OBJECTIVE QUANTIFICATION OF PROGRAM BEHAVIOUR USING DYNAMIC METRICS

by

Bruno Dufour

School of Computer Science
McGill University, Montréal

June 2004

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

In order to perform meaningful experiments in optimizing compilation and runtime 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.

In order to make these intuitive notions of program behaviour more concrete and subject to experimental validation, this thesis introduces a methodology to objectively quantify key aspects of program behaviour using dynamic metrics. A set of unambiguous, dynamic, robust and architecture-independent dynamic metrics is defined, and can be used to categorize programs according to their dynamic behaviour in five areas: size, data structures, memory use, polymorphism and concurrency. Each metric is also empirically validated.

A general-purpose, easily extensible dynamic analysis framework has been designed and implemented to gather empirical metric results. This framework consists of three major components. The *profiling agent* collects execution data from a Java virtual machine. The *trace analyzer* performs computations on this data, and the *web interface* presents the result of the analysis in a convenient and user-friendly way.

The utility of the approach as well as selected specific metrics is validated by examining metric data for a number of commonly used benchmarks. Case studies of program transformations and the consequent effects on metric data are also considered. Results show that the information that can be obtained from the metrics not only corresponds well with the intuitive notions of program behaviour, but can also reveal interesting behaviour that would have otherwise required lengthy investigations using more traditional techniques.

Résumé

Afin d'effectuer des expériences signifiantes dans les domaines de compilation avec optimisation et du design de systèmes d'exécution, les chercheurs se basent habituellement sur une suite de programmes de test pertinents à la technique d'optimisation sous étude. De tels programmes sont souvent décrits comme étant numériques, concurrents, orientés objet, ou faisant un usage intensif de la mémoire à partir d'une évaluation qualitative, et dans certains cas avec peu de preuves à l'appui.

Dans le but de rendre ces notions du comportement des programmes basées sur l'intuition plus concrètes et les assujettir à la validation expérimentale, la présente thèse introduit une méthodologie permettant de quantifier d'une façon objective des aspects clés du comportement des programmes en utilisant une métrologie dynamique. Un ensemble de mesures clairement définies, dynamiques, robustes et indépendantes de l'architecture est proposé, et peut être utilisé afin de classifier les programmes en termes de leur comportement dynamique dans cinq catégories : magnitude, structures de données, utilisation de la mémoire, polymorphisme et colatéralité. Chacune des mesures fait aussi l'objet d'une validation empirique.

Un cadre d'applications polyvalent et extensible permettant d'effectuer des analyses dynamiques a été conçu et implémenté afin de calculer des valeurs expérimentales pour les mesures. Ce cadre d'applications est consitué de trois composantes principales. L'*agent de profilage* amasse des données relatives à l'exécution à partir d'une machine virtuelle Java. Le *programme d'analyse de traces* applique des calculs à ces données, et l'*interface web* présente les résultats des analyses d'une façon pratique et conviviale.

L'utilité de cette approche ainsi qu'une sélection de mesures spécifiques sont validées à partir de l'examen des données provenant de plusieurs programmes de tests communément utilisés. Des études de cas traitant de la transformation de programmes et les effects conséquents sur les valeurs obtenues sont aussi traités. Les résultats démontrent que l'information qui est obtenue à partir des mesures dynamiques correspond non seulement bien avec les notions intuitives du comportement des programmes, mais peut aussi révéler la présence de comportements inattendus qui auraient en d'autres cas nécessité des études approfondies à l'aide de techniques plus traditionnelles.

Acknowledgments

Without key contributions from many people, this thesis could not have been written. I am especially grateful to both of my supervisors, Laurie Hendren and Clark Verbrugge, for making the Sable group such a stimulating research environment. Thanks to Laurie for her unequalled enthusiasm and for continually making us want to better ourselves. Thanks to Clark for always being supportive and available, for his most useful advice, as well as for providing additional funding. I would also like to thank Karel Driesen for his valuable input and participation to this work.

I would also like to extend my thanks to the members of the Sable Research Group, with whom I enjoyed sharing ideas. I am especially greatful to Ondřej Lhoták, for his invaluable feedback and suggestions, and for being a constant source of inspiration. Thanks are also due to the current members of the dynamic analysis group, Christopher Goard and Sokhom Pheng, for their constructive ideas. Many current and former members of the group have also directly or indirectly influenced this work through our many discussions: Rhodes Brown, David Bélanger, John Jorgensen, Mark Berndl, Feng Qian and Etienne Gagnon.

Special thanks go to Tobias Simon for implementing the dynamic website for the dynamic metrics project.

This research has been funded in part by the National Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT), and the McGill Faculty of Graduate Studies and Research (FGSR).

Finally, special thanks to my family and friends, to whom I am indebted for their much appreciated support and encouragements.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

1.1 Motivation

The increasing complexity of software systems combined with their constant evolution has led to an undeniable need for program understanding tools and techniques. Such tools have already found their place in many phases of the software development process. Approaches to program understanding vary greatly, and include many well-known and commonly used techniques such as source code navigation, software metrics, reverse engineering, and software visualization. Program understanding techniques can extract information from the system under consideration in one of three ways: by looking at the documentation, by looking at the source code and by running the program [Cor89]. Traditionally, most techniques could either be classified as *static* or *dynamic*, depending on whether they made use of the source code or execution data, respectively. However, hybrid techniques combining both sources of data are starting to emerge [Ern03].

Different program understanding techniques naturally possess different strengths, making them suitable for different tasks. For instance, program visualization techniques excel at allowing a user to process multiple aspects of the data at once. However, it suffers from poor scalability; in most cases, visualizing large amounts of data quickly becomes very difficult as the total size of the system increases. Software metrics, on the other hand, possess a much better scalability, but achieve this at the cost of losing some of the information that is available. Metric results are, however, much easier to compare than entire graphs.

Program understanding techniques have applications beyond the realm of software maintenance. Understanding the dynamic behaviour of programs is in fact an important aspect in developing effective new strategies for optimizing compilers and runtime systems. Techniques proposed in these areas are often presented as being aimed at programs which are either *numeric*, *loop-intensive*, *pointer-intensive*, *memory-intensive*, *polymorphic*, *concurrent*, etc. However, there appears to be no well-established standard way to determine if a program fits into one or more of these categories. Such categorizations are often based on a qualitative appraisal of the source code of programs.

Moreover, a survey of key conferences in the area of programming languages for the last few years has revealed that many quantitative summaries of benchmarks and results also focus on static program measurements; e.g., number of lines of code (LOC), number of methods, number of loops transformed, number of inlinable methods, and so on. However, because the categories relate to the program's *behaviour*, basing the categorization of benchmark programs on static measurements is not desirable.

This thesis therefore aims to study dynamic metrics as a means of objectively quantifying software behaviour by performing an offline, post-mortem analysis of Java programs. This approach has a wide range of possible applications. Most obviously, it can simply be used to obtain a high-level overview of a program's behaviour. Specific examples of such an application include selecting benchmarks which exhibit particular runtime characteristics, or using dynamic metrics as part of an incremental approach to program understanding. For instance, dynamic metrics can be used to identify interesting aspects of the program's behaviour in order to guide more detailed program understanding techniques such as software visualization. Because the metrics provide summarized behavioural information, they can also be used to quickly identify a specific behavioural pattern, which can then be investigated more closely using common program understanding tools. Dynamic metrics are also effective in quantifying changes in behaviour, such as those observed before and after applying manual or compiler optimizations to a program. Moreover, dynamic metrics can be used not only to measure their effects but also to guide such optimizations.

1.2 Contributions

While dynamic metrics have many potential applications, little research has been done on the subject. This thesis aims to be one of the first comprehensive studies of the subject. As such, the contributions of this research are threefold. They consist of the design of a set of dynamic metrics, the design and implementation of a framework for computing the metrics for Java programs, and a database of empirical metric results for a set of commonly used benchmarks. These contributions are described in the following subsections.

1.2.1 Dynamic Metrics

One of the main objectives of this thesis is to provide a methodology for designing and computing dynamic metrics that can be used to measure relevant runtime properties of programs. To that end, this thesis provides the following key contributions:

- We provide a discussion of the general requirements for good dynamic metrics and outline some problems that are frequently faced when trying to devise objective measurements of a dynamic nature.
- We provide an analysis of the different ways of presenting metrics, and describe three general kinds of dynamic metrics: *values*, *percentiles* and *bins*.
- We provide a discussion of five groups of specific metrics that are particularly relevant to compiler developers and runtime developers: *size*, *data structures*, *memory use*, *concurrency* and *polymorphism*.
- We provide unambiguous descriptions of dynamic metrics for each category, along with some specific examples from benchmark programs, with the ultimate goal of establishing some standard metrics that could be used for quantitative analysis of benchmark programs in compiler research.

1.2.2 **J* Framework

Computing dynamic metrics appears to be a deceptively simple task. In practice, however, the huge amount of data that has to be processed constitutes a problem by itself. Several other factors, such as source of data and profiler-specific issues, contribute to making the computation of dynamic metrics a non-trivial task. This difficulty is in fact likely to be one of the most important factors which contribute to the widespread use of static metrics. In order to develop a set of interesting dynamic metrics, it was however necessary not only to be able to compute them, but also to be able to experiment with metrics with a high level of freedom. To solve this problem, the **J* framework was developed. **J* allows new dynamic metrics to be implemented, tested and validated quickly and easily.

The **J* framework consists of two separate tools:

- **Profiling agent**: Execution data is collected by the **J* agent and stored in an execution trace file. Trace files essentially consist of a stream of serialized runtime events, along with some meta-data which describes the contents of the trace and its encoding.

- **Trace analyzer**: The **J* analyzer processes trace files produced by the agent and performs the required dynamic analyses.

Data collection

The primary data collection component of **J* is a profiler based upon the Java Virtual Machine Profiler Interface (JVMPI). This profiling agent hooks into an executing Java Virtual Machine (JVM) and receives runtime events.

The design of the **J* agent was influenced by many challenges, including:

- **Flexibility**: Because not all analyses require the same information, the agent generates traces using an adaptive format which can be tailored using *event specifications*.

- **Trace size**: When frequent events are recorded (e.g., execution of Java bytecode instructions), trace file reduction strategies become necessary. The **J* agent supports trace compression techniques such as bytecode prediction. It is also capable of outputting trace files to a pipe, thereby avoiding trace storage altogether.

- **JVMPI-related issues**: The implementation of the current JVMPI has a number of problems which must be solved or worked around by the profiler. The **J* agent includes, among other things, an entity resolution mechanism which greatly improves the accuracy of the collected data during the startup phase of the JVM.

Dynamic analysis

The trace analysis component of **J* reads events sequentially from a trace file and applies a series of *operations* on them.

Various factors influenced the design of the trace analyzer, such as:

- **Flexibility**: The key to the flexibility of **J* is its pipelined design, which allows results of some computations to be used by subsequent ones. This design greatly facilitates the implementation of new dynamic analyses by providing most of the necessary support as part of a standard operation library. This design also minimizes direct interactions between operations, therefore allowing them to be easily swapped in and out of the pipeline.

- **Extensibility**: In order to facilitate the addition of new operations to the framework, **J* allows all common trace entities, (e.g., methods, classes, bytecode instructions, ...) to act as *storage containers* at the class level. This makes it possible for new operations to associate data with such entities without having to modify the entity classes directly. For example, it is possible for a metric analysis to associate a counter with every method in the program to record the number of times that it was entered without requiring the use of a hash table.

- **Performance and efficiency**: Although the analysis is performed offline, the amount of data that has to be processed makes performance and efficiency issues very relevant. Making the computations feasible for non-trivial programs was not a trivial endeavour, and required careful design and implementation decisions. Caching and pooling strategies, for example, are used extensively and allow the tool to work in a reasonable and practical time. The memory requirements of **J* are also well within acceptable limits, and scale well with increasing trace sizes.

- **Ease of use**: Because the analysis component of **J* is implemented in Java, it benefits from a strongly object-oriented design. As a result, the implementation of new operations is greatly simplified by using inheritance. Because **J* was primarily designed as a metric computation tool, an emphasis was put on making most metric-related tasks as easy as possible to implement. A majority of metric computing modules can be implemented in under 150 lines of Java code. The more complex ones only require about 400-500 lines of code.

- **Trace size**: significant efforts have been invested in making the analyzer work in a single pass on the trace file, thereby freeing it from the necessity to store the trace data.

1.2.3 Empirical Results

Using the **J* framework, a database of dynamic metric results has been collected for a set of commonly used benchmarks. While this database of metric data constitutes a contribution in itself, experiments were conducted in the following key areas:

- **Program understanding**: We illustrate how the metrics relate to our intuitive qualitative notions of program behaviour, and show how they reveal behaviour that would normally be difficult to ascertain without reverting to a lengthy benchmark investigation.

- **Manual optimizations**: We show how dynamic metrics can help identify bottlenecks in applications, and provide examples of cases where metrics were used to locate and eliminate such problems.

- **Compiler optimizations**: We demonstrate how dynamic metrics can be used to both guide and evaluate the effect of compiler optimizations through real examples.

Results from these case studies 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.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 is a survey of the related work. Chapter 3 defines the set of dynamic metrics. Chapter 4 gives a detailed description of the implementation of the **J* framework. Chapter 5 presents empirical results as well as case studies which illustrate some potential applications of dynamic metrics. Chapter 6 concludes this work, and suggests future directions for research.

Chapter 2
Related Work

This chapter presents previous work on program understanding and software metrics. Section 2.1 is a survey of static software metrics. The survey is not intended as a comprehensive review but is instead meant to provide an overview of the most influential work in the area. Readers interested in a more thorough treatment of the subject are directed, as a starting point, towards Fenton and Pfleeger's book [FP97], or surveys such as [Rig96] or [FN99]. Section 2.2 presents previous dynamic analyses of benchmark suites. Section 2.3 is a survey of the work done so far on dynamic metrics. Section 2.4 discusses previous work on execution trace collection and compression techniques.

