Dynamic Metrics for Java*

Bruno Dufour, Karel Driesen, Laurie Hendren and Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, CANADA H3A 2A7
[bdufou1,karel,hendren,clump]@cs.mcgill.ca

ABSTRACT

In order to perform meaningful experiments in optimizing compilation and run-time system design, researchers usually rely on a suite of benchmark programs of interest to the optimization technique under consideration. Programs are described as *numeric*, *memory-intensive*, *concurrent*, or *object-oriented*, based on a qualitative appraisal, in some cases with little justification. We believe it is beneficial to quantify the behaviour of programs with a concise and precisely defined set of metrics, in order to make these intuitive notions of program behaviour more concrete and subject to experimental validation. We therefore define and measure a set of unambiguous, dynamic, robust and architecture-independent metrics that can be used to categorize programs according to their dynamic behaviour in five areas: size, data structure, memory use, concurrency, and polymorphism. A framework computing some of these metrics for Java programs is presented along with specific results demonstrating how to use metric data to understand a program's behaviour, and both guide and evaluate compiler optimizations.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Experimentation, Languages, Measurement, Performance, Standardization

Keywords

Dynamic Metrics, Software Metrics, Program Analysis, Java, Profiling, Execution Traces, Optimization

1. INTRODUCTION

Understanding the dynamic behaviour of programs is one important aspect in developing effective new strategies for optimizing compilers and runtime systems. Research papers in these areas often say that a particular technique is aimed at programs that are *numeric*, *loop-intensive*, *pointer-intensive*, *memory-intensive*, *object-oriented*, *concurrent*, and so on. However, there appears to be no well-established standard way of determining if a program fits into any of these categories. The goal of the work presented in this paper was to develop dynamic metrics that can be used to measure relevant runtime properties of programs, with the ultimate goal of establishing some standard metrics that could be used for quantitative analysis of benchmark programs in compiler research.

To be useful, dynamic metrics should provide a concise, yet informative, summary of different aspects of the *dynamic* behaviour of programs. By *concise*, we mean that a small number of numeric values should be enough to summarize the behaviour. For example, a complete profile of a program would not be a concise metric, whereas the fact that 90% of program execution is accounted for by 5% of the methods is a concise metric. By *informative*, we mean that the metric must measure some program characteristic of relevance to compiler or runtime developers, and the metric should differentiate between programs with different behaviours. For example, computing the ratio of number of lines of code to number of lines of comments does not capture anything about program behaviour, and is not an informative metric, for our purposes.

In order to get a good overview of program characteristics of interest to compiler and runtime systems developers, we first studied the typical dynamic characteristics reported in papers presented over the last several years in the key conferences in the area. Although we did find many interesting experiments that suggest potential dynamic metrics, we also found that many summaries of benchmarks and results focused on static program measurements (#lines of code, # of methods, # loops transformed, # methods that are inlinable, # number of locations pointed to at dereference sites, and so on). Based on our own experiences, we suspect that this focus on static, rather than dynamic, metrics is at least partly because the static metrics are much easier to compute. Thus, one important goal of this paper is to provide both a methodology to compute the dynamic metrics and a database of dynamic metrics for commonly used benchmarks.

In our development of dynamic metrics we also discovered that different program behaviours need to be summarized in different ways. For example, if one is measuring the behaviour of virtual method calls in an object-oriented language like Java, one summary might be the average number of target methods per virtual call site. However, compiler developers are usually most interested in the special cases where the virtual call is monomorphic (good for inlining) or perhaps corresponds to a very small number of target methods (good for specialization). Thus, a much more relevant metric might be what percentage of virtual calls correspond to 1, 2, or more than 2, targets. Thus, in this paper we define three basic

*This work was supported, in part, by NSERC, FCAR and McGill FGSR. Special thanks to Tobias Simon for the web pages and Marc-André Dufour for designing the icons for the metrics.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-712-5/03/0010 ...$5.00.

ways of reporting dynamic metrics, *values*, *bins*, and *percentiles*, and we suggest which type is most appropriate for each situation.

Based on our overview of the literature, and our own experiences and requirements, we have developed many dynamic metrics, grouped under five categories: (1) size and structure of programs, (2) data structures, (3) polymorphism, (4) memory and (5) concurrency. In this paper we provide definitions for metrics in each category, and we provide specific examples of computing some of these metrics on well known benchmarks.

It is of course rather critical that the metrics we define do capture in a quantitative sense the qualitative aspects of a program. To demonstrate utility and relevance of the metrics we define, we show how one might analyze and compare several benchmarks and relate the numerical results back to their qualitative and relative behaviour. The result is a quantitatively justifiable description and understanding of a benchmark, one that often provides insights that would not be apparent without a very detailed understanding of the benchmark (and runtime library) activity.

Further evidence of the value and potential use of our metrics is given by describing how one can use metric values to both guide and evaluate actual compiler optimizations. Results in this area are very encouraging; metric data not only corresponds well to the expected optimization, but can also reveal interesting and surprising optimization behaviour and interactions. Use of metrics thus simplifies and validates investigations and evaluation of compiler optimizations.

Computing the dynamic metrics turned out to be more difficult than we first anticipated. For this paper we have developed a framework for computing the metrics for Java programs. Our framework consists of two major pieces, a front-end JVMPI-based agent which produces relevant events, and a back-end which consumes the events and computes the metrics.

The major contributions of this paper include:

- We have identified the need to have dynamic metrics for compiler and run-time system developers and discussed why having such metrics would be of benefit.
- We provide an analysis of the different ways of presenting metrics and a discussion the general requirements for good dynamic metrics.
- We provide a detailed discussion of five groups of specific metrics that should be of interest to compiler writers and runtime developers along with specific examples of the metrics on benchmark programs.
- We provide a detailed discussion of three representative benchmarks in different categories, illustrating how the metrics relate to qualitative program behaviour, and how they can reveal behaviour that would normally be difficult to ascertain without a lengthy benchmark investigation.
- We illustrate the utility and relevance of our metrics in the context of compiler optimization, showing how to use our metrics to both guide and evaluate compiler optimizations through real examples.
- We present our framework for computing the metrics for Java programs, and a user-friendly web-based environment for presenting the metric data.

The rest of this paper is organized as follows. Section 2 discusses the requirements of good dynamic metrics and Section 3 presents three different ways of presenting the metrics. Section 4 describes our benchmark suite. In Section 5 we introduce five groups of dynamic metrics, with specific examples of metrics for each group, and we describe how the metric data relates to benchmark behaviour. Sections 6 and 7 demonstrate how to use the metrics to examine and compare benchmarks, and how the metric data can be related to the effects of common compiler optimizations. Section 8 summarizes the framework we used for collecting dynamic metrics and how results are presented. In Section 9 we provide an overview of related work and in Section 10 we address limitations and further improvements to our current framework. Finally, Section 11 gives conclusions.

2. REQUIREMENTS FOR DYNAMIC METRICS

Dynamic metrics need to exhibit a number of characteristics in order to render clear and comparable numbers for any kind of program. The following is a non-exhaustive list of desirable qualities.

- Unambiguous: one lesson learned from static metric literature is that ambiguous definitions lead to unusable metrics. For instance, the most widely used metric for program size is ’lines of code’ (LOC). LOC is sufficient to give a ball park measure of program size. However, without further specification it is virtually useless to compare two programs. Are comments and blank lines counted? What is the effect of pretty-printing? How do you compare two programs from different languages? Within a given language, the LOC of a pretty-printed version of a program with comments and blank lines removed would give an unambiguous measurement that can be used to compare two programs.
- Dynamic: obviously a dynamic metric needs to be dynamic. In other words, *the metric should measure an aspect of a program that can only be obtained by actually running the program*. In compiler optimization papers, static metrics are often given because they are easier to obtain. They tend to relate to the *cost* of a particular optimization technique (e.g. the number of virtual call sites for a de-virtualization technique), whereas dynamic metrics relate to the *relevance* of a technique (e.g. the proportion of dynamically executed monomorphic call sites). While dynamic metrics usually require more work than static measurements, the resulting numbers will be more meaningful since they will not change by adding unexecuted code to the program. Dead code should not influence the measurement. We will refer to instructions that are executed at least once as *instructions touched*, or live code.
- Robust: the other side of the coin of using dynamic measurements is the possibility that those numbers are heavily influenced by program behaviour. Where static numbers may have reduced relevance because non-executed code influences the numbers, dynamic metrics may have reduced relevance because the measured program execution may not reflect common behaviour. Unfortunately, one simply cannot guarantee that a program’s input is representative. However, one can take care to define metrics that are robust with respect to program behaviour. In other words, *a small change in behaviour should cause a correspondingly small change in the resulting metric*.

  In particular, a robust metric should not be overly sensitive to the size of a program’s input. Total number of instructions executed is not a robust metric, since a bubble sort, for example, will execute four times as many instructions if the input size is increased by a factor of two. Number of *different* instructions executed is more robust since the size of the input will not drastically change the size of the part of the program that is executed.

  To categorize aspects of program behaviour, absolute numbers are usually misleading and non-robust. For example, the total amount of allocated bytes, a metric often reported in the literature, says little about the memory-hungriness of a program. Instead one should use a relative metric such as bytes allocated per executed byte code. Merely running a program twice as long will have less effect on a relative metric.

- Discriminating: *a large change in behaviour should cause a correspondingly large change in the resulting metric*. Dynamic metrics should reflect changes in program behaviour. In this study we collected a large number of metrics for each aspect of program behaviour, many of which do not appear in this paper, because they were not the most discriminating of the tested set. For example, the number of loaded classes would seem a fairly good indication of program size, until one observes that any Java program loads at least 275 classes. Small applications (about half of our benchmark suite) load between 275 and 292 classes. Therefore this metric cannot be meaningfully used to compare the size of small programs.
- Machine-independent: since the metrics pertain to program behaviour, *they should not change if the measurement takes place on a different platform* (including virtual machine implementation). For example, number of objects allocated per second is a platform-dependent metric which disallows comparisons between measurements from different studies, because it is virtually impossible to guarantee that they all use identical platforms. On the other hand, number of objects allocated per 1000 executed bytecode instructions (kbc), is a platform-independent metric. In general, metrics defined around the byte code as a unit of measurement are machine-independent for Java programs.

3. KINDS OF DYNAMIC METRICS

While there are many possible metrics one could gather, we have found that the most commonly described metrics, and the ones which seem most useful compiler optimization, tend to be belong to just a few basic categories. This includes the ubiquitous single value metrics such as average, hot spot detection metrics, and metrics based on discrete categorization; we also mention the possibility of more detailed continuous "expansions" of these metrics. It is of course possible to design and use a metric that does not fit into these categories; these initial metric kinds, however, enable us to at least begin to explore the various potential metrics by considering whether an appropriate metric exists in each of our categories.

3.1 Value Metrics

The first kind of metric we present is a standard, usually one value answer. Many data gatherers, for instance, will present a statistic like *average* or *maximum* as a rough indicator of some quantity; the idea being that a single value is sufficiently accurate. Typically this is intended to allow one to observe differences in behaviour before and after some optimization, smoothing out unimportant variations. For example, a value such as running time is perhaps best presented as an average over several executions. Value metrics appear in almost every compiler research article that presents dynamic measurements.

3.2 Percentiles

Often in compiler optimization it is important to know whether the relative contributions of aspects of a program to a metric are evenly or unevenly distributed among the program elements. If a few elements dominate, then those can be considered "hot," and therefore worthy of further examination or optimization. Knowing, for example, that 2% of allocation sites are responsible for 90% of allocated bytes indicates that those top 2% of allocation sites are of particular interest. For comparison, a program where 50% of allocation sites contribute 90% of allocated bytes indicates a program that has a more even use of allocation, and so intensive optimization of a few areas will be less fruitful.

Similar metrics can be found in compiler optimization literature; e.g., the top *x*% of most frequently-executed methods [24].

3.3 Bins

Compiler optimization is often based on identifying specific categories of measurements, with the goal of applying different optimization strategies to different cases. A call-site optimization, for instance, may use one approach for monomorphic sites, a more complex system for polymorphic sites of degree 2, and may be unable to handle sites with a higher degree of polymorphism. In such a situation single value metrics do not measure the situation well, e.g., computing an average number of types per call site may not give a good impression of the optimization opportunities. An appropriate metric for this example would be to give a relative or absolute value for each of the categories of interest, 1, 2, or $\geq$3 target types. We refer to these kinds of metrics as "bins," since the measurement task is to appropriately divide elements of the sample space into a few categories or bins.

There are many examples of bins in the literature; e.g., categorizing runtime safety checks according to type (null, array, type) [20], the % of loops requiring less than *x* registers [33].

3.4 Continuous Metrics

Most of our metrics have continuous analogues, where the value, bin or percentile calculations are calculated at various (or all) partial stages of execution, and plotted as a graph. Motivation for continuous metrics arises from the inherent inaccuracy of a single, summary metric value in many situations: a horizontal line in a graph can have the same overall average as a diagonal line, but clearly indicates very different behaviour.

Additional descriptive values like standard deviation can be included in order to allow further refinement to a single metric datum; unfortunately, secondary metrics are themselves often inadequate to really describe the difference in behaviour, requiring further tertiary metrics, and so on. Specific knowledge of other aspects of the metric space may also be required; correct use of standard deviation, for example, requires understanding the underlying distribution space of result values. Analysis situations in compiler optimization design may or may not result in simple normal distributions; certainly few if any compiler researchers verify or even argue that property.

