to designing specific metric qualities are explored in section 3. An issue particularly important to languages such as Java with large libraries or interpreter environments is the effect of the surrounding system or runtime code on the program; this is discussed in section 4. Finally, we consider the problem of building a program summary through a minimal set of metrics in section 5. Section 6 gives some related work, and future work and conclusions are given in sections 7 and 8 respectively.

2 Collecting Profile Data

Because the availability of data ultimately determines which measurements can or cannot be obtained, it is important to examine data collection as the first step in designing measures. There are two main categories of problems that can limit data collection: data size and profiler implementation.

Complete execution traces are well-known to occupy a very large amount of space even for relatively small benchmark program executions. When recording frequent events, such as executed bytecode instructions, the amount of disk space may well be insufficient to store the uncompressed trace file. This obviously limits the amount of information that can be recorded unless a means of reducing the trace size is found. Fortunately, executed instructions are highly predictable events. Also, memory addresses are frequently found in trace files and often exhibit good locality. As a result, execution traces generally respond very well to ordinary compression schemes. Better tailored compression algorithms can even reduce the trace sizes even further, as the STEP framework[2]. Our system uses an instruction predictor to effectively reduce the size of the trace.

The way in which the profiler gathers execution data is however the main limiting factor in designing dynamic metrics. Our system uses the Java Virtual Machine Profiler Interface (JVMPI) to receive events via a callback mechanism. Several issues with the JVMPI specification make certain dynamic metrics more difficult or even impossible to compute.

Because the JVMPI does not provide any mechanism for inspecting the execution stack, the only way to keep track of object references at the level of executed instructions is to simulate the entire execution. This solution imposes far too much overhead for our purposes. Also, the JVMPI specification states that only instruction offsets only are reported. Thus, obtaining the associated instruction opcode requires class file parsing.

It is also a well-known fact amongst the JVMPI community that some events may be skipped during the VM startup phase. This is a major issue when collecting complete traces, and addressing the problem is a non-trivial endeavour. First, the agent has to keep an internal representation of the data so as to detect such situations. Then, missing events have to be explicitly requested by the agent. However, such requests have to be placed in accordance to the VM's internal state in order to avoid a crash. Producing a correct JVMPI agent is thus a complex task.

3 Desired Qualities

When designing new measures, one must ensure that they adequately capture the aspect of software behaviour that they are intended to measure. There are two important aspects to the representativeness of a measure: it has to be both *robust* and *discriminating*.

3.1 Robust Measures

The concept of robustness is intended to express the intuition that measurements should be left relatively unaffected by insignificant changes. A dynamic metric is *robust* if a "small" change in program behaviour results in a correspondingly small change in the measured value. Because it is difficult to precisely determine what a "small" change is, it is also difficult to achieve robustness when quantifying program behaviour. Robustness will be examined in the context of a program optimization consisting of inlining virtual method invocations for the JOlden `Voronoi` benchmark.

The most often reported measurements in the literature take the form of simple absolute counts. However, the character of the element being counted has a very significant effect of the robustness of a metric. For example, in the case of the number of executed bytecode instructions, the optimization has a dramatic effect on the measured values: 445×10^6 vs. 288×10^6, a reduction of 35%. However, a more robust metric would be the number of executed call sites, which is intuitively a much more robust value. For the `Voronoi` example, it went from 680 to 561, a reduction of only 17.5%.

While absolute measures are very effective in comparing different versions of the *same* benchmark, they are not very indicative when comparing different benchmarks altogether. Relative measures are much more suited to this particular task because they inherently possess some form of standardization. The robustness discussion is also relevant to relative measures. In the same `Voronoi` example, measuring the number of method invocations per 1000 executed bytecode instructions (kilobytecode, or kbc), the difference between the two versions of the benchmark is again very significant, dropping from 111 invocations per kbc to a mere 11 (a reduction of over 90%). This metric is thus not very robust. Applying the same reasoning as for the absolute case, it is possible to compute the density of call sites instead as the total number of touched call sites per 1000 touched instructions. This is intuitively a much more robust measure, which is supported by the experimental data: the density only dropped from 65 to 51, a reduction of only 21%.

It is interesting to note that the difference in the measurements in both cases correspond to two different aspects of program optimization. The more robust metrics relate to the amount of *opportunities* in a program for the optimization in question, whereas the less robust metrics relate to the overall importance of the optimization. By measuring the differences in measurements between the two versions of the benchmark, it is easy to ascertain the effect of the effect of the optimization.