2.1 Static Metrics

Fenton and Neil [FN00] report that during the late 1960's, the *Lines of Code (LOC)* metric has been used to measure both programmer productivity (e.g., LOC per programmer month) and program quality (e.g., defects per LOC). They consider such measurements as being the first active work on static software metrics.

The first attempt to provide a measure of program size which was independent of coding style and programming language is due to Halstead [Hal72, Hal77], who introduced *Software Science*. Halstead's software science is based on the assumption that several properties of software systems can be expressed in terms of a few basic quantities, which are defined in terms of *operators* and *operands*. Halstead based his metrics on four basic quantities: the number of distinct and repeated operators and operands in the source code of a program. Halstead defined the *length* of a program as the total number of occurrences of operators and operands. He also defined the *volume* of a program as the total number of binary digits required to represent the program if all entities are assigned a unique, positive integer value.

By only distinguishing between operators and operands, Halstead aimed to provide a language-independent metric. In practice however, the counting method that is used is not only highly dependent on a specific programming language, but also open to interpretation. Halstead did not judge necessary to provide an unambiguous definition of operators and operations. This lack of a precise definition has been extensively discussed, notably in [Els78], [Sal82] and [Piw82]. Also, empirical studies, such as [HF82], have been performed and denounce the speculative nature of Halstead's metric definitions.

A graph-theoretic approach to measuring software complexity, *cyclomatic complexity*, has been proposed by McCabe [McC76]. McCabe theorized that complexity was only affected by the decision structure of a program. The cyclomatic complexity metric is derived from the notion of cyclomatic number in graph theory, and represents the number of linearly independent paths in the control flow graph (CFG) of a program, or *basic paths*. Equivalently, it represents the number of decisions statements in the CFG plus one. McCabe suggested that any particular module should not have a cyclomatic complexity value that is higher than 10, and provided empirical evidence to support this claim. He also described a testing methodology which equates the cyclomatic complexity of a graph and the minimum number of unit tests that have to be designed. This methodology relies on the assumption that because all paths in the CFG can be expressed in terms of the basic paths, testing all such basic paths is a sufficient testing strategy.

However, McCabe's definition of cyclomatic complexity has a number of associated issues. Myers [Mye77] pointed out that in the presence compound predicates, several distinct CFGs may accurately represent a program's decision structure. In order to address this problem, Myers proposed a variation on the cyclomatic complexity metric which uses an interval of complexity values rather than a single one. The lower bound of the interval corresponds to the original definition of the metric, i.e., the number of decision statements plus one. The upper bound of the interval corresponds to the number of individual boolean predicates plus one.

While McCabe defined his complexity metric in terms of testing difficulty, Harrison and Magel [HM81] argued that a more appropriate complexity metric measures the difficulty related to understanding the source code of a program. They consider McCabe's decision to exclude any information about nesting level from the complexity measure to be a serious limitation of the metric. They proposed a new metric which combines ideas from both cyclomatic complexity and software science. Each node in the CFG is assigned a *raw* complexity value, which consists of Halstead's volume measure for that particular node, and an *adjusted* complexity value. The adjusted complexity value only differs from the raw complexity value in the case of selection (conditional) nodes. For such nodes, the adjusted complexity value is defined as the sum of the adjusted complexities of all of the nodes which are within its range. The concept of range of a selection node is defined in terms of nesting levels inside a function.

Evangelist [Eva84] performed a thorough examination of the theoretical foundations of the cyclomatic complexity metric. He identified several shortcomings of the cyclomatic complexity metric, and concluded that McCabe's metric was poorly developed. For example, he argued that the metric is not a good indicator of testing effort because the total number of circuits in the CFG of an application is bounded by an exponential function of the cylomatic complexity value. Based on the work by Harrison and Magel, Evangelist proposed a new, theoretically founded algorithm for computing a control flow metric. This new algorithm was intended to become the basis of a new complexity metric. However, empirical studies needed to be performed in order to determine how it should be adjusted to correspond to our intuitive notions of complexity. To the best of our knowledge, this empirical validation has yet to be provided.

Nejmeh [Nej88] also pointed out the fact that cyclomatic complexity does not account for nesting. He also considered the fact that all kinds of control flow structures are treated equally a shortcoming of the metric. Because the number of acyclic paths in a program can be an exponential function of the number of basic paths, Nejmeh argues that the cyclomatic complexity metric does not even accurately measure the difficulty of testing such programs. To address these issues, he proposes the *NPATH* metric, which measures the number of acyclic execution paths through a function for the C programming language.

Kearney *et al*. [KST+86] criticized the way in which most complexity metrics have been developed. They argued that because most of the measures are designed without a specific application in mind, their applicability to specific uses is questionable. Kearney *et al*. proposed a set of informally-defined properties that aim at determining how complexity measures can be used and applied.

Weyuker [Wey88] proposed a set of formally-defined properties which compose a framework that can be used to compare software metrics, and determine whether given metrics are suitable for specific purposes. These properties were then used to compare several complexity metrics. However, Fenton [Fen94] pointed out that Weyuker's properties are contradictory (proven by Zuse [Zus92]). He argued that finding a single, general measure of complexity is not feasible, and underlined the need for software metrics work to use concepts from measurement theory.

Chidamber and Kemerer [CK94] proposed a set of six object-oriented software metrics which is still widely used today. Among the most commonly used metrics are *Lack of Cohesion in Methods (LCOM)*, which measures the relative disparate nature of the methods of a class, and *Coupling Between Object Classes (CBO)*, which measures the amount of interaction between classes. Despite all of the criticism surrounding Weyuker's properties, Chidamber and Kemerer decided to use them as a theoretical foundation for their metrics, and provided results from an empirical study to demonstrate their usefulness.

These metrics are however not without criticism. Churcher and Shepperd [CS95] pointed out that a simple count of the number of methods in a class is, without a precise definition, subject to a wide variety of interpretations. Hitz and Montazeri [HM96] criticized the CBO and LCOM metrics from a measurement theory standpoint. They argued that the CBO metric has no acceptable empirical relation system, and that the LCOM metric maps intuitively equivalent cases to different values. In order to address this issue, Hitz and Montazeri defined a graph-theoretic version of the LCOM metric.

Briand *et al*. [BDW98, BDW99] have performed extensive studies of both coupling and cohesion metrics in object-oriented systems, and have found that it was very difficult to compare the work done by any of them. They defined, for each metric, a precise terminology and formalism which can be used to specify it in an unambiguous manner.

2.2 Dynamic Benchmark Analyses

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 [DH99] have presented a detailed study of the allocation behaviour of SPECjvm98 benchmarks. They studied heap size, object lifetimes, and various ways of looking at the heap composition. The work by Shuf *et al.* [SSGS01] 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. Li *et al.* [LJN$^+$00] 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. Eeckhout *et al.* [EGD03] have studied the interaction between executing Java programs and the JVM at the microarchitectural level using hardware performance counters. The results of applying their analysis to several benchmarks, including the SPECjvm98 and Java Grande Forum (JGF) benchmark suites, indicate that differences in program input, as well as different virtual machines, can have a large impact on the observed behaviour of an executing Java program.

Daly *et al.* [DHPW01] performed a platform independent bytecode level analysis of the JGF benchmark suite. 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. Byrne *et al.* [BPW01] used a similar approach to compare the SPECjvm98 and JGF benchmark suites. Power and Waldron [PW02] used the frequency of execution of small methods (i.e., methods which contain less than 10 bytecode instructions) as a measure of the level of object-orientation of an application, and compared both suites. They concluded that the benchmarks from the JGF suite are measurably less object-oriented that those from the SPECjvm98 suite.

2.3 Dynamic Metrics

Kobayashi [Kob97] has defined a set of low-level memory reference metrics based on trace-driven simulation of applications. Such metrics are highly platform-dependent, and include information concerning memory blocks and cache behaviour.

Yacoub *et al*. [YAR99] identified the need for dynamic metrics, and proposed two object-level coupling metrics: *Export Object Coupling (EOC)* and *Import Object Coupling (IOC)*. These metrics aim at measuring the quality of a design early in the development of an application. Because at that stage the application is not available for an execution, the measurements are obtained from a simulation of the behaviour of the application using a executable design model. The authors applied this technique to performing risk-assessment in [YAR00].

Mitchell and Power [Mit02, MP03b, MP03a] have defined dynamic versions of the coupling and cohesion metrics that were originally proposed by Chidamber and Kemerer. Their work focuses on measuring dynamic design metrics for the the Java programming language, and uses dynamic profile data gathered during the execution of the application. They applied the technique to the Java Grande Forum [JGF] and the SPECjvm98 [Spec] benchmark suites. The results show that dynamic coupling metric results exhibit a large variation from their static counterparts. On the other hand, no variation has been observed in the case of the dynamic lack of cohesion metrics, because all modules were found to be maximally cohesive ($\mathrm{LCOM} = 0$) in both the static and dynamic versions of the metrics. Mitchell and Power partly attributed this to a lack of discriminating power of the LCOM metric. This claim is backed up by an empirical study conducted by Basili *et al*. [BBM96].

Gupta and Rao [GR01] have identified some limitations of the static cohesion measurements proposed by Ott and Thuss in [OT89, OT93] which may lead to an overestimate of the real cohesion value. In order to address these limitations, they proposed a cohesion measurement based on def-use pairs in the the dynamic program slices of outputs of an application. They show that their metric is at least as precise as its static counterpart, and provide empirical results to demonstrate the difference in precision between both approaches. A brief discussion of metric-driven code refactoring is also provided.

Aggarwal *et al*. [ASC03] argued that static software metrics are not adequate for purposes of improving the running time of software using code optimizations. To address this limitation of the metrics, they have developed a system for computing dynamic metrics related to finding the most frequently executed modules in C programs. They proposed to use the information that is collected by their tool to concentrate the optimization efforts on the most important modules in an application at runtime.

2.4 Trace Collection

Several methods can be used to collect execution data, such as instrumenting a Java virtual machine (JVM), using a profiling interface such as Sun Microsystem's Java Virtual Machine Profiler Interface [JVMPI], or using a Java bytecode transformation tool such as SOOT [VRGH+00]. For example, Reiss and Renieris [RR00] described a system for gathering Java trace data using the JVMPI. Their system collects runtime events from an executing JVM, and produces one trace per started thread. A separate component in the system is responsible for merging all individual traces into a single, complete execution trace for the entire application.

Because of the large amount of data that is typically stored in execution traces, several trace compression techniques have been developed. Arnold and Ryder [AR01] have designed and implemented a framework for performing low-overhead instrumentation sampling in the Jalapeño JVM [AAB+00]. Although sampling is a lossy trace compression technique, they report a very high accuracy of the data while incurring a very low overhead during profiling (typically around 3%). Reiss and Renieris [RR01] used a two-phase process to filter data from execution traces and encode the result in an appropriate format.

Several lossless trace compression techniques have also been proposed. Burtscher and Jeeradit [BJ03] have used a set of value predictors to reduce the size of an execution trace. STEP [Bro03] is a trace definition compiler system designed to provide profiler developers with a standard method for encoding general program trace data in a flexible and compact format. STEP provides a trace definition language (STEP-DL), which is used to specify both the format of the trace as well as trace-specific compression strategies to be used during the encoding process.

Chapter 3
Dynamic Metrics

This chapter presents a new set of dynamic metrics which are designed to summarize the dynamic behaviour of Java programs. The metrics are not, however, restricted to the Java language; many of them would apply to different, even non object-oriented languages. Section 3.1 outlines some desirable properties that guided the design of the new metrics. Section 3.2 describes the different kinds of dynamic metrics that are present in the set. Section 3.3 discusses how the metrics can be applied to partitions of the sample space, and presents the partitioning schemes that have been used. Section 3.4 gives a detailed description all of the dynamic metrics, along with relevant empirical data.

3.1 Properties

When designing new dynamic metrics, one must ensure that they adequately capture the aspect of software behaviour that they are intended to measure. New metrics must also render clear and comparable numbers for any kind of program. Therefore, we outline some general requirements for dynamic metrics which address some of the most important factors which may impact their usefulness. These properties not only helpful in designing the metrics, but can also be used in the evaluation of the applicability of a particular metric to specific purposes. These desirable properties are only presented informally; it may not be possible to realistically achieve all of them for every metric.

Unambiguous

One lesson learned from static metric literature is that ambiguous definitions lead to unusable metrics. It is therefore crucial to provide a clear, precise and unambiguous definition of all dynamic metrics. For instance, the most widely used static metric for program size is "lines of code" (LOC). LOC is sufficient to give an approximate measure of program size or to evaluate an amount of programming work. However, without further specification, it is virtually useless to compare two programs. Blank lines, comments and coding style obviously have potentially large impacts on the metric. A precise definition of the metric has to be provided in order to make meaningful comparisons of different programs using the LOC metric. For example, 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.

Object-oriented languages often add another source of ambiguity because of inheritance. For example, referring to the methods of a given class is ambiguous, because it is not specified whether the methods which are inherited but not overridden by the class should be included or not.

Dynamic

In order to be considered dynamic, a metric should measure an aspect of a program that can only be obtained by actually executing it. The dynamic nature of a metric makes it unaffected by the addition of *dead code* (unexecuted code) to the program, because code that is never executed will obviously never contribute to the measured value.

In compiler optimization papers, static metrics are often reported instead of their dynamic counterparts, partly because they are easier to obtain. However, they tend to relate to the *cost* of a particular optimization technique, whereas dynamic metrics relate to the *relevance* of such a technique. For example, counting the number of virtual call sites in a program gives an upper-bound for the number of inlining opportunities. However, because some virtual call sites may actually be polymorphic, the number of call sites which can be dynamically de-virtualized may be less than the statically-computed number of opportunities. A dynamic metric would, in this case, be more relevant because it could measure the number of *actually monomorphic* call sites in the program. Moreover, dynamic metrics can better assess the overall impact of the optimization on performance, because they can be used to determine how the monomorphic call sites are actually used. Optimizing a call site which is located within a frequently executed loop intuitively has a greater impact on performance than optimizing a call site which is executed only once during the startup phase of the application.