In order to present a better, less error-prone metric for situations where a single number or set of numbers is potentially inaccurate, a straightforward solution is to present a graph of the metric over a continuous domain (like time, or bytecode instructions executed). Biased interpretations based on a single value are thus avoided, and an astute reader can judge the relative accuracy or appropriateness of the single summary metric themselves. Continuous metrics can then be seen as an extension to metrics, giving a more refined view of the genesis of a particular value. Our focus is on establishing general characterizations of benchmarks that one could use as a basis or justification for further investigation, so we do not explicitly present any actual continuous metrics here.

4. BENCHMARKS

Table 1 lists the benchmarks that were used in this study, along with a short description of each of them. Each benchmark is also assigned an abbreviated name.

Benchmark	Abbr.	Description	Configuration
Cellular Automaton	CA	One-dimensional, two-state cellular automaton	100 cells, 20 steps
Coefficients	COEFF	Computes the coefficients of the least square polynomial curves of order 0 to *n* for the set of points using a pseudoinverse.	Max order 20, 114 points
Empty	EMPTY	Only returns from its main method	N/A
Hello World	HELLO	Only prints a greeting message before returning	N/A
JLex	JLEX	Lexical analyzer generator	Syntax describing the commands of a simple media server
Linpack	LPACK	Numerically intensive program; commonly used to measure floating point performance	N/A
Roller Coaster	RC	Classical Roller Coaster concurrency problem. A cart waits for passengers and goes for a ride, and repeats the process.	7 passengers, 50 rides, 4 passengers per cart
SableCC	SBLCC	Compiler generator	Jimple grammar (3-address representation of Java bytecode)
Soot	SOOT	Java optimization framework. Converts Java class files to a 3-address intermediate representation.	Jimple subpackage (a subset of itself)
Telecom	TCOM	Simulates users placing calls and being charged according to their duration.	15 users, 5 threads making calls
Volano Client	VOLCL	Client side of a chat room simulation	3 chat rooms, depth 4, 2 iterations.
Volano Server	VOLS	Server side of a chat room simulation	3 chat rooms, depth 4, 2 iterations.
JOlden Suite[8]			
Barnes-Hut	Obh	Solves the *N*-body problem, where the movement of the bodies has to be simulated based on the gravitational forces that they exert on each other.	4K bodies
BiSort	Obsrt	Performs two bitonic sorts, one forward and one backward.	128K integers
Em3d	Oem3d	Simulates the propagation electro-magnetic waves in a 3D object using nodes in an irregular bipartite graph to represent electric and magnetic field values.	2000 nodes of out-degree 100
Health	Ohth	Simulates the Columbian health care system, where villages generate a stream of patients, who are treated at the local health care center or referred to a parent center. Nodes in a 4-way tree are used to represent hospitals.	5 levels, 500 time steps
MST	Omst	Computes the minimum spanning tree of a graph	1K nodes
Perimeter	Operm	Computes the total perimeter of a region in a binary image represented by a quadtree. The benchmark creates an image, counts the number of leaves in the quadtree and then computes the perimeter of the image using Samet's algorithm.	64K image
Power	Opow	Solves the Power System Optimization Problem, where the price of each customer's power consumption is set so that the economic efficiency of the whole community is maximized.	10000 customers
TSP	Otsp	Computes an estimate of the best Hamiltonian circuit for the Travelling Salesman Problem.	10000 cities
Voronoi	Ovor	Computes the Voronoi Diagram of a set of points.	20000 points
SPECjvm98 Suite (all size 100, run outside the SPEC harness)			
_201_compress	COMP	A high-performance application to compress/uncompress large files; based on the Lempel-Ziv method(LZW)	
_202_jess	JESS	A Java expert shell system based on NASA 's CLIPS expert shell system	
_205_raytrace	RAY	Ray tracer application	
_209_db	DB	Performs several database functions on a memory-resident database	
_213_javac	JAVAC	JDK 1.0.2 Java compiler	Compiles JavaLex
_222_mpegaudio	MPEG	MPEG-3 audio file compression application	
_227_mtrt	MTRT	Dual-threaded version of _205_raytrace	
_228_jack	JACK	A Java parser generator with lexical analyzers (now JavaCC)	Generates itself 16 times

Table 1: Description of the benchmarks

Metric	EMPTY	HELLO	Omst	LPACK	Operm	COEFF	COMP	SOOT	JAVAC
size.appLoadedClasses.value	1	1	6	1	10	6	22	531	175
size.appLoad.value	4	7	727	1056	863	2374	6555	45111	44664
size.appRun.value	0	4	600	749	785	975	5084	25666	26267
size.appHot.value	0	4	175	59	393	57	396	2549	2759
size.appHot.percentile	n/a	100%	29%	8%	50%	6%	8%	10%	11%
size.loadedClasses.value	275	275	281	278	285	286	310	818	471
size.load.value	71818	71821	76302	77932	76438	80292	90762	126566	133172
size.run.value	7343	7793	10112	10698	9991	12880	14514	37852	37830
size.hot.value	1014	1038	186	115	398	115	396	3097	2258
size.hot.percentile	14%	13%	2%	1%	4%	1%	3%	8%	6%

Table 2: Size metrics.

5. DYNAMIC METRICS

In sections 2 and 3 we have outlined three different kinds of dynamic metrics (value, percentile and bin), and some general requirements for good dynamic metrics. In this section we present some concrete dynamic metrics that we feel are suitable for summarizing the dynamic behaviour of Java programs (although many of the metrics would also apply to other languages).

In developing our dynamic metrics we found that they fit naturally into five groups: (1) program size and structure, (2) measurements of data structures, (3) polymorphism, (4) dynamic memory use and (5) concurrency. In the following subsections we suggest specific metrics for each category, and discuss appropriate benchmark data.

5.1 Program Size and Structure

Dynamic metrics for program size and structure try to answer the question: how large is a program and how complex is its control structure?

5.1.1 Size

Before dynamic loading became commonplace, an approximation of this metric was commonly provided by the size of the executable file. With dynamic loading in place, the program must be run to obtain a useful measurement of its size. Table 2 shows several metrics related to a program's size for four distinctive benchmarks: the empty program (EMPTY), a small computational benchmark (COEFF) and the well-known Compress (COMP) and Javac (JAVAC) benchmarks. We define these metrics below and discuss their robustness, discriminating power and machine-independence. The metrics are also shown in the comprehensive tables in the appendix for all benchmarks.

size.appLoadedClasses.value: The number of application-specific classes loaded. This metric gives a rough, intuitive idea of the size of a program, as shown in Table 2. However, due to a large variability in the code size of loaded classes, it is *ambiguous*. For example, the LPACK benchmark, loads only one application class but touches more bytecode instructions than the Omst benchmark, which loads six classes. The standard libraries are excluded from this measurement, since they distort the numbers (java.*, javax.*, sun.*, com.sun.*,... etc.). For the sake of interest, Table 2 shows all metrics discussed in this section for application code (size.app* metrics, top of Table 2) and for all code including standard libraries (size.* metrics, bottom of Table 2). It is clear that measurements that include libraries are *not discriminating* enough. size.loadedClasses.value starts at 275 for the EMPTY benchmark, rises to 286 for COEFF and to 310 for COMP. Only JAVAC and SOOT really stand out with 471 and 818 total loaded classes. Moreover, size.loadedClasses.value is *not robust*, since it is influenced by the platform on which the code is executed. In preliminary experiments we used a different virtual machine, which loaded 286 classes for EMPTY. Changes in platform therefore influence this measurement more than changes in benchmark program. size.appLoadedClasses.value is robust and gives the same measurement regardless of platform.

size.appLoad.value: The number of bytecode instructions loaded in application-specific classes. Whenever a class is loaded, its size in bytecode instructions is added to a running total. Including the standard libraries in size.load.value again distorts this metric, rendering it *insufficiently discriminative*. size.appLoad.value is the closest equivalent to the static size of an executable. It is *less ambiguous* than size.appLoadedClasses.value, which is illustrated by the difference between JAVAC (175 loaded classes, 44664 loaded instructions) and SOOT (531 classes, 45111 loaded instructions). size.appLoad.value is less sensitive to the different programming styles in these benchmarks.

size.appRun.value: The number of bytecode instructions touched. This metric sums the total number of bytecode instructions which are executed at least once in the entire duration of the program's execution. Run size is smaller than load size, since dead code is not counted. We believe that this metric combines the right amount of *robustness* and *discriminative power*. Table 2 is sorted left-to-right in ascending order with respect to this metric. Separate experiments showed that size.appRun.value is robust with respect to program input: different executions did not result in substantial differences, and did not upset the ordering. It is also robust with respect to various optimizations we performed (see Section 7). Although bytecode-level optimizations changed the number of touched instructions, they did not change the ordering between benchmarks. size.appRun.value clearly discriminates benchmarks according to size, and can be used to provide a classification in five categories: XS,S,M,L,XL. Less than 100 bytecodes touched, for example HELLO or EMPTY, fall in the XS category. Between 100 and 2000 constitutes the S category (Omst,LPACK,Operm,COEFF). Between 2000 and 10K is the M category (COMP), and between 10K and 50K is the L category (SOOT,JAVAC). The XL category with more than 50K bytecodes touched is reserved for very larger programs, such as Forte.

size.appHot.value: The number of bytecode instructions responsible for 90% of execution. This metric is obtained by counting the number of times each bytecode instruction is executed, sorting the instructions by frequency, and reporting the number of (most frequent) bytecodes which represent 90% of all executed bytecodes. While this metric is *discriminating* and classifies the benchmarks in different sizes, it *lacks robustness*. We found that changing a program's input had a large effect on size.appHot.value. For example, SOOT's 2549 hot bytecodes drop to 1191 with different input.

A separate, derived metric, size.appHot.percentile measures the *proportion* of code responsible for 90% of execution, i.e. the size of the program hot spot relative to its whole size, or size.app-

Metric	COEFF	LPACK	CA	Obh	Opow	SBLCC	EMPTY
data.appArrayDensity.value	160.404	157.775	139.890	105.947	97.433	38.868	n/a
data.appCharArrayDensity.value	0.0	0.0	15.494	0.0	0.0	0.0	n/a
data.appNumArrayDensity.value	79.486	148.385	124.334	97.577	96.487	11.209	n/a
data.appRefArrayDensity.value	80.713	9.389	0.015	4.383	0.162	13.274	n/a
data.arrayDensity.value	150.939	152.170	30.881	105.891	93.418	43.551	73.496
data.charArrayDensity.value	1.513	2.077	10.208	0.012	0.016	6.547	32.549
data.numArrayDensity.value	75.033	140.929	14.603	97.513	92.491	9.416	35.025
data.refArrayDensity.value	73.772	8.890	0.400	4.380	0.156	15.636	1.874

Table 3: Array metrics.

Hot.value / size.appRun.value. As with size.appHot.value, this metric is sensitive to program input.

5.1.2 Structure

We also propose metrics to characterize the complexity of program structure. Unfortunately, we are not able to provide results for these metrics at this time, but they are part of our future work. These metrics are obtained by measuring instructions that change control flow (if, switch, invokeVirtual). A program with a single large loop is considered simple, as opposed to a program with multiple loops and/or many control flow changes within a single loop.

structure.controlDensity.value: The total number of control bytecodes touched divided by the total number of bytecodes touched.

structure.changingControlDensity.value: The total number of control bytecodes that change direction at least once divided by the total number of bytecodes touched. This measurement is smaller than the previous one, since many control bytecodes never change direction.

structure.changingControlRate.value: The number of changes in direction divided by the number of control instruction executions. This is the most dynamic measurement. It is the equivalent of the miss rate of the simplest dynamic hardware branch predictor which predicts that a branch will follow the same direction as the last time it was executed. The metric assumes that the branch history table, as this prediction scheme is known [22], is free from interference or capacity misses.

Of these metrics, we expect structure.ControlDensity.value to best characterize the complexity of program structure. It is the reciprocal of average basic block size.

5.2 Data Structures

The data structures and types used in a program are of frequent interest. Optimization techniques change significantly for programs that rely heavily on particular classes of data structures; techniques useful for array-based programs, for instance are different from those that may be applied to programs building dynamic data structures.

5.2.1 Array Intensive

Many “scientific” benchmarks are deemed so at least partially because the dominant data structures are arrays. The looping and access patterns used for array operations are then expected to provide opportunities for optimization. This is not entirely accurate since array intensity can certainly exist without necessarily computing arithmetic values based on arrays; it is however, an important indicator. Moreover, array access in Java has other opportunities for optimization, e.g., array bounds check removal [26].

Determining if a program is array intensive will then be a problem of determining if there are a relatively significant number of array accesses. This is tracked by examining traces for specific array operation bytecodes.

There are complications to such a simple approach in the context of Java. Not only is the separation between application code and runtime libraries important, but in Java multi-dimensional arrays are stored as arrays of arrays, and so the number of array operations required for each multi-dimensional array access is magnified. This skewing factor can be eliminated by ignoring array accesses where the array element is an array itself (this is planned for future work). Evidence of such skewing is given later, in section 6.

data.[app]arrayDensity.value: This is a metric describing the relative importance of array access operations. For uniformity, we express it as the average number of array access operations per kbc of executed code. Further refinement of the metric can be done according to the type of the array being accessed: data.[app]charArrayDensity.value (for character arrays), data.[app]numArrayDensity.value (for arrays of primitive numerical types), and data.[app]refArrayDensity.value for arrays of non-primitive numerical types.