3.2 Discriminating Measures

A measure is discriminating if a "large" change in program behaviour results in a correspondingly large change in the obtained measurement. Intuitively, representative measures should be both robust and discriminating. Consider a measure which would always map a character to the value 1. Obviously, this measure would be very robust, but not at all discriminating. It is equally easy to design a very (although not perfectly) discriminating metric: the number of executed instructions in a program trace has been previously shown to be such an example.

While in general it is easy to think of measures which are equally robust and discriminating, it is much more difficult to design such measures when working with software behaviour. As a result, most dynamic metrics (if not all of them) will show a bias towards one or the other.

4 Library and Startup Effects

A different challenge in quantifying application behaviour relies in the fact that it is not clear how to differentiate the application from its runtime support environment, making it difficult to exactly identify what to measure. There are two main categories of events that must be taken into consideration when designing dynamic metrics: startup and runtime services.

The importance of startup code varies depending on the language and its implementation. In the case of Java, startup is a large program. For instance, the empty program (which only returns from its main method) executes over 534000 bytecode instructions before terminating, excluding native code (JNI). The presence of startup events in measurements can significantly distort measurements on programs which have a short execution. However, this distorting effect can be amortized over long executions.

Moreover, the notion of startup is ill-defined, particularly so in Java. Simply defining startup as any code executed prior to the first instruction in the main method being executed leaves out arguably non-startup code, such as static initializers for the application classes. Also, in the case of Java, garbage collection, class loading, etc. are recurring events that occur "spontaneously" from time to time. Because they occur after the execution of the main method, they would not be excluded as being part of the startup code but are arguably not part of the application itself.

Also, the effect of the runtime libraries is also a concern when designing measures, since it can significantly distort measurements. For example, the `MatrixMult` benchmark features a total around 3 field accesses per 1000 executed bytecodes when only the benchmark classes are taken into account. However, including all of the information leads to the same density jumping to around 75 field accesses per 100 executed bytecodes. However, simply excluding all library calls from a measure might not be fair. For example, a program which makes intensive use of data structures may spend a considerable amount of time execution library code. In such a case, completely excluding libraries is intuitively not desirable. One would rather include calls to the library code which originated from application code, in order to avoid creating an unfair situation with respect to programs which are implemented using their own specialized version of the library classes.

Also, the differences between the different library versions are likely to affect measurements. For example, the amount of observed polymorphism found in the SPEC `_228_jack` benchmark can be increased by 8% simply by changing from a 1.3 JVM to a 1.4 JVM. This change is due to the addition of the `EmptyEnumerator` class in the 1.4 JVM. The evolution of the runtime library is thus reflected on the measurements while the benchmark program itself did not change at all, which is often not desired.

Moreover, it is also difficult to guarantee that the measures are VM-independent and platform-independent, even for Java programs. For example, certain on-the-fly optimizations performed a JVM or a JIT compiler can affect perturb the results of a measure by changing the executed code. For example, even in interpreted mode, the Hotspot VM is known to skip the execution of empty methods (i.e. the events are not sent to the JVMPI agent, thereby reducing the number of recorded `return` instructions). More serious optimizations would include transformations such as inlining, which are very common to current virtual machines.

Even measures that depend on the implementation of the virtual machines are not platform independent. For example, measuring the number of bytes of memory that are allocated during a program's execution is dependent on the header size of the object, which is VM-dependent. Moreover, the amount of utilized memory can be internally affected by alignment issues, thereby making such a measure platform-dependent.

5 Designing a Concise Set of Measures

One of our original goals when starting to define dynamic metrics was to obtain a concise set of measures that would provide a good overview of the runtime behaviour of a program. However, due to the large number of decisions that can affect a measure, it is not likely that such a set can be accurately constructed. It is much easier to come up with a concise set of measures in the context of a specific problem or issue than for a general, multi-purpose setting.

Our current goals involve more of a gradual refinement process which starts with a relatively small set of dynamic metrics. Interesting numbers will lead to further investigation of the benchmark program using a refined and more precise set of metrics that provide more details concerning the underlying behaviour of the program. We currently use a web interface allowing user-defined queries on a database to iteratively select a concise and informative set of dynamic metrics that are relevant to a particular problem. If further investigation is needed, the next logical step is to use visualization techniques. Visualization can be used to view the evolution of particular metrics over the entire execution (*continuous* metrics), providing more accurate information than a single summary value. Traditional software visualization can also provide more detailed information when needed.