Robust

The other side of the coin of using dynamic measurements is the possibility that the measures 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*, *relevant* change in behaviour should cause a correspondingly *small* change in the resulting metric. Conversely, *irrelevant* changes in behaviour should not affect any unrelated metrics. Definitions of *small* and *relevant* of course will depend on the situation under analysis.

In particular, a robust metric should not be overly sensitive to the size of a program's input. A bubble sort, for example, will execute four times as many bytecodes if the input size is increased by a factor of two. Thus, the total number of bytecode instructions executed is not a robust metric of program size. The 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 often misleading and non-robust. For example, the total amount of memory allocated by an executing program, a metric often reported in the literature, says little about the "memory-hungriness" of the program. A relative metric, such as bytes allocated per executed bytecode, is more robust. Merely running a program twice as long will have less effect on a relative metric. Robustness is in practice not always easy to achieve, and mostly depends on what effects are considered “relevant” and “small”.

Discriminating

Robustness is however not by itself a sufficient property. For example, a metric which returns the same value for all programs executions is in some sense maximally robust. No irrelevant—or relevant—changes affect it; it is not *discriminating*. This results in a property that is dual to robustness within a given context. A metric is discriminating if a *large* change in behaviour causes a correspondingly *large* change in the resulting metric.

Platform-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). While it may seem like platform-independence is easily achieved in languages such as Java, which are designed to be multi-platform, reality is otherwise. The Java language is designed to have “as few implementation dependencies as possible” [GJSB00]. However, while typically Java applications may be executed on different platforms without requiring any modifications, the Java Virtual Machines (JVMs) themselves exhibit some differences. For instance, the JVM specification [LY99] does not mandate a particular object header size. As a result, when considering memory metrics one needs to be aware that the size of allocated objects in different JVM implementations may vary. Internal differences in the standard libraries may also be observed. Different JVM vendors may modify the libraries as long as the API is not affected, and are likely to do so for various reasons, including efficiency. Of course, differences in the class hierarchies also exist simply to support platform-independence. The `java.lang.UNIXProcess` class, for example, is not found on non-UNIX platforms.

Differences in implementations are however not the only possible source of platform-dependence in the metrics. For example, the 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 reproduce the exact conditions under which the experiment was executed. On the other hand, number of objects allocated per 1000 executed bytecode instructions (*kilobytecode*, or kbc), is a much less platform-dependent metric. In general, metrics defined around the bytecode as a unit of measurement can be considered machine-independent for Java programs.

3.2 Kinds of Dynamic Metrics

While there are many possible metrics one could gather, a survey of the literature in the field of compilers and runtime systems reveals that the most commonly described metrics, and the ones which seem most useful, tend to be belong to a few basic categories. A new classification of the metrics into three basic categories is presented. These categories correspond to the ubiquitous value metrics such as average, hot spot detection metrics and metrics based on discrete categorization. A more detailed continuous “expansion” of each category is also mentioned.

It is of course possible to design and use a dynamic metric that does not fit into these categories; these initial metric kinds, however, are useful to at least begin to explore the various potential metrics by considering whether an appropriate metric exists in each of our categories.

3.2.1 Value Metrics

The value metric is by far the most commonly used kind of dynamic metric, and corresponds to typical, one value answers. 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 easily compare results for different benchmarks, since the values form an intuitive totally ordered set. It may also be used to allow one to observe differences in behaviour before and after some transformation. For example, the total number of bytes allocated and the number of method invocations are two frequently reported dynamic value metrics. Value metrics appear in almost every research article that presents dynamic measurements.

3.2.2 Percentile Metrics

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.

Percentile metrics are similar to value metrics, but additionally have an associated threshold value which indicates the proportion of the program entities which are to be considered in the computation of the metric, i.e., the “hotness level” that is measured by the metric. A higher threshold is used when looking for more pronounced hotspots, and is thus associated with a higher hotness level.

Similar metrics can be found in compiler optimization literature; e.g., the top x% of most frequently-executed methods [KC01].

3.2.3 Bin Metrics

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 or targets per call site may not give a good impression of the optimization opportunities for de-virtualization. An appropriate metric for this example would be to give a relative or absolute value for each of the categories of interest, namely 1, 2, or $\geq$3 target types.

These kinds of metrics are referred to as bin metrics, 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) [GRS00], the percentage of loops requiring less than x registers [ZLAV00].

3.2.4 Continuous Metrics

All three kinds of dynamic metrics have continuous analogues, where the calculations are performed at various (or all) partial stages of execution (rather than once at the end of the 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. The focus of this work is on establishing general characterizations of benchmarks that one could use as a basis or justification for further investigation; actual continuous metrics are thus left as future work (see Section 6.2.2).

3.3 Sample Space Partitioning

While it is possible to compute dynamic metrics for the entire set of events collected during the execution of a Java program, it is often desirable to compute metric values based on a subset of the dynamic data. For example, differentiating between application and library code is often beneficial, especially for small programs. The virtual machine startup phase is a large program in itself, and thus the inclusion of the contributions from non-application code can significantly distort the metric values. It is therefore useful to think of metric definitions as general "recipes" that can be applied to arbitrary partitions of the sample space rather than representing specific concepts. Two partitioning schemes are introduced and used used as part of this work:

- **Whole program**: this trivial partitioning scheme defines a single partition which includes contributions from the entire sample space.

- **Static application/library**: this partitioning scheme defines two distinct partitions of the sample space, `application` and `library`, based on a package name filtering strategy. One partition corresponds to the Java standard library, the other one to the application itself. Methods, bytecodes, and other class file constructs contribute to the partition which corresponds to the class in which they are found. Object instances contribute to the partition which corresponds to their runtime type; all arrays are considered to be part of the library partition. This partitioning scheme is very relevant in the context of code optimizations, since only the application code can usually be optimized by a compiler or by manually changing the program. Conversely, the library-only version of a metric can provide meaningful information relative to library performance, and could potentially be useful for comparing the libraries from different JVM implementations.

A number of different and more elaborate partitioning schemes can obviously be designed. Examples include a more dynamic subdivision of the sample space based on differentiation between startup/runtime support and the main computation (application + called library code), or a per-thread partitioning strategy.

3.4 Metric Definitions

This section presents a set of new dynamic metrics which aim to summarize the behaviour of Java programs. The metrics naturally fit into five groups: program size and structure, measurements of data structures, polymorphism, dynamic memory use and concurrency. For each group, individual metrics are defined, and appropriate benchmark data is presented in order to validate the metric and discuss its properties. All experimental data that was reported as part of this thesis has been collected using Sun's Hotspot$^{\mathrm{TM}}$ Client VM (build 1.4.0-b92[1], mixed mode), on a Pentium 4 1.8 GHz machine runing Debian Linux. The five metric groups are discussed in the next subsections.

3.4.1 Program Size and Structure

Traditionally, the notions of program size and structure have been studied as measures of a program's complexity. Such attributes have been extensively studied in a static context; the resulting metrics have a large number of potential users. Dynamic metrics which capture the notions of dynamic size and control structure complexity are presented next.

Program Size

Before dynamic loading became commonplace, an approximation of a program's size could be obtained by measuring the size of the executable file. However, in languages such as Java that allow dynamic loading it is necessary to execute the program in order to accurately measure its size.

Table 3.1 shows program size metric values for a number of selected benchmarks (a detailed description of the benchmarks is given in Section 5.1). The metric values are provided for both the entire code (labelled "All") as well as the application (non-library) code only (labelled "Application", or "App."). The description of the metrics are given below. A complete set of values for all benchmarks is also given in Appendix A.

[1] Flaws in the implementation of the JVMPI interface in more recent versions of Sun's JRE caused the collected data to be incomplete, and precluded their use in this study.

	Metric	EMPTY	MST	LINPACK	COEFFICIENTS	COMPRESS	SOOT	JAVAC
All	size.loadedClasses.value	277	281	280	286	310	819	471
	size.load.value	72059	76295	78173	80292	90762	126901	133172
	size.run.value	7354	10703	10708	12887	14514	38404	37831
	size.hot.value	985	186	119	116	396	3323	2261
	size.hot.percentile	13.4%	1.7%	1.1%	0.9%	2.7%	8.7%	6.0%
	size.codeCoverage.value	10.2%	14.0%	13.7%	16.1%	16.0%	30.3%	28.4%
Application	size.loadedClasses.value	1	6	1	6	22	532	175
	size.load.value	4	720	1056	2374	6555	45446	44664
	size.run.value	0	593	749	975	5084	26239	26267
	size.hot.value	0	175	59	57	396	2759	2759
	size.hot.percentile	N/A	29.5%	7.9%	5.8%	7.8%	10.5%	10.5%
	size.codeCoverage.value	0.0%	82.4%	70.9%	41.1%	77.6%	57.7%	58.8%

Table 3.1: Size metrics

size.loadedClasses.value This metric measures the total number distinct classes that are loaded. It gives a rough idea of program size.

From Table 3.1, size.loadedClasses.value starts at 277, rises to 286 for COEFFICIENTS and to 310 for COMPRESS. Only JAVAC and SOOT really stand out with 471 and 819 total loaded classes. The large number of classes that are required to execute the EMPTY benchmark clearly indicates that the startup phase of the JVM is itself a large program; including it in the metrics can therefore significantly skew the values for relatively small applications. The contribution of startup to the metric also tends to make the measured values very similar; about half of the benchmarks load between 277 and 292 classes. As a result, the size.loadedClasses.value values that include the standard libraries are not discriminating enough to be used as a measure of program size. This metric is also not platform-independent, because the number of classes that are loaded during startup can significantly differ depending on the JVM that is used to execute the program. For example, IBM's Classic VM (build 1.4.1, J2RE 1.4.1 IBM build cxia32141-20030522 (JIT enabled: jitc)) loads 293 classes for the EMPTY benchmark. Therefore, the application-only version of the metric is preferred over its whole-program counterpart as a measure of program size.

However, because the size of each class may exhibit a large variation, this metric may not correspond to the intuitive notion of program size. For example, the EMPTY and LIN-

PACK benchmarks both load only one application class, but are clearly not the same size. Also, LINPACK's single class contains more bytecode instructions than MST's six application classes combined. Based on this empirical evidence, the application-only version of the size.loadedClasses.value is more robust than discriminating.

size.load.value This metric measures the total number of bytecodes that are loaded. It is computed by adding the number of bytecodes in all methods of a class to a running total each time a new class is loaded. This metric is the closest dynamic equivalent to the static size of a program, and corresponds more closely to the intuitive notion of program size than size.loadedClasses.value.

However, it suffers from the same problems as the size.loadedClasses.value metric when it includes the standard libraries, and becomes platform-dependent and insufficiently discriminating as a measure of program size. On the other hand, the size.load.value metric is less sensitive to the particular programming style and class decomposition than size.-loadedClasses.value. It is therefore more robust in this respect.

From Table 3.1, the the fact that the EMPTY benchmark loads 4 bytecodes requires further explanations. An empty method requires only a single `return` bytecode. However, the `javac` compiler adds a default constructor to the compiled class file, even though it is not used in this case. This constructor is comprised of 3 bytecode instructions, for a total of 4 loaded bytecodes.

size.run.value This metric measures the number of bytecode instructions that were executed at least once, or *touched*. It is computed by keeping an `executed` bit for each bytecode instruction that is part of a loaded class. Initially, all executed bits are unset. Whenever a bytecode instruction with an unset `executed` bit is executed, a running counter is incremented, and the bit is set. The value of this metric is bounded by the value of the size.load.value metric. In practice, however, size.run.value is usually much smaller because applications tend to contain a significant amount of (dynamically) dead code.

Empirical results show that size.run.value is a discriminating measure of program size. Table 3.1 is sorted left-to-right in increasing order with respect to this metric. It can be observed that there is a clear progression of sizes which closely matches the intuitive notion of program size. This metric can be used to classify benchmark programs in five distinct size categories which are listed in Table 3.2.

Category	size.run.value	Examples
XS	< 100	EMPTY
S	100–2000	MST, LINPACK, COEFFICIENTS
M	2000–10K	COMPRESS
L	10K–50K	SOOT, JAVAC
XL	> 50K	FORTE

Table 3.2: Program size categories.

In order to study the robustness of metrics, a number of benchmarks have been analyzed using different inputs. Such benchmarks include AUTOMATA, COEFFICIENTS, SABLECC, SOOT, as well as all benchmarks from the SPECjvm98 suite (see Section 5.1 for a detailed description of all input sizes). Results show that the size.run.value metric is robust with respect to program input: different executions of a number of benchmarks did not result in substantial differences, and did not upset the relative ordering of benchmarks. It is also robust with respect to various compiler optimizations (see Section 5.2.3). Therefore, the size.run.value metric is the preferred dynamic measure of program size, as it seems to achieve the right balance between robustness and discriminating power.

The results of the EMPTY benchmark again merit an explanation. It was previously argued that an empty method requires a single bytecode; however, EMPTY obtains a size.-run.value value of 0. This is due to an optimization performed by Sun's Hotspot$^{\mathrm{TM}}$ Client, which avoids calling empty methods. Thus, the `main` method of the EMPTY benchmark is in practice never entered.

size.hot.value This metric measures the number of distinct bytecode instructions that are responsible for 90% of the total bytecode executions. It is computed by counting the number of times each bytecode instruction is executed, sorting the bytecodes by frequency, and reporting the number of bytecodes which account for 90% of the total bytecode executions. It represents the size of a program's hotspot.

From Table 3.1, this metric is very discriminating; hotspot sizes range from 116 for LINPACK and COEFFICIENTS to 3323 for SOOT. However, this metric lacks robustness. For example, using a different input in the SOOT benchmark resulted in the number of hot bytecodes to drop from 2759 to 1191.

size.hot.percentile This metric is derived from the size.hot.value metric, and measures the *proportion* of the touched code which is responsible for 90% of the total bytecode executions. It is computed as size.hot.value / size.run.value.