Example metric calculations for array densities are given in table 3. As expected, programs based on arrays (determined by inspection) rank high: COEFF, LPACK, etc. The application versus whole program variations of this metric also show the relative impact of startup and library code; the Cellular Automaton (CA) program ranks very high in *application only* array density, but below the EMPTY program when considered as a whole program. This indicates that while the CA code itself makes extensive use of arrays (and even startup has a significant use of arrays), the library methods the CA program calls during runtime have a limited use of arrays. In fact, while the CA code does consist almost entirely of array operations, it also emits an output description of its current state at each iteration, and the amount of code involved in doing this I/O significantly dilutes the relative number of array operations. An optimization that reduces the cost of array operations (such as removing bounds checks) may thus not realize as much overall benefit as a naive understanding of the algorithm/design of the benchmark may indicate.

In Java, string operations usually reduce to operations on character arrays, and so one would expect string usage would skew results here (the data.[app]charArrayDensity.value metric shows the number of character array operations per kbc). This turns out not to be the case—intense usage of character arrays is largely confined to startup and library code. Since the actual character array for a string is a private field in the String class, almost all such operations are necessarily contained[1] in the library code. Interestingly, the startup code is by far the most intense user of char arrays; none of our benchmarks of any significant duration/size actually have a char array density larger than the EMPTY program (even the benchmarks that do parsing, such as JAVAC and SBLCC).

The threshold for considering a program array intensive is not as clear as with some other metrics; our benchmarks, application or

[1] Copies of the char array can of course be created and used outside the library, but in the benchmarks we have analyzed this does not occur often.

Metric	CA	COMP	JAVAC	SBLCC	LPACK	EMPTY
pointer.appNonrefFieldAccessDensity.value	218.661	145.913	132.827	58.648	0.001	n/a
pointer.appRefFieldAccessDensity.value	0.309	43.662	75.522	106.247	0.000	n/a
pointer.nonrefFieldAccessDensity.value	123.511	145.900	114.121	84.886	3.185	46.573
pointer.refFieldAccessDensity.value	25.372	43.658	49.778	72.381	0.428	6.347

Table 4: Pointer density measurements.

whole, tend to be fairly evenly distributed over the range of reasonable density values. In our estimation, an application density in the high 90's identifies the majority of what one would intuitively call array intensive programs.

5.2.2 Floating-Point Intensive

Programs that do numerous floating-point calculations also tend to be considered "scientific." Different optimizations apply though; including the choice of appropriate math libraries optimized for speed or compatibility, opportunities for more aggressive floating-point transformations and so on. Fortunately, floating-point operations are quite rarely used in most applications that do not actually focus on floating-point data, and so identifying floating-point intensive benchmarks is relatively straightforward.

data.[app]floatDensity.value: This single value describes the relative importance of floating-point operations. It is computed as the number of floating-point operations (including all bytecode instructions that operate on either float or double types) per kbc of executed code.

Benchmark	*data.floatDensity.value*
Opow	474.918
Otsp	471.478
LPACK	286.175
Obh	245.669
Ovor	226.363
COEFF	202.983
Oem3d	13.512
EMPTY	2.075
SOOT	0.717
JAVAC	0.072

Table 5: Floating point density.

As can be seen from table 5, high float density values correlate well with benchmarks or algorithms that have been traditionally considered to rely on numeric, floating-point data (Opow, Otsp, COEFF, LPACK etc), and low values generally correspond to non-numeric benchmarks (JAVAC, SOOT, etc). Some apparently numeric benchmarks are pruned out by this metric; the Oem3d benchmark, for example. While this program does use floating-point data in an iterative calculation, by default it only computes *one* iteration of the algorithm—a relatively significant proportion of the program is devoted to constructing and traversing the (irregular) graph structure that supports that computation, and this is very non-numeric.

We have not given the application only version of this metric here (the website, discussed in section 11, has a more complete set of metrics). Note that the EMPTY program has a very low float density value of 2.075, which necessarily dilutes the relative float density of very small floating-point benchmarks considered in their startup and library context. Even in the whole program metric, though, the division between float-intensive and not is quite sharp: the 226.363 value for Ovor drops to 13.512 for Oem3d, and then rapidly reaches small single digits for less float-intensive programs. From this we conclude that a data.floatDensity.value of at least 100 is a good indicator of floating-point intensive program.

With respect to identifying "scientific" benchmarks, it is useful to know which benchmarks combine floating-point intensity with array usage. In our benchmark list this includes COEFF, Obh, Opow, and LPACK (see tables 3 and 5). Note that while these combine intensive floating-point usage with intensive array usage, they do not necessarily contain perfect loops over arrays. Obh, for instance, uses numerous arrays and vectors, but computationally is largely based on traversing and modifying a tree structure. Structural metrics that would aid in identifying loop intensity are described in section 5.1.2.

5.2.3 Pointer Intensive

Dynamic data structures are manipulated and traversed through pointers or object references. Programs that use dynamic data structures are thus expected to perform a greater number of object dereferences leading to further objects then a program which uses local data or arrays as primary data structures; a basic metric can be developed from this observation. Of course a language such as Java which encourages object usage can easily skew this sort of measurement: an array-intensive program that stores array elements as objects (e.g., Complex objects) will result in as many object references as array references. A further complication is due to arrays themselves; arrays are considered objects in Java, and so an array access will appear as an object access unless special care is taken to differentiate them.

Below we define a few metrics that would measure pointer intensity. The first few are based on a naive, but easily computable method for assessing pointer usage. They do not different between objects and arrays.

pointer.[app]RefFieldAccessdensity.value: and pointer.[app]-NonrefFieldAccessDensity.value. These metrics give a coarse indication of the importance of pointer references in a program. The former is computed as the average number of field access operations that reach an object field, and the latter as field accesses that reach a primitive field, all per kbc of executed code. In a pointer-intensive program, one expects that the effect of following pointers references to object fields (as opposed to references to primitive fields) will result in a relatively high pointer.[app]RefFieldAccessDensity.value.

Examples of this metric are shown in table 4. From this, the CA program is clearly very *not* pointer-intensive—it has a very high application primitive field access density (218.661), but an extremely low reference field access density (0.309). Inversely, SBLCC has an extremely higher reference density (106.247), and a moderate primitive field access density (58.648). LPACK has almost no field accesses of either kind; it is a very non-object-oriented benchmark that creates just one object with only one field (a primitive type).

The nature of JAVAC and COMP is less clear. In the case of COMP, which one would expect not to be pointer intensive, the high reference field access density is (we strongly suspect) largely due to accessing buffers and arrays of constants. The coarseness of this metric thus seems adequate to identify applications that are unambiguously pointer or non-pointer intensive, but is not accurate enough to identify pointer-intensity in all cases, particularly in the face of arrays as objects.

We define potentially more accurate metrics below, but limitations of JVMPI make these difficult to compute with our current system—these are intended for future investigation with an instru-

Metric	COEFF	COMP	JESS	Obh	VOLS	SOOT	JAVAC	Operm
polymorphism.appCallSites.value	85	54	737	129	526	3141	2617	49
polymorphism.callSites.value	677	606	1309	639	2592	3766	3234	550
polymorphism.appInvokeDensity.value	66.0	16.5	59.2	18.7	121.0	70.8	72.3	45.9
polymorphism.appReceiverArity.bin (1)	100.0%	98.1%	98.4%	96.9%	98.1%	93.0%	78.4%	69.4%
polymorphism.appReceiverArity.bin (2)	0.0%	1.9%	0.8%	3.1%	0.8%	3.1%	9.7%	0.0%
polymorphism.appReceiverArity.bin (3+)	0.0%	0.0%	0.8%	0.0%	1.1%	3.9%	12.0%	30.6%
polymorphism.appReceiverArityCalls.bin (1)	100.0%	100.0%	99.2%	86.7%	85.4%	78.1%	72.6%	36.9%
polymorphism.appReceiverArityCalls.bin (2)	0.0%	0.0%	0.1%	13.3%	4.3%	15.1%	15.1%	0.0%
polymorphism.appReceiverArityCalls.bin (3+)	0.0%	0.0%	0.8%	0.0%	10.4%	6.3%	12.3%	63.1%
polymorphism.appReceiverCacheMissRate.value	0.0%	0.0%	0.4%	3.0%	5.9%	4.3%	7.2%	41.1%
polymorphism.appTargetArity.bin (1)	100.0%	98.1%	98.9%	96.9%	99.2%	95.0%	89.7%	77.6%
polymorphism.appTargetArity.bin (2)	0.0%	1.9%	0.4%	3.1%	0.2%	2.2%	3.8%	0.0%
polymorphism.appTargetArity.bin (3+)	0.0%	0.0%	0.7%	0.0%	0.6%	2.8%	6.5%	22.4%
polymorphism.appTargetArityCalls.bin (1)	100.0%	100.0%	99.2%	86.7%	94.5%	83.5%	92.0%	42.8%
polymorphism.appTargetArityCalls.bin (2)	0.0%	0.0%	0.0%	13.3%	0.7%	13.9%	1.9%	0.0%
polymorphism.appTargetArityCalls.bin (3+)	0.0%	0.0%	0.8%	0.0%	4.9%	2.6%	6.1%	57.2%
polymorphism.appTargetCacheMissRate.value	0.0%	0.0%	0.4%	3.0%	3.7%	2.6%	3.1%	37.5%

Table 6: Polymorphism metrics.

mented virtual machine, and we have not yet measured all these values.

Pointer polymorphism is typically measured as an average or maximum number of target addresses per pointer, and symmetrically number of pointers per target address (Cheng and Hwu argue that both are required for a more accurate measurement [10]). This can be computed as a value metric; a bin version can also be appropriate, and is defined following.

pointer.pointsToCount.value: This metric along with the symmetric pointer.pointsFromCount.value metric measures the average number of distinct objects referenced by each object reference and the average number of object references directed at each object respectively.

pointer.pointsTo.bin: A pointer analysis system is most interested in identifying pointers that can be directed at one address, possibly two, but further divisions are often unnecessary. A bin metric can provide a more appropriate view in this case. Each bin gives the percentage of object references that referenced 1 object, 2 objects, and ≥3 objects. The symmetric companion bin, pointer.pointsFrom.bin, has bins for the percentage of objects that had 1, 2 or ≥3 references.

5.3 Polymorphism

Polymorphism is a salient feature of object-oriented languages like Java. A polymorphic call in Java takes the form of an invokeVirtual or invokeInterface byte code. The target method of a polymorphic call depends on the run time type of the object receiving the call. In programs that do not employ inheritance, this target never changes and no call is truly polymorphic. The amount of polymorphism can therefore serve as a measurement of a program's object-orientedness.

Table 6 presents polymorphism metrics for eight distinctive benchmarks. The first three metrics measure the number of potentially polymorphic instructions, but say nothing about whether this polymorphism is realized (i.e. whether the target method or the receiver type of an invokeVirtual actually changes at run time):

polymorphism.[app]CallSites.value: The total number of different call sites executed. This measurement does not include static invoke instructions, but does count virtual method calls with a single receiver. Since in Java all methods are by default virtual, even if they only have a single implementation, this metric does not reflect true polymorphism. Instead, it provides a measurement of program size. From a compiler optimization point of view, this metric gives an indication of the *amount of effort* required to optimize polymorphic calls, but not of their relevance. polymorphism.appCallSites.value, which counts only application-specific call sites, is more discriminating than polymorphism.callSites.value, which counts library calls as well. In the remainder we will only look at application-specific metrics.

polymorphism.appInvokeDensity.value: The number of invokeVirtual and invokeInterface calls per kbc executed. This metric estimates the importance of invoke bytecodes relative to other instructions in the program, indicating the *relevance* of optimizing invokes. In COMP, only 16 out of 1000 bytecodes are invokes, indicating that non-static invokes are not important to a program's performance. In contrast, VOLS executes 120 invokes per 1000 bytecodes, indicating small method sizes and greater relevance of virtual call optimizations. However, these metric say nothing about how polymorphic the invoke instructions are.

The following metrics measure true polymorphism. There are two variants. In the first variant, we take into consideration the number of receiver types, in the second we use the number of different target methods. There are more receiver types than targets, since two different object types may inherit the same method from a common super class. Some optimization techniques, such as class hierarchy analysis [14], optimize call sites with a restricted number of targets. Others, such as inline caching [15], optimize call sites with a restricted number of receiver types.