6 Related Work

There is a significant body of literature devoted to static metrics; e.g., Fenton and Pfleeger's book [6]. Techniques for dynamically analyzing programs, however, are less well developed. Specific studies of dynamic behaviour have been made: Dieckmann and Hölzle study the allocation behaviour of SPECjvm98 benchmarks [4], Daly et al. have looked at the Java Grande benchmarks [3], Shuf et. al have examined the memory behaviour of Java workloads, particularly with respect to evaluating the potential for certain compiler optimizations [10]. Specific approaches such as these focus on particular techniques or understanding programs through absolute measurements rather than establishing dynamic metrics with general properties.

More general dynamic metrics are given by Yacoub, Ammar and Robinson, who discuss a wide variety of metrics meant to evaluate object-oriented design [12]. They give precise definitions, and argue for the utility and applicability of dynamic metrics. Although these definitions are quite useful, they do not establish generally-desirable metric properties.

We have explored dynamic metrics analyses through the use of a specific, custom metric analysis tool, *J. Other, dynamic analysis tools exist; Aggarwal et al, for instance, present a system for computing dynamic metrics related to finding the most frequently executed modules, applied to C programs [1]. "Gadget" is a tool used for discovering the dynamic relations between objects in a Java program [8], "Shimba" is a trace-based reverse-engineering tool [11], and "Caffeine" is an offline trace analyzer with a Prolog query engine [9]. All of these Java tools have to address similar performance problems due to large traces and slow data collection.

7 Future Work

We are currently investigating ways to improve our dynamic metric analysis framework, *J. Two main areas are considered. First, we aim at achieving better compression of the traces. Better instruction predictors and integration with the STEP framework are planned. Also, we are continuously testing and adding new dynamic metrics to our set. So far, we have ideas for branching and looping metrics that could be implemented using the current framework.

Because of the limitations of our current data gathering process, we are also planning to instrument an open-source Java Virtual Machine, SableVM [7], to output traces that could be processed by *J. This would allow us to obtain dynamic information that the JVMPI framework cannot provide, such as object reference resolution. The additional data will of course lead to the development of new dynamic metrics. We currently plan to design new pointer-related metrics, and some more advanced concurrency and locking metrics based on the additional information. We are also investigating the possibility of designing dynamic versions of the classic static software metrics, such as cohesion and coupling measures.

On the theoretical front, we looking at the desired qualities of dynamic metrics, trying to identify a complete set. In particular, we are continuing our experimentation with robustness and discriminating qualities. A precise formalization

of the concepts presented in this paper is still required.

We have tried to identify a set dynamic metrics that we feel covers a wide range of applications for compiler developers. As far as we can tell, our metrics potentially have other applications. We however always welcome user feedback regarding suggestions of new dynamic metrics that would be more suited to other fields of research.

8 Conclusions

We have described and discussed several properties or behaviours that affect dynamic metric calculation and interpretation. Issues of robustness (ensuring metrics are stable with respect to conceptually irrelevant behaviours), being discriminatory (ensuring metrics change when relevant quantities change), and the large impact of library and startup effects in certain environments establishes non-trivial limitations on what metrics one may want, and how they may be used. Once metrics are gathered, they need to be aggregated and presented in an accessible and user-friendly fashion; this itself is a difficult task if the user is not to be overwhelmed, but still needs accurate information. We are continuing to explore and develop these ideas.

Acknowledgments

This work was supported, in part, by NSERC and FQRNT.

References

[1] K.K. Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. A dynamic software metric and debugging tool. *ACM SIGSOFT Software Engineering Notes*, 28(2):1–4, March 2003.

[2] Rhodes H. F. Brown. STEP: A framework for the efficient encoding of general trace data. Master's thesis, McGill University, Montréal, Québec, Canada, 2003.

[3] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 106–115. ACM Press, 2001.

[4] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of ECOOP 1999, LNCS 1628*, pages 92–115, 1999.

[5] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003. (To appear).

[6] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach*. 1997.

[7] Étienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montréal, Québec, Canada, 2002.

[8] Juan Gargiulo and Spiros Mancoridis. Gadget: A tool for extracting the dynamic structure of Java programs. In *Proceedings of the 2001 International Conference on Software Engineering and Knowledge Engineering (SEKE'01),*, pages 244–251, Buenos Aires, Argentina, June 2001.

[9] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No java without caffeine: A tool for dynamic analysis of Java programs. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 117–128, Edinburgh, UK, September 2002.

[10] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, 2001.

[11] Tarja Systä. Understanding the behavior of Java programs. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 214–223, Brisbane, Australia, November 2000.

[12] S.M. Yacoub, H.H. Ammar, and T. Robinson. Dynamic Metrics for Object Oriented Designs. In *Sixth IEEE International Symposium on Software Metrics*, pages 50–61, November 1999.