As it can be observed from Table 3.1, size.hot.percentile is much less discriminating than size.hot.value; all benchmarks except EMPTY have a hotspot proportion which is below 10%. This results supports the well-known 90-10 rule, which claims that in general, only 10% of an application is responsible for 90% of the total execution. The size.hot.-percentile metric is also more robust than size.hot.value. The same variation of inputs for the SOOT benchmark resulted in a 5.5% difference only; a similar variation on the SABLECC benchmark left the metric value almost unaffected.

The application-only version of the metric is a little more discriminating than its whole-program counterpart; the MST benchmark reaches a value of 29.5%, while the JAVAC benchmark barely crosses the 10% boundary.

size.codeCoverage.value This metric measures the *proportion* of the total code that is touched. It is computed as size.run.value / size.load.value.

Table 3.1 shows that most of the library code that is loaded is never executed; code coverage spans a range of 10.2% for EMPTY to 30.3% for SOOT. This is intuitively explained by the fact that the Java standard libraries have been designed with a broad range of applications in mind, and thus any single benchmark application is not likely to cover them all. Conversely, most benchmarks tend to touch most of their own loaded code. COEFFICIENTS stands out because it does not even execute half of its own code; this is due to the fact that the benchmark's `Matrix` class supports many more operations than are required by the simple pseudo-inverse algorithm that it implements. This is analogous to the use of the standard libraries. SOOT and JAVAC also have low size.codeCoverage.value values which can explained in similar ways.

It is clear from the obtained values that both the whole-program and application-only versions of the size.codeCoverage.value metrics lack discriminating power; none of the benchmark's whole code coverage value exceeded 40%, and none of them touched less than 40% of its own code (except for the EMPTY program). In fact, the vast majority of the benchmarks touched over 70% of their application code. This metric is also very robust in terms of both program input and compiler optimizations. With the exception of JAVAC, variations in input for all other benchmarks resulted very small differences in the metric values.

3.4.2 Data Structures

The data structures and types used in a program are of frequent interest. For example, optimization techniques change significantly for programs that rely heavily on particular classes of data structures. Techniques which are useful for array-based programs, for instance are different from those that may be applied to programs building dynamic data structures. The study of data structure manipulations will be broken down into three key aspects which relate to the intensity of use of arrays, floating point operations and pointers.

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 accesses in Java have other opportunities for optimization, e.g., array bounds check removal [QHV02].

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

	Metric	COEFFICIENTS	LINPACK	AUTOMATA	BARNES-HUT	POWER	SABLECC	EMPTY
All	data.arrayDensity.value	150.880	151.953	31.059	105.891	93.417	43.635	73.311
	data.charArrayDensity.value	1.562	2.197	10.430	0.013	0.017	6.638	33.186
	data.numArrayDensity.value	74.986	140.602	14.581	97.512	92.490	9.450	34.213
	data.refArrayDensity.value	73.708	8.867	0.396	4.380	0.156	15.613	1.795
Application	data.arrayDensity.value	160.404	157.775	133.667	105.947	97.433	38.868	N/A
	data.charArrayDensity.value	0.000	0.000	14.805	0.000	0.000	0.000	N/A
	data.numArrayDensity.value	79.486	148.385	118.803	97.577	96.487	11.209	N/A
	data.refArrayDensity.value	80.713	9.389	0.015	4.383	0.162	13.274	N/A

Table 3.3: Array metrics

required for each multi-dimensional array access is magnified. This skewing factor could be eliminated by ignoring array accesses where the array element is an array itself; this is planned as future work. Evidence of such skewing is given later, in Section 5.2.1.

data.arrayDensity.value This metric measures the number of array access bytecodes executed, on average, per kbc. It describes the relative importance of array access operations. Further refinement of the metric can be done according to the type of the array being accessed: data.charArrayDensity.value for character arrays, data.numArrayDensity.-value for arrays of primitive numerical types, and data.refArrayDensity.value for arrays of non-primitive (reference) types.

Example metric calculations for each of the proposed array density metrics are given in Table 3.3. The impact of startup and library code is again very apparent: for example, the AUTOMATA benchmark has a very high array density for its application code only, but ranks below the EMPTY program when the whole program is being considered. This indicates that while the AUTOMATA benchmark itself makes intensive use of arrays (and even startup has a significant use of arrays), the library methods that the benchmark calls have a limited use of arrays. In fact, while the AUTOMATA 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 and 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.carArrayDensity.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[2] in the library code. Interestingly, the startup code is by far the most intense user of char arrays; only one of the benchmarks of significant duration/size actually has a character array density that is slightly larger than the that of the EMPTY program (even the benchmarks that do parsing, such as JAVAC and SABLECC, ranked lower).

The threshold for considering a program array intensive is not as clear as with some other metrics; the benchmarks, application or whole, tend to be fairly evenly distributed over the range of reasonable density values. This however shows the discriminating nature of the array intensity metrics. A value of the application version of the data.arrayDensity.-value metric in the high 90's identifies the majority of what would intuitively be considered array intensive programs.

The array density metrics generally achieve a good balance between discriminating power and robustness. Empirical results show that varying the input of the program has little effect on the metrics, and generally keeps the relative ordering of the benchmarks.

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.

[2]Copies of the char array can of course be created and used outside the library, but in the benchmarks that

Benchmark	data.floatDensity.value	
	All	Application
POWER	474.918	461.224
TSP	471.478	499.775
LINPACK	285.427	306.142
BARNES-HUT	245.669	245.580
VORONOI	226.363	226.489
COEFFICIENTS	202.808	217.956
EM3D	13.512	11.913
EMPTY	1.987	N/A
SOOT	0.717	0.269
JAVAC	0.072	0.000

Table 3.4: Floating-Point density metric

data.floatDensity.value This metric measures the average number of floating point operations per kbc. It is intended to capture the relative importance of floating-point operations in an application. It is computed as the number of executed bytecode instructions that operate on either float or double type divided by the total number of executed bytecodes times 1000.

As can be seen from Table 3.4, high float density values correlate well with benchmarks or algorithms that have been traditionally considered to rely on numeric, floating-point data (POWER, TSP, COEFFICIENTS, LINPACK, etc.), and low values generally correspond to non-numeric benchmarks (JAVAC, SOOT, etc.). Some apparently numeric benchmarks are pruned out by this metric; the EM3D benchmark, for example. While this program does use floating-point data in an iterative calculation, by default it only computes a single 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.

Because the EMPTY program has a very low float density value of 1.987, the relative float density of very small floating-point benchmarks is necessarily diluted when considered in their startup and library context. Even in the whole-program metric, though, the division between floating-point intensive and not is quite sharp: the 226.363 value for VORONOI drops to 13.512 for EM3D, and then rapidly reaches small single digits for less

have been analyzed this does not occur often.

float-intensive programs. This indicates that the metric possesses a good discriminating power. Based on this empirical data, a data.floatDensity.value value of at least 100 appears to be a good indicator of a floating-point intensive program.

Empirical evidence also shows that, unlike most densities, data.floatDensity.value achieves a good amount of robustness. For example, a different input for the COEFFICIENTS benchmark resulted in the number of executed bytecodes to double, but left the float density almost unchanged.

With respect to identifying "scientific" benchmarks, it is useful to know which benchmarks combine floating-point intensity with array usage. In the benchmark list this includes COEFFICIENTS, BARNES-HUT, POWER, and LINPACK (see tables 3.3 and 3.4). Note that while these combine intensive floating-point usage with intensive array usage, they do not necessarily contain perfect loops over arrays. BARNES-HUT, for instance, uses numerous arrays and vectors, but computationally is largely based on traversing and modifying a tree structure.

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 than 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.

pointer.refFieldAccessDensity.value This metric measures the number of reference field access bytecodes executed, on average, per kbc. It coarsely describes the importance of pointer references in a program. In a pointer-intensive program, one expects that

	Metric	AUTOMATA	COMPRESS	JAVAC	SABLECC	LINPACK	EMPTY
All	pointer.nonrefFieldAccessDensity.value	92.139	95.144	93.849	57.370	2.120	30.054
	pointer.refFieldAccessDensity.value	55.905	94.413	70.017	99.673	1.679	23.790
App	pointer.nonrefFieldAccessDensity.value	75.393	95.152	120.080	40.635	0.001	N/A
	pointer.refFieldAccessDensity.value	133.837	94.422	88.269	124.260	0.000	N/A

Table 3.5: Pointer metrics

the effect of following pointer references to object fields will result in a high value for this metric.

pointer.nonrefFieldAccessDensity.value This metric measures the number of primitive field access bytecodes executed, on average, per kbc.

Examples of both metrics are shown in Table 3.5. From this, the AUTOMATA and SABLECC benchmarks are both very pointer-intensive. While it is no surprise in the case of SABLECC, the case of AUTOMATA merits an explanation: AUTOMATA is a simple one-dimensional cellular automaton simulation program, and uses arrays of integers to store the state of the automaton. It is thus expected to spend most of its time executing primitive array operations. In Section 3.4.2, it has already been demonstrated that AUTOMATA is indeed very array-intensive. The pointer intensity of the AUTOMATA benchmark is therefore surprisingly high, and can be attributed its naive implementation. The benchmark uses two arrays to keep track of the state of the cellular automaton; one array is used to store the state of the automaton at each iteration, the other is used as a temporary buffer. After each iteration, contents of the temporary array are copied back to the "real" array. Both arrays are allocated once, and stored in fields. All accesses to the arrays, however, are made directly from the field reference, resulting in an artificially high pointer.refFieldAccessDensity.-value. A similar situation can be observed for the array size, which is also stored in a field. Modifying the source code to cache field values in local variables has the expected effect on the metric values, as shown in Table 3.6.

LINPACK 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). In the case of

	Metric	Original	With caching
App	pointer.nonrefFieldAccessDensity.value	75.393	54.275
	pointer.refFieldAccessDensity.value	133.837	18.750

Table 3.6: Impact of caching field values on pointer metrics in AUTOMATA benchmark

COMPRESS, which one would expect not to be pointer intensive, the benchmark has a high reference and primitive field accesses. This is largely due to accessing buffers and arrays of constants; in particular, COMPRESS makes an intensive use of a byte buffer, which requires multiple primitive field operations for manipulating the current index, and one reference field access to obtain a reference to the byte array itself. The nature of JAVAC is less clear; it has a very high density of primitive field accesses, and a moderately high density of reference field accesses. Conversely, SABLECC has a very high density of reference field accesses, and a moderate primitive field acess density; this corresponds to the intuition for this benchmark. The coarseness of this metric thus seems adequate to identify applications that are unequivocally 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.

3.4.3 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` bytecode. The target method of a polymorphic call depends on the runtime type of the receiver object. In programs that do not employ polymorphism, such as programs that have a limited use of 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-orientation.

Table 3.7 presents polymorphism metrics for seven distinctive benchmarks.

polymorphism.callSites.value This metric measures the number of potentially polymorphic bytecode instructions (i.e., `invokevirtual` or `invokeinterface`) that have been touched. This measurement excludes all `invokestatic` and `invokespecial` bytecodes, but includes all `invokevirtual` and `invokeinterface` call sites, even if they have only one associated receiver at runtime. Therefore, this metric does not re-

	Metric	Compress	Jess	Barnes-Hut	Soot	Javac	Perimeter
All	polymorphism.callSites.value	606	1309	667	3802	3233	559
	polymorphism.invokeDensity.value	16.532	55.283	18.664	51.691	39.158	45.766
	polymorphism.receiverArity.bin(1)	97.5%	97.7%	97.6%	93.6%	81.8%	95.2%
	polymorphism.receiverArity.bin(2)	2.3%	1.6%	2.2%	3.0%	8.3%	2.0%
	polymorphism.receiverArity.bin(3+)	0.2%	0.7%	0.1%	3.5%	9.9%	2.9%
	polymorphism.receiverArityCalls.bin(1)	100.0%	93.3%	86.7%	68.3%	70.7%	37.0%
	polymorphism.receiverArityCalls.bin(2)	0.0%	2.4%	13.3%	13.2%	12.6%	0.0%
	polymorphism.receiverArityCalls.bin(3+)	0.0%	4.3%	0.0%	18.5%	16.6%	63.0%
	polymorphism.receiverCacheMissRate.value	0.0%	3.2%	3.0%	8.9%	8.8%	41.0%
	polymorphism.targetArity.bin(1)	98.4%	98.5%	98.4%	95.2%	91.2%	96.8%
	polymorphism.targetArity.bin(2)	1.5%	0.9%	1.5%	2.1%	3.4%	1.1%
	polymorphism.targetArity.bin(3+)	0.2%	0.6%	0.1%	2.6%	5.4%	2.1%
	polymorphism.targetArityCalls.bin(1)	100.0%	93.4%	86.7%	72.8%	86.8%	42.9%
	polymorphism.targetArityCalls.bin(2)	0.0%	2.4%	13.3%	11.9%	1.7%	0.0%
	polymorphism.targetArityCalls.bin(3+)	0.0%	4.3%	0.0%	15.3%	11.5%	57.1%
	polymorphism.targetCacheMissRate.value	0.0%	3.2%	3.0%	4.3%	4.8%	37.4%
Application	polymorphism.callSites.value	54	737	129	3186	2617	49
	polymorphism.invokeDensity.value	16.532	59.194	18.672	69.927	71.771	45.891
	polymorphism.receiverArity.bin(1)	98.1%	98.4%	96.9%	93.3%	78.4%	69.4%
	polymorphism.receiverArity.bin(2)	1.9%	0.8%	3.1%	3.0%	9.7%	0.0%
	polymorphism.receiverArity.bin(3+)	0.0%	0.8%	0.0%	3.7%	12.0%	30.6%
	polymorphism.receiverArityCalls.bin(1)	100.0%	99.2%	86.7%	78.4%	72.6%	36.9%
	polymorphism.receiverArityCalls.bin(2)	0.0%	0.1%	13.3%	15.4%	15.1%	0.0%
	polymorphism.receiverArityCalls.bin(3+)	0.0%	0.8%	0.0%	6.2%	12.3%	63.1%
	polymorphism.receiverCacheMissRate.value	0.0%	0.4%	3.0%	4.4%	7.2%	41.1%
	polymorphism.targetArity.bin(1)	98.1%	98.9%	96.9%	95.1%	89.7%	77.6%
	polymorphism.targetArity.bin(2)	1.9%	0.4%	3.1%	2.1%	3.8%	0.0%
	polymorphism.targetArity.bin(3+)	0.0%	0.7%	0.0%	2.8%	6.5%	22.4%
	polymorphism.targetArityCalls.bin(1)	100.0%	99.2%	86.7%	83.5%	92.0%	42.8%
	polymorphism.targetArityCalls.bin(2)	0.0%	0.0%	13.3%	13.9%	1.9%	0.0%
	polymorphism.targetArityCalls.bin(3+)	0.0%	0.8%	0.0%	2.6%	6.1%	57.2%
	polymorphism.targetCacheMissRate.value	0.0%	0.4%	3.0%	2.7%	3.1%	37.5%

Table 3.7: Polymorphism Metrics

flect the use of polymorphism, but rather a measure of program size. It is intended to be a dynamic version of the frequently reported static measure which involves counting all call sites in the code.