5.3.1 *Receiver Polymorphism*

Receiver polymorphism can be measured in at least three different ways, which are progressively more dynamic:

polymorphism.appReceiverArity.bin: This is a bin metric, showing the percentage of all *call sites* that have one, two and more than two different receiver types. The metric is dynamic, since we measure the number of different types that actually *occur* in the execution of the program (thus representing an ideal case for comparison with type inference techniques which conservatively estimate the number of receiver types without running the program). However, by counting call sites, the metric does not reflect the importance of those sites.

polymorphism.appReceiverArityCalls.bin: This is a bin metric, showing the percentage of all *calls* that occur from a call site with

Metric	EMPTY	Obsrt	Oem3d	JACK	JESS	JAVAC	SOOT	SBLCC
memory.byteAllocationDensity.value	1750	11	36	314	294	132	274	345
memory.byteAppAllocationDensity.value	n/a	6	15	19	175	78	148	71
memory.averageObjectSize.value	192	43	446	54	67	41	35	62
memory.averageAppObjectSize.value	56	24	291	31	42	29	21	24
memory.appobjectSize.bin (8)	0%	0%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin (16)	0%	0%	0.1%	8.4%	0%	23.2%	61.1%	48.9%
memory.appobjectSize.bin (24)	0%	100%	0%	1.6%	20.0%	22.1%	29.8%	22.2%
memory.appobjectSize.bin (32)	0%	0%	0%	81.0%	30.3%	39.3%	0.7%	18.8%
memory.appobjectSize.bin (40)	0%	0%	33.3%	8.9%	1.1%	9.9%	7.6%	9.4%
memory.appobjectSize.bin (48-72)	100%	0%	0%	0%	48.3%	5.4%	0.6%	0.5%
memory.appobjectSize.bin (80-136)	0%	0%	0%	0%	0.2%	0%	0.1%	0.1%
memory.appobjectSize.bin (144-392)	0%	0%	10.9%	0%	0%	0%	0%	0.1%
memory.appobjectSize.bin (400+)	0%	0%	55.7%	0%	0%	0%	0%	0%

Table 7: Dynamic Memory Metrics

one, two and more than two different receiver types. This metric measures the importance of polymorphic calls. It is more dynamic than the previous metric. For example, COMP has 1.9% call sites with two receiver types, but these are almost never called (polymorphism.appReceiverArityCalls.bin(2) = 0%). Typically, monomorphic call sites are executed less frequently than polymorphic call sites. In polymorphism.appReceiverArity[Calls].bin(1) the percentage of monomorphic calls is lower than the percentage of monomorphic call sites for Obh, VOLS, SOOT, JAVAC and Operm. Operm is the most extreme case, with 30.6% heavily polymorphic call sites, which are executed 63.1% of the time. We consider this metric to be best for an appraisal of polymorphism, since it highlights polymorphism that actually occurs, weighted by the frequency of its occurrence. Table 6 is therefore sorted in descending order from left to right, using polymorphism.appReceiverArityCalls.bin.

polymorphism.appReceiverCacheMissRate.value: This metric shows as a percentage how often a call site switches between receiver types. This is the most dynamic measurement of receiver polymorphism, and it represents the miss rate of a true inline cache. It is potentially non-robust, since the miss rate of an inline cache can be heavily influenced by the ordering of the receiver types: a call site with two different receiver types can have a cache miss rate varying between 0% (objects of one type precede all objects of the other type) and 100% (two types occur in an alternating sequence). SOOT, for example, executes 21.9% non-monomorphic calls (1 - polymorphism.appReceiverArityCalls.bin(1)), but has a receiver cache miss rate of only 4.4%. Operm, on the other hand, has 63.1% non-polymorphic calls, and a receiver cache miss rate of 41.1%, indicating that the polymorphic call sites actually switch often between receiver types.

5.3.2 *Target Polymorphism*

Target polymorphism can be measured in a similar manner as receiver polymorphism, but now we count the number of different targets instead of the number of different receiver types:

polymorphism.appTargetArity.bin: This is a bin metric, showing the percentage of all *call sites* that have one, two and more than two different target methods. Like polymorphism.appReceiverArity.bin, this metric is dynamic, but does not reflect the run time importance of call sites. This metric is useful for compiler optimizations aimed at devirtualization: whenever a compiler can prove that a call site only has a single possible target, the call can be replaced by a static call or inlined [29]. The number of target-monomorphic call sites is always larger than that of receiver-monomorphic call sites.

polymorphism.appTargetArityCalls.bin: This is a bin, showing the percentage of all *calls* that occur from a call site with one, two and more than two different target methods. The same observations hold as for receiver polymorphism, but the number of monomorphic calls is larger.

polymorphism.appTargetCacheMissRate.value: This shows as a percentage how often a call site switches between target methods. It represents the miss rate of an idealized branch target buffer [18]. It is always lower than the corresponding inline cache miss rate (polymorphism.appReceiverCacheMissRate.value), since targets can be equal for different receiver types. Accordingly, this metric can also be heavily influenced by the order in which target methods occur.

5.4 Memory Use

For considering the memory use of programs, we concentrate on the amount and properties of dynamically-allocated memory (memory use for the stack is related to the call graph metrics, and memory for globals is not usually a dynamically varying value).

5.4.1 *Allocation Density*

The first metric required is just a simple value metric to measure how much dynamic memory is allocated by the program, per 1000 bytecode instructions (kbc) executed, and there are two variations.

memory.byte[App]AllocationDensity.value: Measures the number of *bytes* allocated per kbc executed. It is computed as the total number of bytes allocated by the program, divided by the (number of instructions executed/1000).

memory.object[App]AllocationDensity.value: Similar to the previous metric, but reports the number of *objects* allocated per kbc executed.

For the memory metrics, the App version of the metric counts only those objects (and arrays) that have a type that is a user-defined class, whereas the ordinary version counts all memory allocated.

In Table 7 we see the memory allocation density varies quite widely. The EMPTY program shows a surprisingly high allocation density, 1750 bytes per kbc. This shows that system startup and class loading of library methods is quite memory hungry. Of the remaining benchmarks, Obsrt and Oem3d have fairly low densities, while the rest are quite high, with SBLCC having the highest at 345 bytes per kbc. One would expect that benchmarks with low allocation densities would not be as suitable for use in examining different memory management schemes.

Although these metrics give a simple summary of how memory-hungry the program is overall, they do not distinguish between a program that allocates smoothly over its entire execution and a program that allocates only in some phases of the execution. To show this kind of behaviour, there are obvious continuous analogues, where the number of bytes/objects allocated per kbc is computed per execution time interval, and not just once for the entire execution (memory.byteAllocationDensity.continuous and memory.objectAllocationDensity.continuous).

5.4.2 Object Size Distribution

memory.averageObjectSize.value: The average size of objects allocated can be computed using the ratio of memory.byteAllocationDensity.value to the memory.objectAllocationDensity.value. This metric is somewhat implementation dependent, as the size of the object header may be different in different JVM implementations.

In Table 7 we see that the EMPTY program allocates relatively large objects, meaning that large objects are created on VM startup and class loading. The Oem3d benchmarks also allocates large objects (arrays of type Node), although one should keep in mind that overall it has a low allocation density, so perhaps these allocations are not so important. All other benchmarks allocate fairly small objects.

Rather than just a simple average object size, one might be more interested in the distribution of the sizes the objects allocated. For example, programs that allocate many small objects may be more suitable for some optimizations such as object inlining, or special memory allocators which optimize for small objects.

memory.[app]objectSize.bin: Object size distributions can be represented using this bin metric, where each bin contains the percentage of all objects allocated corresponding to the sizes associated with each bin. In order to factor out implementation-specific details of the object header size we use bin 0 to represent all objects which have no fields (i.e. all objects which are represented only by the header). In order to capture commonly allocated sizes in some detail, bins 1, 2, 3, and 4 correspond to objects using $h+1$ words ($h+4$ bytes), $h+2$ words, $h+3$ words and $h+4$ words respectively, where h represents the size of the object header. Then, increasingly coarser bins are used to capture all remaining sizes, where bin 4 corresponds to objects with size $h+5\ldots h+8$, bin 5 corresponds to objects with size $h+9\ldots h+16$, bin 6 corresponds to objects with size $h+17\ldots h+48$ and bin 7 corresponds to all objects with size greater than $h+48$. Note that the sum of all bins should be 100%.

In Table 7 we give the bins for the application objects (i.e. allocations of user-defined type). Each bin corresponds to objects with a size, or range of sizes, in bytes. With the exception of Oem3d, all benchmarks seem to allocate relatively small objects, indicating that the most frequently used user objects contain relatively few fields. The Obsrt benchmarks stands out because all its allocated objects are in 1 bin, objects of size 24 bytes. Note that the EMPTY benchmark distribution includes the application class object itself.

5.5 Concurrency and Synchronization

Optimizations that focus on multithreaded programs need to identify the appropriate opportunities. A basic requirement is to know whether a program does or can actually exhibit concurrent behaviour, or is it effectively single-threaded, executing one thread at a time. This affects the application of various optimization techniques, most obviously synchronization removal and lock design, but also the utility of other analyses that may be constrained by conservative assumptions in the presence of multithreaded execution (e.g., escape analysis).

Since the use of locks can have a large impact on performance in both single and multithreaded code, it is also useful to consider metrics that give more specific information on how locks are being used. A program, even a multithreaded one that does relatively little locking will obviously have a correspondingly reduced benefit from optimizations designed to reduce the cost of locking or number of locks acquired. Lock design and placement is also often predicated on knowing the amount of contention a lock experiences; this can also be exposed by appropriate metrics.

5.5.1 Concurrent

Identifying concurrent benchmarks involves determining whether more than one thread[2] can be executing at the same time. This is not a simple quality to determine; certainly the number of threads started by an application is an upper bound on the amount of execution that can overlap or be concurrent, but the mere existence of multiple threads does not imply they can or will execute concurrently.

For an ideal measurement of thread concurrency, one needs to measure the application running on the same number of processors that would be available at runtime, and also the same scheduling model. Unfortunately, these properties, as well as timing variations at runtime that would also affect scheduling, are highly architecture (and virtual machine) dependent, and so truly robust and accurate dynamic metrics for thread concurrency are difficult, perhaps impossible to define. The metrics we describe below are therefore necessarily approximate; note that limitations in JVMPI have not allowed us to compute these metrics yet. Future work based on a modified virtual machine will be used to calculate and validate these metrics.

concurrency.threadDensity.value: An approximate, but at least computable dynamic metric for thread concurrency is to consider the maximum number of threads simultaneously in the ACTIVE or RUNNABLE states. These are threads that are either running, or at least capable of being run (but which are not currently scheduled). In this way we do not require as many processors as runnable threads to show concurrent execution. Unfortunately, this will certainly perturb results: two short-lived threads started serially by one thread may never overlap execution on a 3-processor; on a uniprocessor, however, scheduling may result in all three threads being runnable at the same time. Given the considerable potential variation in thread activity already permitted by Java's thread scheduling model (which provides almost no scheduling guarantees) we do not feel that this amount of extra imprecision will overly obscure the actual concurrency of an application.

concurrency.threadDensity.bin: The amount of code executed while another thread is executing is also of interest; it gives an indication of how "much" concurrent execution exists. As with the previous metric, number of processors and scheduling discipline will make this a difficult concept to measure accurately; again we resort to coarser approximations based on active and runnable threads. This quantity is then calculated as % of kbc executed while there are specified levels of concurrency: 1, 2, ≥ 3 threads active or runnable.

5.5.2 Lock Intensive

An important criterion in optimizing synchronization usage is to know whether a program does a significant number of lock opera-

[2] Note that the JVM will start several threads for even the simplest of programs (e.g. one or more garbage collector threads, a finalizer thread, etc). When identifying concurrency by the number of existing threads it is necessary to discount these if every benchmark is not to be considered trivially concurrent.

Metric	CA	CA-NO	DB	JACK	RC	SBLCC	TCOM	VOLCL	VOLS	EMPTY
concurrency.lockDensity.value	4.827	0.134	2.137	2.126	2.018	1.291	1.164	0.738	0.571	0.163
concurrency.lock.percentile	31.6%	66.7%	3.2%	21.2%	36.0%	5.8%	25.7%	12.2%	21.0%	66.7%
concurrency.lockContendedDensity.value	0.0	0.0	0.0	0.0	0.451	0.000	0.086	0.026	0.017	0.0
concurrency.lockContended.percentile	n/a	n/a	n/a	100.0%	33.3%	87.1%	19.1%	63.0%	54.3%	n/a

Table 8: Locking metrics.

tions (entering of synchronized blocks or methods). This is quickly seen from the single value metric of an average number of lock operations per execution unit (kbc). Continuous versions of the same would enable one to see if the locking behaviour is concentrated in one section of the program, or is specific to particular program phases.

concurrency.lockDensity.value: This single value metric gives average number of lock (`monitorenter`) operations per kbc. Programs which frequently pass through locks will have a relatively high density—note that since synchronized blocks are defined without knowing whether more than one thread will be running, this metric is irrespective of any actual concurrency.

Table 8 shows metrics for benchmarks with the highest lock density of our benchmark suite. This includes CA, DB, JACK, RC, SBLCC, TCOM, Volano (VOLCL and VOLS), spanning a lock density range from 4.827 for CA to 0.571 for VOLS. RC, Volano, and TCOM are explicitly multithreaded benchmarks that contain significant amounts of synchronization and relatively little actual computation, so one would expect them to have a high lock density. Inclusion of the others, CA in particular, is less intuitive, and merits further investigation.

The actual CA code itself consists of iteratively applying arithmetic operations on arrays. However, as mentioned in section 5.2.1, each iteration within CA requires generating output on `System.out`. The Java library calls for streamed output naturally incorporate synchronization, and this turns out to be the source of the relatively high synchronization count. This is further supported by the metrics for the same benchmark with the output methods disabled (shown as "CA-NO" in table 8); in this case the lock intensity drops to 0.134, below that of the EMPTY program.

SBLCC does not do any locking itself, but does through frequent invocation of library methods (I/O and strings). In the case of JACK and DB, we note that JACK has been previously reported to have the highest absolute number of synchronized objects of any of the Spec JVM98 benchmarks [1], while DB has the highest absolute number of total synchronizations. [23].

concurrency.lock.percentile: Locks may be amenable to hot spot optimization—specific locks can be optimized for use by a certain number of threads, or code can be specialized to avoid locking. Whether high-use locks exist or not can be identified through a percentile metric, showing that a large percentage of lock operations are performed by a small percentage of locks; for our metric we define this as the percentage of locks responsible for 90% of lock operations.

From table 8, the EMPTY benchmark has 66.7% of locks responsible for 90% of locking. Lock usage in the startup code is thus not perfectly evenly distributed, but does not indicate significant hot spots. SBLCC and DB have the smallest number of hot locks, just 5.8% and 3.2% respectively. Hot spots here exist, but at least for SBLCC they are contained in the library code.

concurrency.lockContendedDensity.value: Adaptive locks can make use of knowing whether a lock will experience contention. This allows them to optimize behaviour for single-threaded access, but also to adapt to an optimized contended access behaviour if necessary [6]. Similarly, lock removal or relocation strategies will be better if they have information on which locks are (perhaps just likely) high, low or no-contention locks. A simple metric relevant to these efforts is to try and measure the importance of contention; this can be a value giving the average number of contended lock entry operations per kbc.

For most benchmarks, contention is relatively rare, and the contended density is less than one in a million. Volano, RC and TCOM, the benchmarks designed to test multithreading and synchronization, have the highest density. Note that even for these, the actual density value is small; this suggests that techniques based on presumed low contention will be (and indeed are) effective [1, 23].

The concurrency.lockContended.percentile metric shows the existence of hot spots of contention. In our suite only highly multithreaded programs, TCOM, Volano, and RC, have percentiles significantly less than 90%. RC has a somewhat mild contended percentile of 33%; this actually corresponds with the algorithm design of this particular implementation, which uses multiple locks to avoid contention bottlenecks. The cause for the distribution of Volano's hot spots is not entirely clear (Volano is closed source, which impedes investigation), but we note that it has a relatively high number of locks [1], and so contention hot spots are less likely.

The telecom benchmark has the smallest percentile, 19.1%. Although the benchmark tries to reduce lock contention through the use of separate locks for each of 15 resources, the resources themselves are all accessed through a single `java.util.Vector`. High contention on the single lock associated with that object results in an overall high contended percentile.

concurrency.lockContended.bin: Knowing the relative number of contended and uncontended lock operations can give a more precise idea of how contention is being experienced—as in TCOM, the amount of contention may not be evenly distributed among contended locks. A locking algorithm that adapts to contention level may then be appropriate. A bin metric, can be used to classify locks and the relative amount of contention; this metric will describe the relative percentage of lock requested while already held by 0, 1, or $\geq$2 threads. We hope to compute this metric as part of our future work.

6. ANALYZING BENCHMARKS

In this section we describe four different benchmarks in terms of our various metrics. This analysis is meant to demonstrate not only the information available through such metrics, but also how combined metric information can lead to a more complete picture of benchmark behaviour. The benchmarks we consider are Coefficients (COEFF), compress (COMP), javac (JAVAC), and for startup cost comparison, the empty benchmark (EMPTY). A table of metrics for each is shown in table 9 (this table includes a version of Coefficients that has had loop invariant removal applied (COEFF-LI)).

Program Size and Structure

As discussed in section 5.1, these benchmarks form a progression from small to large. What is surprising is the relative impact of startup code. Considering the size.run.value metric which includes startup and libraries, the relative sizes of these programs is

Metric	COEFF	COEFF-LI	COMP	JAVAC	EMPTY
size.appRun.value	975	989	5084	26267	0
size.run.value	12880	12894	14514	37830	7343
data.appArrayDensity.value	160.404	129.877	52.150	15.471	n/a
data.appNumArrayDensity.value	79.486	90.443	52.150	0.359	n/a
data.appRefArrayDensity.value	80.713	39.200	0.0	3.543	n/a
data.arrayDensity.value	150.939	122.175	52.152	37.919	73.496
data.fbatDensity.value	202.983	228.784	0.0	0.072	2.075
polymorphism.appReceiveArityCalls.bin (1)	100.0%	100.0%	100.0%	72.6%	n/a
polymorphism.appReceiveArityCalls.bin (2)	0.0%	0.0%	0.0%	15.1%	n/a
polymorphism.appReceiveArityCalls.bin (3+)	0.0%	0.0%	0.0%	12.3%	n/a
polymorphism.appInvokeDensity.value	65.973	75.390	16.532	72.305	n/a
polymorphism.appReceiverCacheMissRate.value	0.0	0.0	0.0	0.072	n/a
polymorphism.receiverCacheMissRate.value	0.001	0.001	0.0	0.088	0.090
memory.averageObjectSize.value	144.603	144.633	13077.619	41.447	191.949
memory.objectAllocationDensity.value	0.751	0.848	0.001	3.181	9.119
memory.objectSize.bin (8)	0.1%	0.1%	0.3%	0.0%	0.6%
memory.objectSize.bin (16)	14.4%	14.4%	10.2%	14.6%	14.7%
memory.objectSize.bin (24)	38.8%	38.8%	37.3%	41.9%	32.9%
memory.objectSize.bin (32)	3.7%	3.7%	6.2%	20.7%	8.7%
memory.objectSize.bin (40)	4.1%	4.1%	5.4%	6.2%	9.5%
memory.objectSize.bin (400+)	7.6%	7.6%	3.2%	0.2%	0.9%
concurrency.lockDensity.value	0.217	0.244	0.0	0.230	0.163

Table 9: Metrics used for analysis in section 6.

far less apparent; e.g., COMP is only 1.12 times larger than COEFF, whereas in the size.appRun.value metric the ratio is more than 5.2. Startup code, in fact, accounts for the bulk of touched bytecode instructions in all but JAVAC; in our parlance startup already constitutes a large program. When assessing a small benchmark it is thus fairly critical to separate out the potentially large effects that may be due startup.

Data Structures

Array usage of the applications is reflected in the data.appArrayDensity.value metric; the EMPTY program has no arrays, JAVAC has some array accesses (15.471), COMP a significant but not large number (52.15), and COEFF has the highest (160.404). These numbers correspond to a reasonable perception of the relative importance of array usage in these benchmarks. Again, the non-application versions of this metric indicates the strong influence of startup code; in the data.arrayDensity.value metric, the EMPTY program has a density of 73.496, diluting the value of COEFF's density (150.939), and doubling the density of JAVAC. Although JAVAC is a large benchmark in terms of bytecodes touched, it is not a particularly long-running one, and so startup is more evident in the metrics. COMP is however largely unaffected; in this case there are enough application bytecode instructions executed to almost eliminate the relative effect of startup.

A further breakdown of the array density metric can also show the importance of understanding exactly how a metric is computed, and how potential skewing factors even within the application itself may influence it. COEFF has an almost equal density of accesses to numerical (primitive types) arrays as reference arrays (all object types), whereas the array density of compress comes entirely from numerical arrays; this is shown through the data.appNumArrayDensity.value and data.appRefArrayDensity.value metrics. In fact, from an inspection of the source both benchmarks actually use almost exclusively numerical arrays. This can be explained by the way arrays are represented in Java; in the case of COMP, arrays are primarily one-dimensional, mostly `byte` arrays, and so each array element access corresponds to an access to an array of numerical type. For COEFF, however, arrays are almost all two-dimensional, and so each element access requires first an access to the outermost dimension (elements are of array and hence reference type), followed by a numerical array access to the actual primitive value. This can be demonstrated through an optimization such as loop invariant removal (the COEFF-LI column); in this case outer array index calculations are found to be invariant in the inner loops, and moved outside the inner loop reducing the number of reference array operations. The difference is evident in a comparison of the same metrics for both versions of Coefficients—application numerical and reference array densities of 79.486 and 80.713 respectively change to 90.443 and 39.200 when loop invariant removal is applied.

Use of floating point is relatively uncomplicated. The EMPTY program uses a small amount of floating point in startup, and JAVAC and COMP use none (small values for the data.floatDensity.value metric are due to startup). COEFF, the only benchmark that does use floating point data is clearly identified as float-intensive through its relatively high score of 202.983. Float density of the loop invariant removal version of Coefficients is higher still (228.784)—the reduction in number of array operations increases the relative density of floating point operations.

Polymorphism

The benchmarks also illustrate essential differences with respect to "object-orientedness", as measured through method polymorphism. By examining the application code, one would expect JAVAC to be reasonably polymorphic and compress to be very non-polymorphic (one large method dominates computation). COEFF is composed of several classes, and superficially appears to have potential for polymorphism; closer inspection reveals that there is no significant application class inheritance, and there should be no polymorphism.

These perceptions are validated by the various polymorphism metrics. The polymorphism.appReceiver.ArityCalls.bin metric, for instance shows that 100% of both COEFF's and COMP's invokeVirtual or invokeInterface call sites reach exactly one class type; in other words, they are completely monomorphic. The lack of inheritance in COEFF is thus evident in the metric. JAVAC does have a significantly smaller percentage of monomorphic call sites, and even has some sites associated with 3 or more types. It's non-0 polymorphism.appReceiverCacheMissRate.value also supports the perception that JAVAC is qualitatively more polymorphic than the other two.

Memory Use

Memory use between the benchmarks is also quite different. This can be seen simply through the memory.averageObjectSize.value metric: JAVAC has an average object allocation size of 41.447 bytes, COEFF 144.603 bytes, and COMP over 13,077 bytes. These programs are of course not all allocating objects at the same rate; the memory.objectAllocationDensity.value metric shows that while COMP allocates large objects, it does not allocate very many (density of 0.001), and so despite its large average size, it is not a memory intensive program. COEFF allocates more often (0.751), and JAVAC allocates relatively frequently (3.181).

These numbers and judgements are quite reasonable given the algorithms the benchmarks implement. JAVAC's data structures for parsing and representing its input would naturally be reasonably small and numerous. There is further evidence for this in the memory.objectSize.bin metric, where JAVAC allocates proportionally more small objects (24-32 bytes) than the other benchmarks. COMP allocates a few large arrays to use as buffers and tables, but otherwise does little allocation. COEFF iteratively allocates two-dimensional arrays of increasing size, and this aggregate effect also shows up in a larger proportion of larger objects ($\geq$ 400 bytes).

Concurrency and Locking

None of the benchmarks being compared here are explicitly concurrent; any actual concurrency is entirely due to library and/or internal virtual machine threads. Locking is also very low in all cases; JAVAC and COEFF have the highest lock density (0.230 and 0.216 respectively), and this rises only slightly for the loop invariant version (due to the decreased number of (invariant) calculations). COMP unsurprisingly has a very low lock density (below 0.001), owing to its long internal calculations. For these three programs, none have a density dramatically higher than the EMPTY program (0.163); they are all minimally lock intensive.

7. OPTIMIZATION EFFECTS

Dynamic metrics can be used for both suggesting opportunities for program optimizations/transformations and for evaluating the effect of such transformations. In order to demonstrate this we have studied the effect of several transformations on the `Voronoi` benchmark (all versions of the benchmark have been run through Soot, a Java bytecode transformation framework, for consistency).

In Table 10 we give both the relevant dynamic metrics (top part) and the runtime performance (bottom part) of four variations of the `Voronoi` benchmark. In the following subsections we first discuss the top part of the table, and then the bottom part.

7.1 Effect of Transformations on Dynamic Metrics

We started our study by first examining the dynamic metrics for the original benchmark (column labelled *orig.*). Note that the benchmark executes about 445 million bytecode instructions (base.executedInstructions.value), but has a relatively small size of application code, only 1008 different bytecode instructions are run (size.appRun.value). Of these 1008 instructions, 330 instructions represent 90% of the execution (size.appHot.value).

The most interesting metrics for this benchmark have to do with the density and polymorphism of the virtual method calls. The benchmark has a very high density of calls to virtual methods, as the polymorphism.appInvokeDensity.value is 116.17. This means that this benchmark executes a virtual invocation about 1 out of every 10 bytecode instructions. Furthermore, none of these invokes are polymorphic at runtime (polymorphism.appReceiverPolyDensityCalls.value is 0 and polymorphism.appTargetArity.bin(1) is 1). The high invoke density indicates that inlining is probably a good idea, and the low polymorphism suggests that compiler techniques could likely resolve each call site to exactly one method (thus enabling the inlining).

The column labelled *Inline* gives the dynamic metrics for the same benchmark after we applied inlining using the Soot framework. Note that this had a dramatic effect on the benchmark as the transformed benchmark executes about 288 million instructions (down from 445M), and the invocation density has reduced from 116.17 to 10.84. However, we also note potentially negative effects, the size of the running application has increased from 1008 to 1449, and the size of the hot part of the application has increased from 330 to 683.

The column labelled *-O* gives the dynamic metrics for the inlined version of the benchmark after we applied intra-method scalar optimizations enabled by the *-O* option of Soot. Note that this does make a small impact on the benchmark, reducing the number of executed instructions to 280M, and the size of the application to 1417 instructions.

After applying inlining and scalar optimizations, we looked for further opportunities for optimization, and by examining the dynamic metrics we found the the density of field accesses was very high in our transformed program (pointer.appFieldAccessDensity.value = 200.9, in column *-O*). About 1 in 5 bytecode instructions is an access (either a get or a put) to a field. In order to reduce the density of field accesses we applied a whole program points-to analysis, followed by common-sub-expression removal of field accesses, as shown in the column labelled *PT + CSE*. In this transformation a segment of code with a repeated use of some field, say `...a=p.x; ...b=p.x; ...` is transformed to put the field in in a scalar variable, and then reuse the scalar, for example `...temp = p.x; a=temp; ...b=temp; ...`. Note that in this transformation we reduce the number of field accesses, but increase the number of total instructions, since we have to insert the assignment to the temporary. This extra assignment may eventually get eliminated via copy propagation, but it may not. Indeed, our metrics show that after applying the transformation the number of executed instructions increased from 280M to 282M and the size of the application goes up from 1417 instructions to 1425 instructions. However, it does have the desired effect on the field accesses, where the field access density has been reduced from 200.9 to 177.8.