From a compiler optimization point of view, the polymorphism.callSites.value metric gives an indication of the amount of effort required to optimize polymorphic calls, but not of the relevance of such optimizations to performance.

As it was previously observed with the size.run.value metric, this metric is robust with respect to both program input and various compiler optimizations. The impact of changing the input of all benchmarks except JAVAC had a very small impact on the metric. In the case of JAVAC, going size 1 to size 10 doubled both the whole program and application-only variations of the polymorphism.callSites.value metric value. This result tends to indicate that the changes between the different input sizes for JAVAC result in a rather pronounced change in the overall behaviour of the benchmark.

Changes in the input of all benchmarks had only a small impact on the metric, and did not change the relative ordering of the benchmarks.

The application version of the metric is more discriminating than the whole program version. For instance, about half of the benchmarks touch less than 100 call sites in their application code. The whole program version of the polymorphism.callSites.value metric therefore reaches a value of around 550 for many of the benchmarks. Also, the EMPTY program touches 437 call sites before terminating, and thus startup can significantly distort the results even for benchmarks which themselves touch a large number of call sites. For example, the SOOT benchmark touches 3845 call sites during its entire execution, but only 3229 of them are found in application code, which represents a difference of around 16%.

polymorphism.invokeDensity.value This metric measures the average number of `invokevirtual` and `invokeinterface` bytecodes executed, on average, per kbc. It indicates the relative importance of virtual method invocations, and thus can be used to assess the relevance of optimizing invokes. For example, both COMPRESS and PERIMETER have similar number of touched call sites in their application code (54 and 49, respectively), but COMPRESS has an application polymorphism.invokeDensity.value value of 16.532, while PERIMETER has an invoke density of 45.891. Optimizing method invocations can be expected to be more beneficial when applied to PERIMETER than to COMPRESS.

Both the whole program and application versions of the polymorphism.invokeDensity.-value metric are easily perturbed by variations in the program's input. For example, SOOT's invoke density dropped from 69.927 down to a mere 37.510 by only changing the input. A similar, although less pronounced, variation can be observed by changing SABLECC's input. The metrics also shows a clear progression of values, starting around 0.5 for POWER and reaching values in the 120-130 range for a number of benchmarks, indicating that it is very discriminating.

The previous metrics did not provide information relative to the more realistic amount of polymorphism. The following metrics capture a more accurate notion of polymorphism. There are two variants. In the first variant, polymorphism is expressed in terms of the number of receiver types associated with a call site, and in the second variant, polymorphism is defined in terms of the number of distinct target methods that are invoked by a call site. The number of receiver types at any given call site is an upper bound on the number of target methods that it may invoke. Two different receiver types can, however, result in the same target method being invoked in cases where a method of a common super class has been inherited but not overridden by several distinct receiver classes.

Optimization techniques that are related to virtual method invocations are usually designed to take advantage of a low level of polymorphism in one of the variants. For example, devirtualization techniques based on Class Hierarchy Analysis (CHA) [DGC95] optimize call sites that have a limited number of targets. Other techniques, such as inline caching [DS84], optimize call sites with a restricted number of receiver types. Specific metrics for each variant are described next.

Receiver Polymorphism

polymorphism.receiverArity.bin This metric shows the percentage of all touched call sites that have one, two and more than two distinct receiver types at runtime. The nature of this metric makes it ideal for comparison with type inference techniques which conservatively estimate the number of receiver types without running the program. However, by only counting call sites, the metric does not reflect the importance of those sites.

polymorphism.receiverArityCalls.bin This metric shows the percentage of all calls originating from call sites that have one, two and more than two distinct receiver types at runtime. It is "more dynamic" than the previous polymorphism.receiverArity.value metric because it measures the *importance* of polymorphic calls. For example, the COMPRESS benchmark has 1.9% of polymorphic call sites in its application code, but the zero value of the polymorphism.receiverArityCalls.bin(1) in Table 3.7 indicates that those call sites are almost never executed.

From Table 3.7, all benchmarks except JAVAC appear to mostly have monophorphic call sites. However, looking at the polymorphism.receiverArityCalls.bin values clearly shows that many of the benchmarks have polymorphic call sites that are executed frequently; polymorphism.receiverArity.bin appears to lack discriminating power and to be a rather poor indicator of the use of polymorphism. The application-only counterparts of these metrics are clearly more discriminating. For instance, the application-only version of the polymorphism.receiverArity.bin metric correctly identifies both JAVAC and PERIMETER as being polymorphic benchmarks, but SOOT is still pruned out. The application-only polymorphism.receiverArityCalls.bin metric from Table 3.7, however, reveals the actual use of polymorphism for each benchmark. COMPRESS and JESS are not polymorphic, BARNES-HUT and VOLANO-SERVER are moderately polymorphic, and SOOT, JAVAC and PERIMETER make a significant use of polymorphism.

In the application-only part of Table 3.7, the percentage of monomorphic calls is lower than the percentage of monomorphic call sites for BARNES-HUT, VOLANO-SERVER, SOOT, JAVAC and PERIMETER. PERIMETER is the most extreme case, with 30.6% heavily polymorphic call sites, which are executed 63.1% of the time. This metric is preferred as an appraisal of polymorphism, since it highlights polymorphism that actually occurs, weighted by the frequency of its occurrence. Table 3.7 is therefore sorted in descending order from left to right, using polymorphism.receiverArityCalls.bin(1).

polymorphism.receiverCacheMissRate.bin This metric shows as a percentage how often a call site switches between receiver types. It represents the miss rate of a true inline cache. Inline caching works by keeping a cached copy the result of the previous method lookup at the call site (*inline*). Subsequent executions of the call site will require no lookup as long as the receiver type does not change. Apart from the initial miss, an inline cache which never misses indicates a dynamically monomorphic call site.

This metric 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 21.7% non-monomorphic calls (1 - polymorphism.receiverArityCalls.bin(1)) in its application code, but has a receiver cache miss rate of only 4.3%. PERIMETER, on the other hand, has 63% non-polymorphic calls, and a receiver cache miss rate of 37.4%, indicating that the polymorphic call sites actually switch often between receiver types.

Target Polymorphism

Target polymorphism can be measured in a similar manner as receiver polymorphism, however the metrics focus on the actual target method invoked at a call site rather than the receiver type for the call.

polymorphism.targetArity.bin This metric shows the percentage of all touched call sites that have one, two and more than two distinct target methods at runtime. This metric is very similar to polymorphism.receiverArity.bin, but has a greater relevance in the context of compiler optimizations; 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 [SHR+00].

Like polymorphism.receiverArity.bin, this metric is dynamic, but does not reflect the importance of the calls at runtime.

polymorphism.targetArityCalls.bin This metric shows the percentage of all calls originating from call sites that have one, two and more than two distinct target methods at runtime. The same observations hold as for receiver polymorphism, but the number of monomorphic calls is larger.

polymorphism.targetCacheMissRate.bin This metric shows as a percentage how often a call site switches between receiver types. It represents the miss rate of an idealized branch target buffer [Dri01]. It is always lower than the corresponding inline cache miss rate, as measured by the polymorphism.receiverCacheMissRate.value, since different receiver types can be result in the same target being invoked. This metric can also be heavily influenced by the order in which target methods occur.

3.4.4 Memory Use

Understanding the memory use of programs is very important when trying to understand their behaviour, especially in modern, garbage-collected languages. Specific memory use patterns can be exploited by various compiler and runtime optimizations. Dynamic metrics that measure the amount and properties of dynamically-allocated memory are presented next.

Allocation Density

In order to measure the how memory-hungry a program is, one can look at the rate of memory allocation in a program. There are two variations:

memory.byteAllocationDensity.value This metric measures the number of bytes of memory that are allocated, on average, per kbc. The application-only version of the metric only counts objects that have a type that is a user-defined class, whereas the whole program version includes all dynamically allocated objects. Arrays are implemented as objects in Java, and are included in the computations as instances of a library class. This metric is platform-dependent, because the number of bytes in the object headers is dependent on the JVM implementation.

	Metric	EMPTY	BISORT	EM3D	JACK	JAVAC	SOOT	SABLECC
All	memory.averageObjectSize.value	191.6	34.9	443.5	53.8	41.4	35.2	62.0
	memory.byteAllocationDensity.value	1684.216	8.917	36.069	313.836	131.801	290.029	343.095
	memory.objectAllocationDensity.value	8.790	0.255	0.081	5.831	3.180	8.235	5.536
	memory.objectSize.bin(8)	0.6%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
	memory.objectSize.bin(16)	14.7%	0.5%	3.6%	34.6%	14.6%	46.3%	22.3%
	memory.objectSize.bin(24)	32.8%	97.3%	8.1%	39.8%	41.9%	33.9%	40.8%
	memory.objectSize.bin(32)	8.7%	0.3%	2.1%	4.2%	20.7%	1.0%	7.2%
	memory.objectSize.bin(40)	9.5%	0.3%	20.6%	7.1%	6.2%	6.5%	12.8%
	memory.objectSize.bin(48-72)	21.7%	0.9%	6.1%	12.7%	11.4%	10.6%	6.5%
	memory.objectSize.bin(80-136)	7.2%	0.3%	2.7%	0.7%	4.2%	1.0%	5.3%
	memory.objectSize.bin(144-392)	3.9%	0.2%	7.1%	0.5%	0.8%	0.5%	5.0%
	memory.objectSize.bin(400+)	0.9%	0.1%	49.7%	0.4%	0.2%	0.2%	0.2%
Application	memory.averageObjectSize.value	N/A	24.0	39.9	31.2	29.9	21.1	23.3
	memory.byteAllocationDensity.value	N/A	5.895	0.680	19.056	69.435	152.356	64.138
	memory.objectAllocationDensity.value	N/A	0.246	0.017	0.610	2.320	7.222	2.758
	memory.objectSize.bin(8)	N/A	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	memory.objectSize.bin(16)	N/A	0.0%	0.2%	8.4%	14.8%	58.8%	49.9%
	memory.objectSize.bin(24)	N/A	100.0%	0.0%	1.6%	22.7%	30.9%	19.8%
	memory.objectSize.bin(32)	N/A	0.0%	0.0%	81.1%	45.1%	0.8%	20.2%
	memory.objectSize.bin(40)	N/A	0.0%	99.8%	8.9%	11.3%	8.8%	10.1%
	memory.objectSize.bin(48-72)	N/A	0.0%	0.0%	0.0%	6.1%	0.7%	0.0%
	memory.objectSize.bin(80-136)	N/A	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	memory.objectSize.bin(144-392)	N/A	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	memory.objectSize.bin(400+)	N/A	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 3.8: Memory metrics

memory.objectAllocationDensity.value This metric measures the number of objects that are allocated, on average, per kbc.

From the data presented in Table 3.8, it can be observed that the memory allocation density varies widely. The EMPTY program shows a surprisingly high allocation density, 1684 bytes per kbc. This shows that system startup and class loading of library methods is quite memory hungry. Of the remaining benchmarks, BISORT and EM3D have fairly low densities, while the rest are quite high, with SABLECC having the highest at 343 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.

The discriminating nature of the allocation density metrics is easy to see from the data presented in Table 3.8. The metrics, both in their whole program and application versions, are sensitive to changes in the program inputs, and therefore are not very robust. For example, changing the input of the SOOT benchmark halved its allocation densities. JAVAC, JESS and RAYTRACE also exhibit similarly large variations.

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 of the metrics, where the number of bytes/objects allocated per kbc is computed per execution time interval, and not just once for the entire execution. Such extensions are planned as future work (see Section 6.2.2).

Object Size Distribution

In order to further study the memory requirements of applications, the next metrics look at the size of the objects being dynamically allocated. Knowing this information is useful for various optimization strategies, as well as manual memory management techniques. For example, if small objects are frequently allocated (and deallocated), it might be a good idea to manage them in a pool. However, pooling large objects may artificially increase the memory footprint of an application, and is probably not desirable.

memory.averageObjectSize.value This metric measures the average size of the objects that are dynamically allocated. This metric can be computed by taking the ratio of memory.byteAllocationDensity.value to memory.objectAllocationDensity.value. Because the size of the object headers is included in the total size of the objects, this metric is platform-dependent.

From Table 3.8, the EMPTY benchmark allocates large objects. This is due to the fact that during class loading and JVM initialization, several data structures have to be built. Many of them correspond to an internal representation of the classes being loaded. The EM3D benchmark allocates large objects (actually arrays of `Node` objects), but has a very low object allocation density. This pattern is characteristic of programs that first build a data structure, then perform computations on it. All other benchmarks presented in Table 3.8 allocate fairly small objects.

The memory.averageObjectSize.value metric is much more robust than the allocation density metrics. Intuitively, the application-only version of the metric is generally more robust than the whole-program version. Different optimizations, as well as changes in program input, had little effect of the metric value for all of the benchmarks for which multiple inputs were available.

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

memory.objectSize.bin This metric shows as percentages the proportion of allocated objects which fall within various size ranges. In order to factor out implementation-specific details of the object header size, bin 0 is used 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. The value of h is of course platform-dependent[3]; a typical value used by the major JVM implementations is currently 8 bytes.

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 $h + 48$. Readers should note that the sum of all bins should be 100%.

Table 3.8 presents empirical data for this metric. With the exception of EM3D, all benchmarks seem to allocate relatively small objects, indicating that the most frequently used user objects contain relatively few fields. The BISORT benchmark stands out because all its application objects are in one bin, objects of size 24 bytes. This is explained by

[3]A command-line option is provided in our tool to set the size of the object header.

	Metric	RollerCoaster	Telecom	Volano-Client	Volano-Server
All	concurrency.lockDensity.value	18.142	4.259	8.730	5.593
	concurrency.lock.percentile	14.1%	20.6%	10.8%	11.7%
	concurrency.contendedLockDensity.value	0.271	0.088	0.017	0.017
	concurrency.contendedLock.percentile	50.0%	30.0%	28.6%	50.0%
Application	concurrency.lockDensity.value	32.800	5.503	7.949	9.467
	concurrency.lock.percentile	14.3%	50.0%	23.1%	25.0%
	concurrency.contendedLockDensity.value	0.100	0.099	0.006	0.165
	concurrency.contendedLock.percentile	50.0%	100.0%	100.0%	88.9%

Table 3.9: Synchronization metrics for multithreaded benchmarks

the fact that the benchmark operates on a tree of `Value` objects. As with the memory.-averageObjectSize.value metric, the memory.objectSize.bin metric is generally robust with respect to program input and program optimizations.

3.4.5 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 whether it is 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.

	Metric	Automata	Automata-NoOutput	DB	Jack	SableCC	Empty
All	concurrency.lockDensity.value	6.677	1.266	14.673	15.748	12.764	1.528
	concurrency.lock.percentile	15.1%	43.2%	2.9%	19.2%	5.6%	40.9%
	concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000	0.000
	concurrency.contendedLock.percentile	N/A	N/A	N/A	100.0%	33.3%	N/A
Application	concurrency.lockDensity.value	0.000	0.000	0.017	1.142	0.000	N/A
	concurrency.lock.percentile	N/A	N/A	50.0%	100.0%	N/A	N/A
	concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000	N/A
	concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	N/A	N/A

Table 3.10: Synchronization metrics for single-threaded benchmarks

An important criterion in optimizing synchronization usage is to know whether a program does a significant number of lock operations (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 metric measures the average number of lock (`monitorenter`) operations per kbc. Programs which frequently pass through locks will have a relatively high density. Because synchronized blocks are often defined without knowing whether more than one thread will be running, this metric is irrespective of any actual concurrency.

Tables 3.9 and 3.10 show metrics for benchmarks with the highest lock density of the benchmark suite. This includes Automata, DB, Jack, RollerCoaster, SableCC, Telecom, and Volano (Volano-Client and Volano-Server), spanning a lock density range from 18.142 for RollerCoaster to 1.528 for Empty. RollerCoaster, Volano, and Telecom 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, AUTOMATA in particular, is less intuitive, and merits further investigation.

The actual AUTOMATA code itself consists of iteratively applying arithmetic operations on arrays. However, as mentioned in Section 3.4.2, each iteration within AUTOMATA 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 AUTOMATA-NOOUTPUT in Table 3.10); in this case the lock intensity drops to 1.266, below that of the EMPTY program.

SABLECC does not do any locking itself, but does through frequent invocation of library methods (I/O and `String`s). 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 SPECjvm98 benchmarks [ADG+99], while DB has the highest absolute number of total synchronizations [KKO02].

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 the current metric we define this as the percentage of locks responsible for 90% of lock operations.

From Table 3.10, the EMPTY benchmark has 40.9% 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. SABLECC and DB have the smallest number of hot locks, just 5.6% and 2.9% respectively. Hot spots here exist, but at least for SABLECC they are contained in the library code.

concurrency.contendedLock.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 [BKMS98]. 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, ROLLERCOASTER and TELECOM, 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 [ADG+99, KKO02].

concurrency.contendedLock.percentile This metric shows the existence of hot spots of contention. In the suite only highly multithreaded programs, TELECOM, VOLANO, and ROLLERCOASTER, have percentiles significantly less than 90%. ROLLERCOASTER has no contention hotspot since it has a concurrency.contendedLock.percentile 50%; 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 [ADG+99], and so contention hot spots are less likely.

While the TELECOM benchmark has a small percentile for the whole program, the contention on the locks that are found in its own code is completely evenly distributed. This is due to the fact that the benchmark tries to reduce lock contention through the use of separate locks for each of the 15 resources, effectively eliminating contention hotspots in its own locks. However, 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 for the whole program.

The contention that is found in the whole-program version of the concurrency.contendedLock.percentile metric for single-threaded benchmarks like JAVAC and JACK also deserves some explanation. The fact that these benchmarks obtain a concurrency.contendedLockDensity.value value of 0 [4] indicates that contention does not occur frequently. The observed contention is due to the fact that all Java applications are in practice multi-

[4]The actual metric value is in fact non-zero, but is reported as 0 due to rounding.

threaded, since many JVM implementations rely on additional threads to perform runtime support functions. For example, the lock contention in this case is due to the garbage collector attempting to run `finalize()` methods from its own thread.

Both concurrency.lockDensity.value and concurrency.lockDensity.percentile are potentially non-robust metrics because the order in which different threads acquire the locks may have a very significant impact of the actual lock contention. Such metrics are also highly platform dependent: different thread scheduling policies are likely to have an impact on the observed lock contention.

Chapter 4
**J* Framework

Empirical validation is an essential part of the metric development process. However, dynamic metrics do not yet form a well-defined corpus of calculations. Because individual metrics were motivated from ongoing investigations into program behaviour, it was necessary to build a tool to allow new metric calculations to be easily added and tested.

This chapter describes the **J* framework, a complete system for gathering trace data from runs of Java programs and performing dynamic analyses on that input (offline). Section 4.1 describes the major goals that the tool had to achieve. Section 4.2 presents an overview of the design of the framework, along with a discussion of how it addresses important issues. As a proof of concept, Section 4.3 lists applications of the tool to dynamic analysis tasks other than dynamic metric computations.

4.1 Goals

Because the development of individual metrics stemmed from investigating the behaviour of programs, it was necessary to build a framework which would not only make it possible to investigate dynamic metrics with a high level of freedom, but which would also be extensible in order to accommodate other possible dynamic analysis needs. **J* was therefore intended to be a general-purpose dynamic analysis framework rather than a specialized metric computation tool.

The main factor which influenced the development of **J* was the need for flexibility and extensibility, in terms of both the data side and the analysis side. In particular, it was necessary to ensure that new dynamic metrics could be quickly and easily added, modified or removed. Because there are multiple ways of collecting execution data, each of which offered different possibilities and constraints, it was also necessary to make sure that **J* was not tied to any particular data source. This mandated the design of two distinct tools, one to collect execution data and one to perform the analyses.

On the other hand, the amount of computations that the tool has to perform imposes another constraint: the performance of **J* had to stay in an acceptable range in order to make sure that non-trivial benchmarks could be successfully analyzed.

4.2 Design and Implementation

The **J* framework consists of two major components: a profiling agent which records dynamic information regarding the execution of a program in the form of an *execution trace*, and an analysis back-end which uses the traces as input for various dynamic analyses. This design allows either component to be used independently. A third component of the system is a versatile web-based interface which allows users to view the metric results in a convenient, user-friendly way.

Figure 4.1 shows an overview of the **J* framework. Individual components of the framework will be discussed in details next. Section 4.2.1 describes the major features of the profiling agent. Section 4.2.2 presents the design of the analysis back-end. Section 4.2.3 discusses the web interface to the metric database.

4.2.1 Agent

Execution traces in **J* are formed of a trace header followed by records of runtime events, and use a simple, generic and extensible trace format. This simplifies the task of collecting data from multiple sources. The main trace generator uses the built-in Java Virtual Machine Profiling Interface (JVMPI) [JVMPI] to dynamically receive events from any JVM which implements the interface, which it then serializes into a single event stream. The profiling

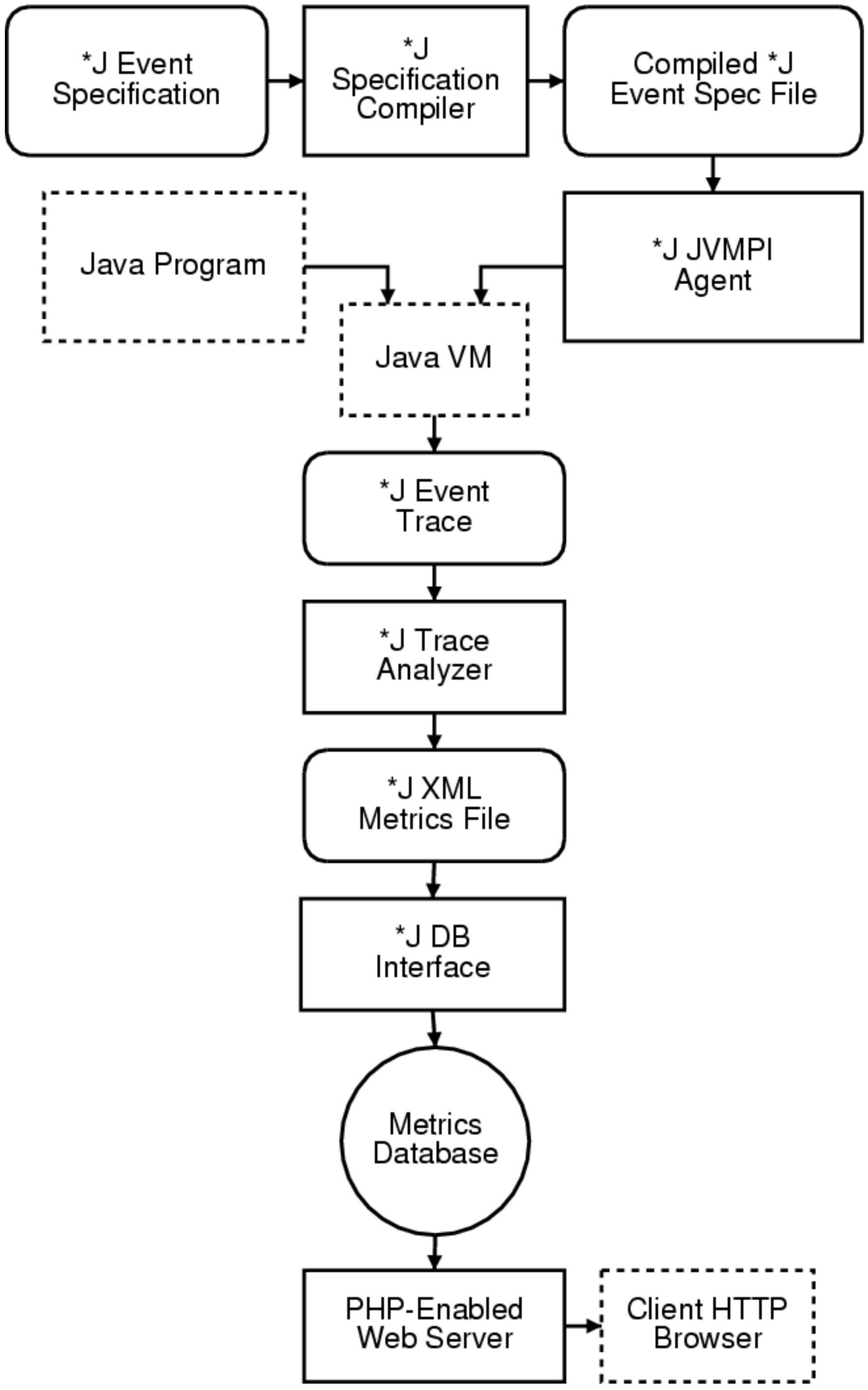

Figure 4.1: The **J* framework

agent is compatible with any JVM that implements the JVMPI.

Flexibility

Because not all of the events that are available through the JVMPI may be needed for the analysis phase, **J* uses an adaptive trace format that can be customized via *specification files*. Specification files are defined in a pattern-based domain-specific language for simplicity. For example, Listing 4.1 shows a tiny specification file, which will lead to only `ClassLoad`, `GCStart` and `GCFinish` events being included in the trace.

```
default {
   recorded: yes; # Record all events for which there is a definition
}

event ClassLoad {
   class_name: yes; # Record class name field
   methods: yes; # Record method table field (implies method count field)
}

event GC* {
   # 'GC*' matches GCStart and GCFinish
}
```

Listing 4.1: Simple event specification

Specification files must be compiled before being used with the agent. This allows the compiler to perform a dependency check to make sure that a sane specification is received. For example, it is not possible to record the `methods` field without recording the `num_methods` field as well (an internal array size); the compiler will thus introduce the missing dependency. Compiling the specification also has the additional advantage of producing a precise binary encoding of it, which is then including as an integral part of the trace header. This allows data consumers to know precisely what kind of data is available in the trace.

In order to make the trace format more generic and easily extensible, the trace header also contains a section reserved for attributes. The attributes used in **J* traces are very similar in spirit to those found in the compiled representation of Java classes [LY99]. Attributes allow data producers to encode information in the trace in a portable way. For example, the particular class path that was used to generate the trace can be included as an attribute. The trace consumer is then free to use this information if it needs it.

Trace Size

Complete execution traces are well-known to occupy a very large amount of space even for relatively small benchmark program executions. When recording frequent events, such as executed bytecode instructions, the amount of disk space may well be insufficient to store the uncompressed trace file. This obviously limits the amount of information that can be recorded unless a means of reducing the trace size is found. Fortunately, the data the composes execution traces is usually highly predictable, and can be compressed very effectively.