7.2 Effect of Optimizations on Runtime Performance

As we have seen in the previous subsection, the dynamic metrics help us identify opportunities for optimizations/transformations and they can also help us understand the effect, both positive and negative, of the optimization. In the bottom part of Table 10 we give some runtime measurements to see if the behaviour predicted by the metrics has any correlation with the real runtime behaviour observed when running the program on a real VM. The runtime experiments were done using Sun's Hotspot(TM) Client VM (build 1.4.1_01-b01,mixed mode), running under Debian Linux. In order to get reliable and repeatable results, for the runtime experiments we used a slightly larger problem size than when collecting the metrics (we used 20000 nodes for collecting the metrics and 100000 nodes for the runtime experiments). The runtime numbers are the average of five runs, reporting the total time as re-

Metric	Orig.	Inline	-O	PT+CSE
base.executedInstructions.value	445.13 M	287.86 M	280.41 M	282.08 M
size.appRun.value	1008	1449	1417	1425
size.appHot.value	330	683	662	663
polymorphism.appInvokeDensity.value	116.17	10.84	11.13	11.06
polymorphism.appReceiverPolyDensityCalls.value	0	0	0	0
polymorphism.appTargetArity.bin(1)	1	1	1	1
pointer.appFieldAccessDensity.value	126.7	196.1	200.9	177.8
Interpreter (runtime)	51.40	33.38	32.42	32.17
JIT-noinlining (runtime)	11.06	8.61	8.67	8.64
(compile-time)	0.070	0.059	0.062	0.060
(compiled-bytes)	3588	4004	3952	3924
JIT (runtime)	8.81	8.21	8.25	8.23
(compile-time)	0.105	0.073	0.073	0.072
(compiled-bytes)	6591	4803	4751	4723

Table 10: Dynamic Metrics and Runtime Measurements for the Voronoi Benchmark

ported by the benchmark. The JIT compile time and compiled size are the average of five runs, as reported by Sun's VM using the `-XX:+CITime` option.

On the bottom part of Table 10 we give runtime measurements for three configurations of the VM. The first configuration, labelled *Interpreter*, runs only in interpretive mode (`java -Xint`). The second configuration, labelled *JIT-noinlining*, uses the ordinary mixed mode VM, but the JIT has inlining disabled (`java -client -XX:MaxInlineSize=0 -XX:FreqInlineSize=0`). The third configuration, labelled *JIT*, is the normal mixed mode VM using its defaults (`java -client`).

The results from the interpreter are easiest to analyze since the interpreter does not perform any optimizations of its own and there is no overhead due to JIT compilation. These results follow the dynamic metrics very closely. The *Orig.* version takes 51.40 seconds, whereas the *Inline* version is much faster, executing in 33.38 seconds. The *-O* version shows a small improvement, with an execution time of 32.4 seconds, which corresponds quite well with the small improvement in executed instructions that we saw in our metrics. The *PTR+CSE* shows a very slight improvement, executing in 32.17 seconds. This shows that there is a benefit to removing the field instructions, even though it executes more instructions overall.

The result from the JIT-noninlining configuration again show that the statically-inlined version of our benchmark (column *Inline*) executes much faster, 8.61 versus 11.06 seconds. However, the *-O* and *PT+CSE* versions have no significant impact (even a slight negative impact) on runtime. This is probably because the JIT optimizations and the static optimizations negatively affect each other. However, note that the amount of compiled code does go down slightly.

When run with the ordinary *JIT* configuration, we note that the JIT inliner is quite effective, giving an execution time of 8.81 versus 11.06 when the JIT inliner is turned off. This indicates that two different inliners (our static inliner and the JIT dynamic inliner) work very well for this benchmark (as predicted by looking at our dynamic metrics). However, note that the JIT inliner does pay a price in compiling more code (6591 bytes vs 3588 bytes for when the inliner is off). Recall that we also saw this effect in our metrics, where size.appHot.value doubled after applying our static inlining. It is also interesting to note that the JIT inliner (row *JIT (runtime)*) and the static inliner (column *Inline*) actually combine to give the overall best result. The runtime is the best (8.21 seconds), the JIT compile time is very reasonable (0.072 seconds), and the amount of compiled code is quite small (4803 bytes).

8. SYSTEM FOR COLLECTING DYNAMIC METRICS

An essential part of any metric development process being empirical validation of the data, we required a way of easily computing metrics for a variety of Java benchmark programs. This led to the development of a new tool, *J, which is designed to allow us to quickly implement and test dynamic metrics for Java applications. This section describes the design objectives, the implementation and the future work of our new framework in detail.

8.1 Design Objectives

Computing dynamic metrics appears to be a deceptively simple task. In fact, the huge amount of data that has to be processed constitutes a problem by itself. Because we were aiming at developing an offline analysis tool, performance was not a critical issue, but ensuring the tool works in reasonable and practical time was nevertheless not a trivial endeavour. The main design objective that influenced the development of *J was however flexibility—we needed a tool which would let us investigate dynamic metrics with a high level of freedom.

We also needed a way to easily visualize and manipulate the collected data in order to study the dynamic metrics. We needed a system which would make it easy for us to share, organize, query and compare the results obtained from *J.

Figure 1 presents an overview of the entire system. We describe all of the components next.

8.2 Design

The *J tool itself consists of two major parts: a Java Virtual Machine Profiler Interface (JVMPI) agent which generates event traces for the programs to be analyzed, and a Java back-end which processes the event traces and computes the values for the various metrics. This design allows the two ends of the framework to be used independently. A third component is the web interface that allows users to view the results in a friendly interface, allowing them to manipulate the data in various ways in order to make the most out of the computed results.

The JVMPI was selected as a source of trace data primarily because of the ease with which it is possible to obtain specific information about the run-time behaviour of a program, and also because it is compatible with a number of commercial virtual machine implementations. Unfortunately, there are a number of limitations that are imposed by using the JVMPI for collecting data, the most serious of which being the fact that it is currently not possible to obtain information about the state of the execution stack using this approach. Within JVMPI the only solution to this problem is to simulate the entire execution, which imposes far too much over-

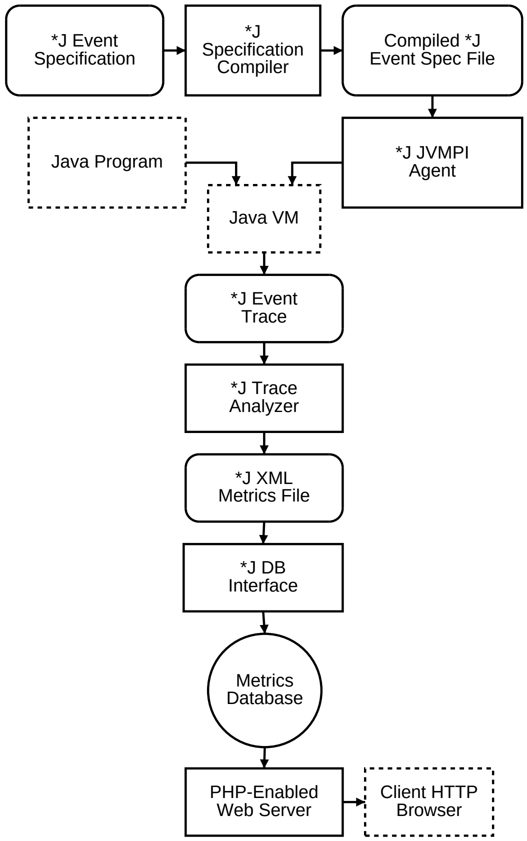

Figure 1: System for collecting metrics (dashed boxes represent components that are not part of the system, but are displayed for completeness)

head for our purposes. JVMPI also has a complex specification, including a variety of restrictions on agent behaviour in certain circumstances: producing a correct agent is not necessarily trivial.

8.2.1 The JVMPI Agent

The main responsibility of the *J JVMPI agent is to gather the actual profile data; a secondary goal is to ensure the trace size is manageable. We note that although the JVMPI interface is well-defined, it is not always well-implemented, and some amount of programming effort is required to ensure the data gathered is complete. For instance, the agent keeps track of all entity identifiers, and explicitly requests events from the JVMPI interface when it encounters an event containing an unknown ID (these can occur for events that happen prior to JVMPI initialization). To reduce the size of the trace file, the agent will collapse several instruction events together when they correspond to a contiguous sequence in the method's code; this is then further reduced by using a binary trace output format. Using these combined strategies complete traces for the SPEC benchmarks (using size 100) can easily be stored on a regular hard disk, as they only require several gigabytes of memory.

The agent also allows the user to specify which events and which of their fields are to be recorded in the trace file using a simple domain-specific language, which is then compiled to a compact binary representation and included in the header of the trace. This ensures that any consumer of the trace file will be able to tell exactly what information is contained in the trace, and determine if all of its requirements are fulfilled before proceeding to the analysis.

8.2.2 The Analysis Back-End

*J's back-end is implemented in Java, and thus benefits from an object-oriented design. Event objects are created from the event trace file and passed to a sequence of *operations*. Operations in the sequence receive the event objects in a well-defined and fixed order, and are free to modify them without restrictions, or even to stop the processing of an event and restart with the next one at the beginning of the processing chain. This allows a great level of flexibility because specific operations can provide services for the subsequent ones in the processing chain, essentially "preparing" the event for further processing. This allows for a very modular design. In order for this to work properly, operations can specify dependencies between each other, so that disabling a particular operation will also disable all of the other operations for which it provides a service. The standard operation library that is part of *J includes built-in services that will manage JVMPI identifiers, modify the events corresponding to an instruction to include the bytecode information and filter an event based on the name of its associated class, to only name a few.

In order to make it easy to group operations together, *J provides *packs*, which are essentially containers. A pack can contain any number of other packs or operations. This allows the creation of a hierarchical organization of the operations, which is very convenient when working with the tool from the command line. For example, the pack which contains all of the operations that compute the various metrics can be enabled or disabled using one simple command. The order in which operations are added to the pack uniquely determines the order in which they appear in the processing chain. Conceptually, there exits a "super-pack" which contains all other packs and sends each event in the trace to each of the elements that it contains. These elements can, in turn, dispatch the event if they are packs themselves or proceed with the computation if they are not. However, for efficiency concerns, the hierarchy is first flattened into a processing chain. Also, each operation is required to specify ahead of time which kinds of events it wants to receive. These dependencies can be of two kind: required or optional. An operation which has unmet required dependencies will be automatically disabled, which is not the case for unmet optional dependencies. The way in which the operations specify their event dependencies is in practice is more versatile, but the technical details are beyond the scope of this paper. For example, *J allows specifying alternate events in the case where a dependency is not met, or even allows specifying the dependencies at the level of the fields of particular events.

*J provides a `MetricAnalysis` class that every metric-computing module should extend. `MetricAnalysis` is a specialized version of the generic operation class, and takes care of most of the work that is common to all metric computations, such as outputting the results in XML form. This allows us to quickly implement new metrics, and allows the programmer to only focus on the code that is specific to the particular metric computations.

Every operation has a unique, fully-qualified name that is built from the name of the operation, along with the names of all packs to which it belongs. This is identical to fully-qualified class names in Java. So, an operation `op` which is a member of a pack `subpack`, itself a member of the pack `mainpack` will be identified by the string `mainpack.subpack.op`. This is necessary in order to control individual operations using the command line interface. *J was designed so that it would be easy to automate the process of gathering metrics using scripts, and thus allows each pack or operation to specify a list of options that it accepts from the command line.

A significant effort has also been put into making the analysis back-end require no more than a single pass on the trace file, making it usable using FIFO special files (or pipes) when the size of the trace is too large to be stored on disk.

8.3 The Web Interface

In order to allow multiple users to be able to use and share metrics data, a web interface has been set up. It allows us to look at metrics or benchmarks individually, but has a lot of more advanced features such as benchmark comparisons, complex search using custom queries. It can be used to easily build tables which summarize the results, and supports sorting features on almost every page.

This website is almost completely generated dynamically by PHP scripts, which allows the distribution of metrics results within seconds. Using *J's database interface, the metrics results files can be automatically parsed and inserted into the database automatically, making the sharing of metrics data very quick and easy.

Using a web interface to view the metrics data has the inherent advantage of being accessible by a large number of users and can constitute the basis of a benchmark knowledge base. In fact, the required facilities are in place to allow users to browse through the metrics, selecting benchmarks as they find interesting ones using a "shopping cart" approach. The benchmarks can then be automatically packaged and downloaded in various compressed archive formats.

9. RELATED WORK

In developing our dynamic metrics we first studied a large body of literature for static metrics, many of which are covered in Fenton and Pfleeger's book [19]. Although some static metrics have a use for compiler developers (for example, a normalized measure of static code size measures the size of the input to an optimizer), we found that many static metrics were somewhat ill-defined, and that static metrics did not capture program behaviour that may be of interest to compiler developers.

We then searched the recent compiler publications to get a feel for the types of dynamic metrics that would be useful, and also the sorts of dynamic measurements already in common use in the field. Thus, our work is both a formalization of many familiar concepts and a development of some new concepts and metrics.