The most frequent kind of event that is found in the traces produced by **J* corresponds to the execution of a bytecode instruction. Such events are in fact so common that recording all of them in a naive way would require many gigabytes (GB) of storage space even for relatively small benchmarks. **J* therefore tries to predict what the next executed bytecode will be, keeping a count of how many of them were successfully predicted. Then, it will merge correctly predicted events into a single one. Note that there is absolutely no trace size penalty incurred for mispredictions; in this case the profiling agent would fall back to the original encoding. **J* employs a very simple predictor, and yet achieves significant trace compression. Table 4.1 shows the total size (in bytes) of traces for the HELLOWORLD benchmark (which contain over half a million bytecode execution events for the VM startup) using no predictor and the default predictor. Empirical results suggest that the prediction of bytecode executions is an effective trace compression scheme, as it achieves a reduction in trace size of over 65% in the case of the HELLOWORLD benchmark.

Generally, execution traces contain a significant number of memory addresses. The traces produced by **J* are no exception. Such addresses often exhibit good locality; as

	Predictor		% Difference
	None	Default	
Trace size (bytes)	6 950 804	2 381 852	65.7%

Table 4.1: Trace compression for HELLOWORLD using a predictor

	Compression		% Difference
	None	`gzip`	
Trace size (bytes) – No predictor	6 950 804	428 567	93.8%
Trace size (bytes) – Default predictor	2 381 852	194 321	91.8%

Table 4.2: Trace compression for HELLOWORLD using `gzip`

a result, execution traces generally respond very well to ordinary compression schemes. The **J* profiling agent is able to directly generate traces using the `gzip` [GZIP] format. Table 4.2 shows the amount of compression that is achieved using the `gzip` format with traces using no predictor or the default predictor.

Combining both trace size reduction techniques can therefore achieve a total compression of 97% of the original trace. Better tailored compression algorithms, such as the the ones used by the STEP framework [Bro03], can reduce the trace sizes even further. Integration of STEP and **J* is planned as future work (see Section 6.2.4).

**J* also supports splitting of the trace files into multiple, smaller trace segments, if compression techniques are either not used or still insufficient. It also supports writing the trace to a FIFO file (also called "named pipe"), eliminating the need to store the trace on disk altogether.

JVMPI-related Issues

The JVMPI was selected as a source of trace data primarily because it can work with the major virtual machine implementations, and provides access to information that is impossible to obtain by instrumenting the application code. Unfortunately, while the JVMPI is reasonably ubiquitous it has several drawbacks which affect the performance and complexity of the profiler, as well as the kind of data that can be recorded.

The most obvious drawback associated with the JVMPI is the fact that its callback mechanism is inherently slow for frequent events, which results in a very significant increase in execution time even for simple benchmark programs. For example, running the

JAVAC benchmark using an empty profiling agent slows down the execution by a factor of 6.5.

Moreover, the JVMPI does not guarantee complete trace data—some events may be skipped during JVM startup, this requires non-trivial internal state in order to track and handle missing events. The profiling agent has to keep an internal representation of all trace entities in order to detect missing entity definitions, and explictly request the corresponding events from the JVM. However, such requests have to be placed while respecting the constraints imposed by the internal state of the virtual machine; failure to do so would most likely result in its immediate failure. Producing a correct JVMPI agent is therefore a complex task.

A further limitation of the JVMPI is in the data it provides. Not only are event types fixed (limiting metric possibilities), but even the event data can be insufficient: the JVMPI reports instruction executions using code offsets, and so locating the actual opcode of the executed bytecode requires classfile parsing. The JVMPI also does not allow the profiler to inspect the contents of the execution stack. The only way to keep track of object references at the level of executed instructions is thus to simulate the entire execution, which of course imposes a very high overhead in both performance and code complexity.

4.2.2 Analyzer

The trace analyzer component takes as input the traces produced by the profiling agent. It is then responsible for applying any number of dynamic analyses (*operations*) to the data stored in the trace.

Flexibility

Conceptually, the trace analyzer uses a pipeline design through which events are sent. Dynamic analysis operations are placed along this pipeline, and are free to manipulate all events that flow through it, for example by inspecting, modifying, replacing or terminating the processing of events. Because the order in which an event travels through the set of analyses is determined by their respective order in the pipeline, it is possible for some analyses to execute with a higher priority in order to provide services to subsequent ones.

This design is the key to the flexibility in **J*.

**J* provides several kinds of analyses in its standard library:

- **Services**: Services are helper operations that decouple common tasks from the specific analyses by making their results available to other operations. Most services that are distributed as part of the standard operation library are mutator operations, i.e., operations that modify the event objects that are propagated to other operations.
- **Metric operations**: Metric operations are responsible for computing specific dynamic metrics.
- **Printers**: Printer operations simply convert some information from the execution trace into an alternate representation. **J* includes simple printers which simply output the trace in a human-readable format, and more involved ones such as the dynamic call graph printer (discussed in more details in Section 4.3.2).
- **Propagation operations**: Propagation operations perform propagation of data in the dynamic control flow graph (CFG) of the application. This is further discussed in Section 4.3.

Analysis operations in **J* are organized hierarchically as `Pack`s and `Operation`s. `Pack` objects are containers for other `Pack`s or `Operation`s, whereas `Operation`s perform computations. `Pack`s and `Operation`s can be added, removed or redefined. This hierarchical formation is conceptually clean and 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.

Each pack maintains a uniquely determined ordering of the operations that it contains, which in turn determines the order in which they appear in the processing chain. Events can therefore be thought of as "flowing" through the hierarchy, which constitutes the pipeline. Because of the large amount of information that is to be processed for each trace file, however, an event dispatch system that simply propagates events through the `Pack` tree is impractical. Instead, **J* preprocesses this tree and generates a mapping from events to sets of `Operation`s, effectively "flattening" the hierarchy while keeping the relative ordering of operations. `Operation`s are required to specify which events they want to receive. Only operations interested in an event will receive it, thus eliminating the cost of recursively traversing the tree of analyses for each event.

Extensibility

In order to support the addition of new dynamic analyses without having to modify the core of the framework, **J* provides several extension mechanisms.

For instance, **J* uses the notion of *storage container* to allow user-defined data storage to be associated at runtime with trace entities, such as classes, methods, objects, bytecode instructions, and so on. There are several kinds of data that can be associated with storage containers, such as counters and object references. Any operation can dynamically request more storage space from any storage container class; all existing instances of that class will automatically be notified of the change and reserve the additional storage space. Without this support from the framework, a simple computation such as counting the number of times each bytecode instruction is executed would require either a modification of the framework to add a field to the `Instruction` class, or the use of hash tables.

Moreover, it had to be possible to gather data from different sources. To this end, **J* supports the creation of custom trace reader objects, which allow the tool to read traces stored in other formats. This can be easily achieved since all trace file readers only have to implement the very simple `TraceReader` interface. Communication between the **J* core and trace readers is only achieved through queries that are placed by the framework, and belong to one of the following categories:

- **Contents queries**: contents queries are issued by the framework in order to determine if some particular information is available from the trace file. Because all `Operation`s are required to explictly state their event dependencies, content queries are used to verify that all dependencies for a particular `Operation` are met before allowing it to execute. The **J* framework performs queries for each event individually, and the trace reader must respond with an event descriptor which contains information regarding the status of the event (whether or not it is available), and its available fields.

- **Attribute queries**: attribute queries involve retrieving information concerning trace attributes. For example, if the class path that was used to generate the trace is included in the trace, then it can be accessed from another part of the framework using atttribute queries.

- **Event queries**: event queries are issued in order to request the next event from a trace reader. Because in general it is not possible to know the number of events in advance, the end of the trace is signalled by returning `null` in response to this query. Additionally, a trace reader can be asked to provide the number of events that it has successfully returned in response to event queries so far.

The simplicity of the communication between trace readers and the rest of the framework is further increased by the fact that apart from guaranteeing that no request will be issued before the trace reader has been initialized, there is no restriction on the ordering of the requests; different kinds of requests can even be interleaved.

The analyzer also relies on its own class file parser, which can easily be extended using custom attribute readers, which simply have to be registered with the class file parser. Attribute readers rely on the *factory* design pattern [GHJV95] to instantiate new instances of the `Attribute` class.

Performance and Efficiency

Although the analysis is performed offline, it was necessary to ensure that the tool would run in practical time. A right balance between performance and flexibility had to be achieved. The large amount of data that has to be processed for most benchmarks makes this task especially challenging, and required careful design to keep the memory footprint of **J* as small as possible.

In order to increase the efficiency of the framework pooling techniques are used extensively. For example, a single instance of each kind of event is allocated at runtime, thereby saving a significant amount of memory. Also, because allocating objects and performing garbage collection are relatively expensive operations, reducing the number of allocations has a positive impact on the overall performance of the framework.

Caching techniques are also used in many situations, and help considerably to keep the number of expensive computations as low as possible. For example, many entity resolution code segments use caches in order to avoid costly lookups in data structures. Also, **J* possesses many highly specialized data structures in order to avoid the typically high overhead that is associated with the ones provided in the Java standard libraries (e.g., for primitive types used in hash maps).

Ease of Use

**J* is designed to make the implementation of new dynamic analyses as quick and pain-free as possible while making sure that the tool will work in practical time. The analyzer itself is entirely written in Java; it therefore relies on inheritance to simplify most of the work related to the implementation of new analyses.

New dynamic metrics can be easily implemented by extending the `AbstractMetricOperation` class, and providing only a minimal amount of functionality. All `Operation`s in **J* are required to provide a set of *event dependencies*, and to specify how to handle the events that they receive. Additionally, metric operations are required to implement two specific callbacks. One of them is used to ask the metric operation class to compute and return its metric data, the other is used to instantiate a `MetricRecord`, which allows **J* to associate computations with arbitrary metric spaces, as discussed in Section 3.3. As a result, the entire code which is needed to implement the computations for some entire metric catgories, such as the pointer category, can easily be written within a few minutes.

In **J*, each operation and/or pack is responsible for generating its own output. This is flexible, but not always convenient, and so the framework uses the *visitor* pattern [GHJV95] to walk the `Pack` tree before and after the computation, therefore providing a simple way to collect the output from different analyses into a single output file. For example, a default metric walker will collect all metric data from the enabled `MetricOperation`s, and emit it in the form of an XML or a text document.

4.2.3 Web Interface

The current set of output processing tools focuses on the XML file that is emitted by **J* for the dynamic metric results. An XML format was selected to store the metric data because of the ease with which XML data can be processed or converted to other formats.

For example, **J* includes several XSLT style sheets which allow the XML metric data files to be transformed into various other formats, including HTML or plain text. This makes it possible to view the information in common web browsers or text editors.

While XSLT style sheets are convenient for processing a small number of files, more complex investigations involving many benchmarks require a more versatile approach. Therefore, **J* includes a parser for the XML files that is designed to insert the data that they contain into a central database, which can then be used as the basis of a dynamically generated website.

This web interface[1] has the inherent advantage of allowing multiple users to view and query the data simultaneously, as well as being easy to maintain. Publishing new results merely requires them to be inserted into the database, and makes it possible to distribute them within seconds. The web interface allows users to look at metrics or benchmarks individually, but also supports various advanced features such as benchmark comparisons and complex search using custom queries. It can be used to easily build tables which summarize the results, and includes sorting features on almost every page.

Such a website 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 selected benchmarks can then be automatically packaged and downloaded in various compressed archive formats.

4.3 Extending the **J* Framework

In order to demonstrate the flexibility and extensibility of the framework, three case studies are presented. Section 4.3.1 explains the necessary steps to add a simple metric computa-

[1]The initial version of the web interface has been provided by Tobias Simon.

tion to the framework. Section 4.3.2 presents an extension which produces dynamic call graphs. Section 4.3.3 discusses extensions related to the study of the behaviour of AspectJ programs.

4.3.1 Adding a Simple Metric Computation

One of the most common tasks that has to be performed with **J* is the addition of a new metric computation. In order to demonstrate the necessary steps to implement and use new metric operations, a new metric, example.instructionMix.bin will be used as an example. This metric reports the proportion of the executed bytecodes that is due to each individual opcode, and thus has one bin per valid Java bytecode.

In order to provide as much flexibility as possible, all metric operations are only required to implement the `MetricOperation` interface. In almost all cases, extending the `AbstractMetricOperation` class is preferred as it readily provides some commonly used facilities. The skeleton from which the new metric operation will be built is as follows:

```
package examples;

import starj.EventBox;
import starj.dependencies.*;
import starj.toolkits.metrics.*;

public class InstructionMixMetric extends AbstractMetricOperation {
   public InstructionMixMetric(String name, String description) {
      super(name, description);
   }

   public OperationSet operationDependencies() {
      /* State which other operations provide information used by this
         operation */
      return super.operationDependencies();
   }

   public EventDependencySet eventDependencies() {
      /* Register to receive required events */
      return super.eventDependencies();
   }

   public void apply(EventBox box, MetricRecord[] records) {
        /∗ Event callback ∗/
    }

    public MetricRecord newRecord() {
        /∗ Create and return a new custom metric record to store computation
           state ∗/
        return new MetricRecord(){};
    }

    public void accept(MetricVisitor visitor, MetricRecord record) {
        /∗ Compute the metric values for the given record ∗/
    }
}
```

Listing 4.2: Skeleton for the `InstructionMixMetric` class

The code from Listing 4.2 forms a valid—although useless—metric operation class. In order to perform a meaningful computation, the `InstructionMixMetric` class has to receive events from the execution trace. In this particular case, only one type of event needs to be monitored: instruction start events. Such events are triggered whenever the JVM is about to execute a bytecode instruction. Events in **J* are represented by objects, and are sent to the operations through a callback mechanism. In order for any operation to receive execution events, it must first register them as dependencies. Event dependencies are specified using the `eventDependencies()` method, and used by **J* to determine if any given operation can be allowed to execute. Event dependencies can be *required* or *optional*. If an operation has unmet required dependencies, it will automatically be disabled. *Optional* dependencies, indicate that a given operation wants to receive the certain events if they are available, but that it should not be disabled otherwise. Nonetheless, an operation will be disabled if none of its required or optional event dependencies can be satisfied.

```
public EventDependencySet eventDependencies() {
    /∗ Register to receive INSTRUCTION_START events ∗/
    EventDependencySet deps = super.eventDependencies();
    deps.add(new EventDependency(
            Event.INSTRUCTION_START, // Event type ID
            new TotalMask( // Field dependencies
                    Constants.FIELD_RECORDED
                    | Constants.FIELD_OFFSET
            )
    ));
    return deps;
}
```

Listing 4.3: Specifying event dependencies

The code from Listing 4.3 registers a single event dependency. For performance reasons, **J* refers to event types using numerical identifiers. Each event dependency consists of an event identifier and a field mask. In this case, the required event type identifier is `INSTRUCTION_START`, since the metric operation needs to monitor instruction start events. In order to compute the instruction mix, only the opcode for each `InstructionStartEvent` object is required. However, the current JVMPI implementation does not provide this value directly; it reports bytecode offsets instead. Therefore, the field mask that is specified for this event consists of two field descriptors: the `FIELD_RECORDED` descriptor specifies that the event under consideration should be present in the trace file, and the `FIELD_OFFSET` descriptor specifies that the `offset` field of the event must be present as well. A `TotalMask` instance is created because all field descriptors are required for the dependency to be met. In contrast, a `PartialMask` instance would only require that one of the descriptors be available. Combining masks is also possible, for example using the `OrMask` or `AndMask` classes, but not required for this example.