In our literature overview we found that dominant data structures and data types are usually identified by hand. Although most researchers will give relevant qualitative descriptions of the benchmarks in their test suite (floating point, array-based etc), terminology is not standard and categorization is rarely justified quantitatively. Pointer-based programs, however, receive more direct attention. Average size of points-to sets are computed by several researchers in order to show efficacy of pointer-analysis algorithms [11, 13, 21]; the symmetric requirements of showing points-to and points-from are argued in [10].

Dynamic memory allocation is actually a very well studied area, and researchers in the garbage collection community have made many studies about the dynamic behaviour of allocations. In this paper we have tried to distill out some of the most common measurements and report them as meaningful metrics (as opposed to profiles).

Size metrics in the literature are typically based on a static measurement of program size, often lines of code or size of the executable. We did not come across size metrics which are based on the number of instructions touched during execution. Program structure and complexity in terms of control instructions are often reported as a side-effect of hardware branch prediction studies [4, 32, 9, 31, 17]. Unfortunately, the missrates are usually incomparable, since the predictors use limited tables and therefore include capacity misses, distorting the metric.

Polymorphism metrics can be found in three areas: studies using static compiler analysis to de-virtualize object-oriented programs [14, 29, 3], virtual machine implementation papers reporting inline cache miss rates [15, 30], and indirect branch prediction studies reporting branch target buffer miss rates for object-oriented programs [31, 17]. The latter are also usually distorted by limited branch target buffer sizes.

Concurrency is rarely measured dynamically, though some researchers do measure number of threads started [5]. Other metrics we present for measuring concurrency are unique. Measurement of synchronizations is considerably more common, if generally consisting of absolute counts of synchronization operations [7, 20, 27]. More detailed breakdowns are sometimes given; e.g., in [6].

Since the SPECjvm98 benchmarks appear to drive a lot of the development and evaluation of new compiler techniques, several groups have made specific studies of these benchmarks. For example, Dieckmann and Hölzle have presented a detailed study of the allocation behaviour of SPECjvm98 benchmarks [16]. In this paper they studied heap size, object lifetimes, and various ways of looking at the heap composition. The work by Shuf et. al also looked at characterizing the memory behaviour of Java Workloads, concentrating, on the actual memory performance of a particular JVM implementation and evaluating the potential for various compiler optimizations like field reordering [28]. Li et. al. presented a complete system simulation to characterize the SPECjvm98 benchmarks in terms of low-level execution profiles such as how much time is spent in the kernel and the behaviour of the TLB [25].

All of these studies are very interesting and provide more detailed and low-level information than our high-level dynamic metrics. Our intent in designing the high-level dynamic metrics was to provide a relatively few number of data points to help researchers find those programs with interesting dynamic behaviour. Once found, more detailed studies are most certainly useful. We also hope that by providing standardized dynamic metrics for programs outside of the SPECjvm98 suite of programs, we can help to expand the number of programs used to evaluate compiler optimizations for Java.

Daly et. al. performed a bytecode level analysis of the Java Grande benchmarks which is somewhat in the same spirit as our work, in the sense that they were interested in platform independent analysis of benchmarks [12]. Their analysis concentrated mostly on finding different distributions of instructions. For example, how many method calls/bytecodes executed in the the application, and how many in the Java API library, and what is the frequency of executions of various Java bytecodes. Our focus is also on platform independent analysis, but we are concentrating on developing a wide-variety of metrics that can be used to find different high-level behaviours of programs.

A recent article by Aggarwal et al [2] presents a system for computing dynamic metrics related to finding the most frequently executed modules, applied to C programs. Our metrics are intended to be more comprehensive, and include metrics specific to object-oriented programs. Nevertheless, they also emphasize the importance of *dynamic* metrics when examining programs.

10. FUTURE WORK

We plan to continue to extend our work in several ways. First, we need to find an alternate source of profiling information which would allow us to overcome the limitations of JVMPI. In particular, we plan to implement a general-purpose profiling framework within SableVM, a free, open-source and portable Java virtual machine. This profiling framework is intended to be more versatile than what JVMPI currently provides. It will also allow for a greater flexibility in terms of the information that can be recorded. For instance, all of the memory-related metrics that we cannot compute yet would be easy to compute using such a framework because they are very difficult (if not impossible) to compute within the context of a closed-source virtual machine. We also feel that such a profiling framework may be useful to other researchers who also often need to instrument a Java virtual machine in order to obtain dynamic information.

Also, we want to continue to extend our set of metrics. Although it already contains more metrics that what we are able to cover in this paper (more details available from our webpage), it does not yet fulfill all of our needs. We have concentrated so far on developing metrics that allow to characterize benchmark programs and identify the ones which are most likely to benefit from certain program transformations or optimizations. However, we found it is not always easy to measure the difference in executions between two benchmarks that have different characteristics. In particular, if an transformation modifies the number of executed instructions significantly, then all of the densities will also change for the transformed version of the benchmark, even though they might not be directly related to the transformation. We will therefore develop a set of metrics which will allow us to perform such comparisons. Two main avenues can be considered at this time: one possibility is to design dynamic metrics which are more absolute in nature, or that are relative to more stable quantities. Another possibility would be to define dynamic metrics that are relative to another program execution.

11. CONCLUSIONS

Static metrics are quite common in the compiler research literature, and provide important, and relatively easily-computed information. In this paper we have focused on the more difficult problem of computing dynamic metrics as a means of assessing the actual runtime behaviour of a program. This dynamic information can result in a much more relevant view of the program to compiler and runtime optimization developers.

We have defined five families of dynamic metrics which characterize a program's runtime behaviour in terms of size and control structure, data structures, polymorphism, memory use, and concurrency and synchronization. These metrics were designed with the goals of being unambiguous, dynamic, robust, discriminative, and machine-independent, and are meant to quantitatively characterize a benchmark in ways relevant to compiler and runtime developers.

We have built a metrics collection system around JVMPI that allows us to measure most of the defined metrics, and to distill a concise subset that characterizes a program. We have demonstrated that the metrics we do compute can be easily used to evaluate and compare benchmarks; a compiler researcher should be able to tell at a glance if a particular program exhibits the kind of behaviour he/she is interested in. We have also shown that our metrics do indeed capture qualities that are appropriate for compiler optimization developers; by inspecting metrics one can find out whether a particular benchmark will respond to a specific optimization, and also show the variety of effects (positive and negative) of a transformation.

Researchers are invited to visit the website `http://www.-sable.mcgill.ca/metrics` to inspect and contribute to the full range of benchmarks and metrics, which is being continuously updated. We especially welcome new metric or benchmark suggestions.

Our experience in defining and producing these metrics has resulted in a number of insights:

- Collecting dynamic metrics takes significant effort. It usually fairly easy to verify the correctness of a static metric by inspection of the program source. For a dynamic metric, code inspection does not suffice, and verification can be a difficult process. We hope that the numbers contained in this report will help others to verify their own collection of program behaviour statistics.
- In order to be discriminative, dynamic metrics should ignore the effects produced by standard libraries and focus on application code. The Java standard libraries form such a large part of execution that it is as if one always runs a separate application, distorting the numbers; for small benchmarks, startup code can easily account for a majority of code executed. Understanding and controlling the impact of startup code is clearly an important aspect of optimization design.
- Robust metrics are difficult to define. Startup and library code have a significant impact, and of course different program inputs can easily exercise different parts of the program code. Given this, many of our metrics are surprisingly robust, and stable across a variety of inputs. Even when applying compiler optimizations, only the metrics directly measuring optimization-related behaviour exhibited large changes.

We believe there is a tremendous need for rigorously defined program metrics. Many studies we examined reported various numbers that were either not well-defined or could easily be skewed by small changes in program input or measurement style. We intend our work here to be foundational, giving specific and unambiguous metrics to the compiler community. It is our hope that this will inspire other researchers to describe their benchmarks with more rigour, and will also provide validity for the qualitative judgements researchers employ when collecting a benchmark suite

12. REFERENCES

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. Technical Report TR-99-76, Sun Microsystems, 1999.

[2] 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.

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, July 1996. Springer.

[4] A.N.Eden and T.Mudge. The YAGS branch prediction scheme. In *Proceedings of the International Symposium on Microarchitecture*, pages 69–77, November 1998.

[5] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 92–103. ACM Press, 2001.

[6] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 258–268. ACM Press, 1998.

[7] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM Press, 1999.

[8] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java controller. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, September 2001.

[9] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the International Symposium on Computer Architecture*, pages 274–283, June 1997.

[10] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 57–69. ACM Press, 2000.

[11] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 191–202. ACM Press, 2001.

[12] 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.

[13] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2000.

[14] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, Aug 1995. Springer.

[15] L. P. Deutsch. Efficient implementation of the Smalltalk-80 system. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[16] 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.

[17] K. Driesen and U. Hölzle. Multi-stage cascaded prediction. In *EuroPar '99 Conference Proceedings, LNCS 1685*, pages 1312–1321, September 1999.

[18] Karel Driesen. *Efficient Polymorphic Calls*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach*. PWS Publishing Company, 1997.

[20] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 334–344. ACM Press, 2000.

[21] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 47–58. ACM Press, 2001.

[22] John L. Hennessy and David A.Patterson. *Computer Architecture: A Quantitative Approach (Third Edition)*. Morgan Kaufmann, San Francisco, 2002.

[23] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 2002 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 142–160, November 2002.

[24] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 156–167. ACM Press, 2001.

[25] Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the 14th International Conference on Supercomputing*, pages 22–33. ACM Press, 2000.

[26] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *CC'02: International Conference on Compiler Construction*, number 2304 in LNCS, pages 325–341, 2002.

[27] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 208–218. ACM Press, 2000.

[28] 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.

[29] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 264–280. ACM Press, 2000.

[30] David Ungar. *The Design and evaluation of a high-performance Smalltalk System*. MIT Press, Cambridge, 1987.

[31] N. Vijaykrishnan and N.Ranganathan. Tuning branch predictors to support virtual method invocation in Java. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies ans Systems*, May 1999.

[32] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[33] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 134–144. ACM Press, 2000.

Metric	CA	COEFF	EMPTY	HELLO	JLEX	LPACK	RC	SBLCC	SOOT	TCOM	VOLCL	VOLS	COMP
size.appLoadedClasses.value	1	6	1	1	24	1	4	304	531	13	19	38	22
size.appLoad.value	344	2374	4	7	14243	1056	395	42606	45111	914	4149	9861	6555
size.appRun.value	293	975	0	4	10465	749	320	30705	25666	671	2389	4843	5084
size.appHot.value	59	57	0	4	758	59	36	1099	2549	360	551	1285	396
size.appHot.percentile	20%	6%		100%	7%	8%	11%	4%	10%	54%	23%	27%	8%
size.loadedClasses.value	275	286	275	275	310	278	282	663	818	322	397	551	310
size.load.value	72158	80292	71818	71821	87405	77932	72460	146253	126566	91228	88494	149940	90762
size.run.value	8225	12880	7343	7793	21326	10698	8530	49488	37852	22065	19658	52024	14514
size.hot.value	920	115	1014	1038	420	115	1251	1758	3097	713	1238	2412	396
size.hot.percentile	11%	1%	14%	13%	2%	1%	15%	4%	8%	3%	6%	5%	3%
data.appArrayDensity.value	139.9	160.4		0.0	53.7	157.8	48.0	38.9	39.1	0.0	15.5	25.6	52.2
data.appCharArrayDensity.value	15.5	0.0		0.0	1.4	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.0
data.appNumArrayDensity.value	124.3	79.5		0.0	47.0	148.4	0.0	11.2	19.5	0.0	0.0	0.0	52.2
data.appRefArrayDensity.value	0.0	80.7		0.0	0.6	9.4	47.9	13.3	9.3	0.0	6.4	11.2	0.0
data.arrayDensity.value	30.9	150.9	73.5	73.3	24.5	152.2	45.9	43.6	44.6	53.1	60.6	62.9	52.2
data.charArrayDensity.value	10.2	1.5	32.5	32.6	1.2	2.1	16.6	6.5	6.2	10.8	21.3	24.2	0.0
data.fbatDensity.value	0.4	203.0	2.1	2.0	0.0	286.2	0.6	0.0	0.7	0.0	0.3	0.4	0.0
data.numArrayDensity.value	14.6	75.0	35.0	34.7	9.5	140.9	13.2	9.4	13.0	5.1	22.3	21.2	52.1
data.refArrayDensity.value	0.4	73.8	1.9	1.9	12.5	8.9	12.6	15.6	13.9	34.8	0.5	2.5	0.0
concurrency.lock.percentile	32%	10%	67%	65%	28%	70%	36%	6%	11%	26%	12%	21%	57%
concurrency.lockContended.percentile					90%		33%	87%	88%	19%	63%	54%	
concurrency.lockContendedDensity.value	0.00	0.00	0.00	0.00	0.00	0.00	0.45	0.00	0.00	0.09	0.03	0.02	0.00
concurrency.lockDensity.value	4.83	0.22	0.16	0.19	0.12	0.02	2.02	1.29	0.35	1.16	0.74	0.57	0.00
pointer.appFieldAccessDensity.value	219.0	99.6		250.0	129.2	0.0	175.0	164.9	190.9	136.2	169.0	157.9	189.6
pointer.appNonrefFieldAccessDensity.value	218.7	66.7		0.0	95.5	0.0	111.6	58.6	102.8	59.5	85.0	77.8	145.9
pointer.appRefFieldAccessDensity.value	0.3	32.9		250.0	33.7	0.0	63.4	106.2	88.1	76.6	84.1	80.1	43.7
pointer.fieldAccessDensity.value	148.9	97.7	52.9	53.4	167.9	3.6	109.8	157.3	161.4	111.1	109.8	105.0	189.6
pointer.nonrefFieldAccessDensity.value	123.5	66.7	46.6	47.0	127.1	3.2	89.3	84.9	95.2	103.6	96.6	89.9	145.9
pointer.refFieldAccessDensity.value	25.4	31.0	6.3	6.4	40.8	0.4	20.4	72.4	66.2	7.5	13.2	15.1	43.7
memory.averageAppObjectSize.value	44.0	20.1	56.0	56.0	25.5	36.0	52.6	23.7	20.8	26.7	34.4	31.9	29.1
memory.appobjectSize.bin (8)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin (16)	0%	66%	0%	0%	13%	50%	29%	49%	61%	0%	0%	6%	42%
memory.appobjectSize.bin (24)	0%	33%	0%	0%	72%	0%	0%	22%	30%	67%	9%	23%	42%
memory.appobjectSize.bin (32)	50%	0%	0%	0%	0%	0%	7%	19%	1%	33%	77%	61%	0%
memory.appobjectSize.bin (40)	0%	0%	0%	0%	15%	0%	0%	9%	8%	0%	3%	3%	0%
memory.appobjectSize.bin (48-72)	50%	0%	100%	100%	0%	50%	57%	1%	1%	0%	9%	7%	8%
memory.appobjectSize.bin (80-136)	0%	0%	0%	0%	0%	0%	7%	0%	0%	0%	2%	1%	8%
memory.appobjectSize.bin (144-392)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin (400+)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
memory.byteAllocationDensity.value	442.2	108.6	1750.4	1734.2	301.7	148.7	553.2	344.8	273.7	124.2	240.9	225.7	11.1
memory.byteAppAllocationDensity.value	0.7	0.3		14000.0	6.4	0.0	2.5	71.0	148.5	155.3	232.0	169.7	0.0
memory.objectAllocationDensity.value	4.3	0.8	9.1	9.1	6.1	0.6	3.9	5.5	7.9	1.6	3.6	3.4	0.0
memory.objectAppAllocationDensity.value	0.0	0.0		250.0	0.3	0.0	0.0	3.0	7.1	5.8	6.7	5.3	0.0
polymorphism.appCallSites.value	8	85	0	1	636	39	33	2619	3141	82	288	526	54
polymorphism.callSites.value	482	677	437	472	1267	530	536	3978	3766	967	1580	2592	606
polymorphism.appInvokeDensity.value	16.0	66.0		250.0	27.6	1.3	125.3	69.2	69.7	134.2	122.1	120.9	16.5
polymorphism.appReceiverArity.bin (1)	100%	100%		100%	100%	100%	100%	92%	93%	94%	99%	98%	98%
polymorphism.appReceiverArity.bin (2)	0%	0%		0%	0%	0%	0%	3%	3%	6%	0%	1%	2%
polymorphism.appReceiverArity.bin (3+)	0%	0%		0%	0%	0%	0%	5%	4%	0%	1%	1%	0%
polymorphism.appReceiverArityCalls.bin (1)	100%	100%		100%	100%	100%	100%	94%	78%	93%	89%	85%	100%
polymorphism.appReceiverArityCalls.bin (2)	0%	0%		0%	0%	0%	0%	0%	16%	7%	0%	4%	0%
polymorphism.appReceiverArityCalls.bin (3+)	0%	0%		0%	0%	0%	0%	5%	6%	0%	11%	10%	0%
polymorphism.appReceiverCacheMissRate.value	0%	0%		100%	0%	0%	0%	0%	4%	3%	5%	6%	0%
polymorphism.appTargetArity.bin (1)	100%	100%		100%	100%	100%	100%	97%	95%	99%	99%	99%	98%
polymorphism.appTargetArity.bin (2)	0%	0%		0%	0%	0%	0%	3%	2%	1%	0%	0%	2%
polymorphism.appTargetArity.bin (3+)	0%	0%		0%	0%	0%	0%	1%	3%	0%	1%	1%	0%
polymorphism.appTargetArityCalls.bin (1)	100%	100%		100%	100%	100%	100%	95%	84%	98%	95%	94%	100%
polymorphism.appTargetArityCalls.bin (2)	0%	0%		0%	0%	0%	0%	0%	14%	2%	0%	1%	0%
polymorphism.appTargetArityCalls.bin (3+)	0%	0%		0%	0%	0%	0%	5%	3%	0%	5%	5%	0%
polymorphism.appTargetCacheMissRate.value	0%	0%		100%	0%	0%	0%	0%	3%	0%	3%	3%	0%

Table 11: All metrics.

Metric	JESS	DB	JAVAC	JACK	MTRT	Obh	Obsrt	Oem3d	Ohth	Omst	Operm	Opow	Otsp	Ovor
size.appLoadedClasses.value	158	14	175	66	35	9	2	4	8	6	10	6	2	6
size.appLoad.value	22370	6436	44664	23424	11193	2023	565	713	1004	727	863	1949	955	1783
size.appRun.value	11634	4546	26267	18721	9460	1631	484	563	920	600	785	1854	868	1063
size.appHot.value	476	67	2759	802	754	160	137	42	65	175	393	402	54	351
size.appHot.percentile	4%	2%	11%	4%	8%	10%	28%	8%	7%	29%	50%	22%	6%	33%
size.loadedClasses.value	458	304	471	356	324	285	277	280	283	281	285	281	278	281
size.load.value	111661	91730	133172	107697	99161	77605	76140	76518	76579	76302	76438	77524	76760	77358
size.run.value	22621	14571	37830	28883	20148	11141	9752	9859	10554	10112	9991	10950	10167	10575
size.hot.value	564	152	2258	1475	832	160	137	75	66	186	398	409	110	354
size.hot.percentile	3%	1%	6%	5%	4%	1%	1%	1%	1%	2%	4%	4%	1%	3%
data.appArrayDensity.value	58.1	86.2	15.5	22.5	39.1	105.9	0.0	101.2	4.4	20.5	0.0	97.4	0.0	28.3
data.appCharArrayDensity.value	0.0	0.0	3.5	1.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
data.appNumArrayDensity.value	0.8	2.5	0.4	11.5	3.7	97.6	0.0	3.4	0.0	0.0	0.0	96.5	0.0	0.0
data.appRefArrayDensity.value	53.1	83.7	3.5	2.0	34.9	4.4	0.0	92.6	4.0	20.5	0.0	0.2	0.0	27.7
data.arrayDensity.value	55.2	73.5	37.9	32.8	39.3	105.9	0.1	89.3	4.6	19.6	0.3	93.4	0.8	28.3
data.charArrayDensity.value	0.2	33.8	17.6	6.3	0.7	0.0	0.0	0.1	0.1	0.1	0.1	0.0	0.4	0.0
data.fbatDensity.value	12.1	0.0	0.1	0.2	308.5	245.7	0.0	13.5	6.0	0.0	0.0	474.9	471.5	226.4
data.numArrayDensity.value	0.8	1.0	6.8	4.7	3.9	97.5	0.0	3.1	0.1	0.1	0.1	92.5	0.4	0.0
data.refArrayDensity.value	48.6	38.6	5.6	13.0	34.0	4.4	0.0	81.5	4.0	19.4	0.0	0.2	0.2	27.7
concurrency.lock.percentile	42%	3%	6%	21%	50%	69%	65%	70%	62%	73%	65%	70%	70%	73%
concurrency.lockContended.percentile	33%		82%	100%	93%									
concurrency.lockContendedDensity.value	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
concurrency.lockDensity.value	0.00	2.14	0.23	2.13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pointer.appFieldAccessDensity.value	156.0	129.7	208.3	161.3	158.5	111.2	120.9	115.3	249.7	76.6	103.0	32.8	220.7	120.7
pointer.appNonrefFieldAccessDensity.value	148.1	84.0	132.8	101.9	132.4	100.4	47.5	111.5	81.2	35.7	21.7	32.0	191.2	78.3
pointer.appRefFieldAccessDensity.value	7.8	45.7	75.5	59.5	26.1	10.8	73.4	3.8	168.5	40.9	81.3	0.9	29.6	42.4
pointer.fieldAccessDensity.value	151.2	123.3	163.9	156.6	157.7	111.1	120.8	110.7	249.3	80.1	102.9	31.5	212.4	120.6
pointer.nonrefFieldAccessDensity.value	141.4	101.0	114.1	124.5	131.9	100.3	47.5	107.3	81.1	41.5	21.9	30.6	184.5	78.2
pointer.refFieldAccessDensity.value	9.8	22.2	49.8	32.1	25.8	10.8	73.3	3.4	168.2	38.6	81.0	0.9	27.9	42.3
memory.averageAppObjectSize.value	41.9	1334.2	29.0	31.3	25.0	16.2	24.0	291.1	19.4	25.0	32.0	30.4	48.0	25.6
memory.appobjectSize.bin(8)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin(16)	0%	98%	23%	8%	2%	99%	0%	0%	71%	0%	0%	0%	0%	3%
memory.appobjectSize.bin(24)	20%	0%	22%	2%	89%	0%	100%	0%	14%	100%	0%	48%	0%	75%
memory.appobjectSize.bin(32)	30%	0%	39%	81%	5%	1%	0%	0%	14%	0%	100%	43%	0%	19%
memory.appobjectSize.bin(40)	1%	0%	10%	9%	3%	0%	0%	33%	0%	0%	0%	0%	0%	2%
memory.appobjectSize.bin(48-72)	48%	0%	5%	0%	1%	0%	0%	0%	0%	0%	0%	9%	100%	0%
memory.appobjectSize.bin(80-136)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin(144-392)	0%	0%	0%	0%	0%	0%	0%	11%	0%	0%	0%	0%	0%	0%
memory.appobjectSize.bin(400+)	0%	2%	0%	0%	0%	0%	0%	56%	0%	0%	0%	0%	0%	0%
memory.byteAllocationDensity.value	294.4	24.8	131.8	313.9	93.4	248.7	11.0	35.5	67.4	225.4	101.8	21.5	84.6	84.4
memory.byteAppAllocationDensity.value	174.8	19.1	77.7	19.1	64.0	65.2	5.9	14.9	60.4	137.2	85.9	0.6	15.5	78.6
memory.objectAllocationDensity.value	4.4	0.8	3.2	5.8	3.1	8.0	0.3	0.1	3.1	10.4	2.7	0.6	1.0	3.1
memory.objectAppAllocationDensity.value	4.2	0.0	2.7	0.6	2.6	4.0	0.2	0.1	3.1	5.5	2.7	0.0	0.3	3.1
polymorphism.appCallSites.value	737	128	2617	1124	939	1	36	63	74	60	49	34	56	175
polymorphism.callSites.value	1309	695	3234	1699	1510	6	537	568	579	570	550	531	560	685
polymorphism.appInvokeDensity.value	59.2	80.6	72.3	86.7	126.8	18.7	11.5	3.6	57.1	49.2	45.9	0.5	24.8	110.7
polymorphism.appReceiverArity.bin(1)	98%	100%	78%	98%	94%	97%	100%	100%	100%	100%	69%	100%	100%	100%
polymorphism.appReceiverArity.bin(2)	1%	0%	10%	1%	4%	3%	0%	0%	0%	0%	0%	0%	0%	0%
polymorphism.appReceiverArity.bin(3+)	1%	0%	12%	1%	2%	0%	0%	0%	0%	0%	31%	0%	0%	0%
polymorphism.appReceiverArityCalls.bin(1)	99%	100%	73%	90%	91%	87%	100%	100%	100%	100%	37%	100%	100%	100%
polymorphism.appReceiverArityCalls.bin(2)	0%	0%	15%	10%	7%	13%	0%	0%	0%	0%	0%	0%	0%	0%
polymorphism.appReceiverArityCalls.bin(3+)	18%	0%	12%	0%	2%	0%	0%	0%	0%	0%	63%	0%	0%	0%
polymorphism.appReceiverCacheMissRate.value	0%	0%	7%	3%	0%	3%	0%	0%	0%	0%	41%	0%	0%	0%
polymorphism.appTargetArity.bin(1)	99%	100%	89%	99%	99%	97%	100%	100%	100%	100%	78%	100%	100%	100%
polymorphism.appTargetArity.bin(2)	0%	0%	4%	0%	1%	3%	0%	0%	0%	0%	0%	0%	0%	0%
polymorphism.appTargetArity.bin(3+)	1%	0%	7%	1%	0%	0%	0%	0%	0%	0%	22%	0%	0%	0%
polymorphism.appTargetArityCalls.bin(1)	99%	100%	92%	90%	99%	87%	100%	100%	100%	100%	43%	100%	100%	100%
polymorphism.appTargetArityCalls.bin(2)	0%	0%	2%	10%	1%	13%	0%	0%	0%	0%	0%	0%	0%	0%
polymorphism.appTargetArityCalls.bin(3+)	1%	0%	6%	0%	0.0%	0%	0%	0%	0%	0%	57%	0%	0%	0%
polymorphism.appTargetCacheMissRate.value	0%	0%	3%	3%	0%	3%	0%	0%	0%	0%	38%	0%	0%	0%

Table 12: All metrics.