Because resolving the opcode from the limited information that is provided by the JVMPI interface is a fairly common but non-trivial task, the **J* framework provides facilities to perform the necessary bookkeeping and provide the information to other operations. The `InstructionResolver` operation handles opcode resolution. It then makes this information available to subsequent operations by mutating the `InstructionStartEvent` objects to include, among other things, the opcode which corresponds to the executed bytecode. In order to make use of this information from the `InstructionMixMetric` class, the `InstructionResolver` operation needs to be declared as an operation dependency in the `operationDependencies()` method, as illustrated in Listing 4.4. This ensures that if the `InstructionResolver` class is disabled, then the `InstructionMixMetric` class will be disabled as well. This can happen if, for example, the `InstructionResolver` operation is manually disabled by a user, or in cases where the information that is stored in the trace file is not sufficient for it to perform its task.

```
public OperationSet operationDependencies() {
   /* Add InstructionResolver as a dependency */
   OperationSet deps = super.operationDependencies();
   deps.add(InstructionResolver.v());
   return deps;
}
```

Listing 4.4: Specifying operation dependencies

In this case, the `InstructionResolver` operation is a *Singleton* [GHJV95] whose instance can be accessed via the `v()` method, which simplifies the specification of the operation dependencies. References to regular operations can always be obtained from the singleton `Scene` instance.

The next step is to determine which information needs to be recorded in to compute the value for each bin. In this case, the computation is very simple and only requires recording the number of times that each kind of bytecode is executed. Because the JVM specification [LY99] states that there is a maximum of 256 possible opcodes, execution counts can be stored in an array.

Recall from Section 3.3 that the entire sample space for the metric computations can in fact be subdivided using arbitrary partitioning schemes. In order to relieve the operation implementor from the burden of managing such partitions, metric operations are required to store their computation state in a class that implements the `MetricRecord` interface. **J* will automatically associate one metric record to every sample space partition that is created by requesting new records as needed. Therefore, a private metric record class is defined as follows, along with the required factory method:

```
private class InstructionMixRecord implements MetricRecord {
   private long[] counts; // Execution counts per opcode
   private long unknown_count; // Unresolved bytecode count

   public InstructionMixRecord() {
      this.counts = new long[256];
      this.unknown_count = 0L;
   }
```

```
    public long getCount(short opcode) {
        return this.counts[opcode];
    }

    public long getTotal() {
        long total = 0;
        for (int i = 0; i < this.counts.length; i++) {
            total += this.counts[i];
        }
        return total;
    }

    public long getUnknownCount() {
        return this.unknown_count;
    }

    public void stepCount(short opcode) {
        this.counts[opcode] += 1L;
    }

    public void stepUnknownCount() {
        this.unknown_count += 1L;
    }
}

public MetricRecord newRecord() {
    /* Create and return a new custom metric record to store computation
       state */
    return new InstructionMixRecord();
}
```

Listing 4.5: Providing custom `MetricRecord`s

Note that the `InstructionMixRecord` class includes storage to keep track of "unknown" bytecodes, in case **J* fails to resolve some of the bytecode instructions. This can happen if, for example, some classes cannot be found during the analysis. Also note that all opcodes are declared as `short`s for convenience since Java does not possess an unsigned byte primitive type.

The code from Listing 4.5 provides sufficient information for **J* to manage the record by itself. The event callback can now be implemented:

```
public void apply(EventBox box, MetricRecord[] records) {
      // Obtain the current event
      InstructionStartEvent e = (InstructionStartEvent) box.getEvent();
      // Obtain the opcode of the executed instruction
      short opcode = e.getOpcode();

      // Check for a valid list of records
      if (records != null) {
         for (int i = 0; i < records.length; i++) {
            // For each record, increase the appropriate counter
            if (opcode >= 0) {
               ((InstructionMixRecord) records[i]).stepCount(opcode);
            } else {
               ((InstructionMixRecord) records[i]).stepUnknownCount();
            }
         }
      } // else: this event should not be taken into account
   }
```

Listing 4.6: Providing an event callback

Note that for performance reasons, iterating through all records is left to the implementor of the metric operation. This allows for a much more efficient implementation of the metric computations, since expensive computations which are common to all records can be performed only once.

The final step consists of computing the actual metric data and emitting it in a form that is recognized by the framework. **J* uses a *Visitor* pattern [GHJV95] for this task. Metric data is represented by objects which can be visited by any number of visitors. Different visitors can format the data in different ways. For instance, **J* provides a plain text visitor and an XML visitor. The code which performs the metric computation is as follows:

```
   public void accept(MetricVisitor visitor, MetricRecord record) {
      /* Compute the metric values for the given record */
      InstructionMixRecord r = (InstructionMixRecord) record;

      // Create a new BinMetric
      BinMetric inst_mix = new BinMetric("examples", "instructionMix");

      long total = r.getTotal();
      // Add the 'unknown' bin to the metric
      inst_mix.addBin(new SimpleBin(
            new StringKey("unknown"),
            new PercentageMetricValue(r.getUnknownCount(), total)
```

```
    ));

    // Add one bin for each valid opcode
    for (short opcode = 0; opcode < 256; opcode++) {
       String opcode_name = Code.getOpcodeName(opcode);
       if (opcode_name != null) {
          inst_mix.addBin(new SimpleBin(
                new StringKey(opcode_name),
                new PercentageMetricValue(r.getCount(opcode), total)
          ));
       } // else: not a valid opcode
    }

    // Make the specified visitor visit the new metric
    visitor.visit(inst_mix);
}
```

Listing 4.7: Emitting metric data

The code from Listing 4.7 performs the actual metric computation for each `MetricRecord` instance. The computed metric value is represented by a `BinMetric` object. `BinMetric` objects are themselves containers for `Bin` objects. In this case, only one kind of `Bin`—the `SimpleBin` class—is used. `Bin` objects always possess a key and a value. A bin key serves as an identifier for the bin, and must thus be unique. In this case, the type of key used is a `StringKey`, which allows the bins to be identified by the name of their associated opcode (or the string “`unknown`”). The `MetricValue` objects that are created are instances of the `PercentageMetricValue` class. If, for example, the absolute execution count for each bin was to be reported instead of the proportion of the executed bytecodes that it represents, `LongMetricValue` objects would have been used. Note that the formatting of the `MetricValue` objects is handled by the metric visitor.

While the implementation of the `InstructionMixMetric` is completed, there is one additional step that is required to make the operation usable. There are two possible ways to register a new operation with **J*. The newly implemented operation can be directly incorporated into the **J* source code. This solution has obvious drawbacks, and requires the recompilation of some parts of the toolkit each time that a change is made. A much more convenient solution involves the creation of a *driver class*, such as the one presented in Listing 4.8.

```
package examples;

import starj.*;

public class InstructionMixDriver {
   public static void main(String[] args) {
      // Obtain the RootPack instance from the Scene
      RootPack root_pack = Scene.v().getRootPack();

      // Create a new pack for the examples
      Pack examples_pack = new Pack("examples",
            "Contains example operations");
      root_pack.add(examples_pack);

      // Add a new instance of the InstructionMixMetric class to the
      // 'examples' pack
      examples_pack.add(new InstructionMixMetric("imix",
            "Instruction mix metric"));

      // Delegate all processing to the Main class
      Main.main(args);
   }
}
```

Listing 4.8: Implementing a driver class

The driver class from Listing 4.8 first obtains the `RootPack` instance from the `Scene` object. The `Scene` keeps track of global configurations, and is responsible for performing the analysis. The driver class then creates a new pack and adds it to the `RootPack`. An instance of the `InstructionMixMetric` class is also added to the "example" pack. Finally, the command-line arguments are passed on to the `Main` class for normal processing.

4.3.2 Producing Dynamic Call Graphs

Call graphs represent call relations between methods of a program. Traditionnally, call graphs have been approximated using static type analysis techniques, and have been used for performing interprocedural optimizations. Recently, dynamic call graph construction techniques have been proposed, such as in [QH04].

*J can be extended to compute dynamic call graphs from its execution traces. Several services that are part of the standard analysis library make the implementation a lot easier than it would otherwise be. For instance, *J can automatically notify an analysis when a new method is entered and exited, allowing it to provide callbacks to be executed upon such events. *J will also correctly associate a call site with the invocation, and will provide an access to the context in which the method invocation occured. This is the basis of the *propagation framework*, which is designed to facilitate the implementation of common dynamic propagation analyses.

Using such facilities, the implementation of the dynamic call graph analysis is very compact [2], and its full source code is reproduced in Listing 4.9.

```java
package starj.toolkits.printers;

import java.io.PrintStream;
import java.util.*;

import starj.*;
import starj.coffer.InvokeInstruction;
import starj.dependencies.OperationSet;
import starj.options.*;
import starj.toolkits.services.*;

public class CallGraphPrinter extends AbstractPrinter
      implements Propagation {
   private boolean use_dot = false;
   private Map edges;

   public CallGraphPrinter(String name, String description) {
      super(name, description);
      PropagationManager.v().addPropagation(this);
   }

   public CallGraphPrinter(String name, String description,
         PrintStream out) {
      super(name, description, out);
      PropagationManager.v().addPropagation(this);
   }

   public OperationSet operationDependencies() {
```

[2]The original version of the code was contributed by Ondřej Lhoták.

```
        OperationSet ops = super.operationDependencies();
        ops.add(PropagationManager.v());
        return ops;
    }

    public void init() {
        super.init();
        this.edges = new HashMap();
    }

    public void apply(EventBox box) {
        // Intentionally empty
    }

    public void propagate(InvokeInstruction call_site,
            MethodEntity new_method, ExecutionContext call_context) {
        MethodEntity current_method = call_context.getMethod();
        if (current_method != null) {
            Set targets = (Set) this.edges.get(current_method);
            if (targets == null) {
                targets = new HashSet();
                this.edges.put(current_method, targets);
            }

            if (!this.edges.containsKey(new_method)) {
                this.edges.put(new_method, null); // Insert placeholder
            }

            targets.add(new_method);
        }
    }

    public void unpropagate(ExecutionContext context) {
        // Intentionally empty
    }

    public void configure(ElementConfigArgument config, Object value) {
        String name = config.getName();
        if (name.equals("dot")) {
            if (value != null) {
                this.use_dot = ((Boolean) value).booleanValue();
            }
        } else {
            super.configure(config, value);
        }
    }

    public ElementConfigSet getConfigurationSet() {
        ElementConfigSet set = super.getConfigurationSet();
        ElementConfigArgument dot_arg = new ElementConfigArgument(
                "dot",
                "Specifies whether the 'dot' output format will be used",
                "Specifies whether the 'dot' output format will be used "
                      + "to print the call graph",
                false
        );
        dot_arg.addArgument(new BooleanArgument(
                "value",
                true, // required
                false, // not repeatable
                "boolean",
                "true or false"
        ));
        set.addConfig(dot_arg);
        return set;
    }

    public void done() {
        PrintStream out = this.out;
        Set keys = this.edges.keySet();
        if (this.use_dot) {
            out.println("digraph CallGraph {");

            // Output node descriptions
            out.println(" /* Nodes */");
            out.println(" node [shape=box];");
            for (Iterator i = keys.iterator(); i.hasNext(); ) {
                MethodEntity src = (MethodEntity) i.next();
                out.println(" " + src.getID() + " [label=\""
                      + src.getClassEntity().getClassName() + "\\n"
                      + src.getMethodName() + "\"];");
            }

            // Output edge descriptions
            out.println("\n /* Edges */");
            for (Iterator i = keys.iterator(); i.hasNext(); ) {
                MethodEntity src = (MethodEntity) i.next();
                Set targets = (Set) this.edges.get(src);
                if (targets == null) {
                   continue;
                }
                for (Iterator j = targets.iterator(); j.hasNext(); ) {
                   MethodEntity tgt = (MethodEntity) j.next();
                   out.println(" " + src.getID() + " -> " + tgt.getID()
                         + ";");
                }
            }
            out.println("}");
        } else {
            for(Iterator i = keys.iterator(); i.hasNext(); ) {
                MethodEntity src = (MethodEntity) i.next();
                Set targets = (Set) this.edges.get(src);
                if (targets == null) {
                   continue;
                }
                for(Iterator j = targets.iterator(); j.hasNext(); ) {
                   MethodEntity tgt = (MethodEntity) j.next();
                   out.println(src + " -> " + tgt);
                }
            }
        }
    }
 }
```

Listing 4.9: The `CallGraphPrinter` class

4.3.3 Measuring the AspectJ-related Metrics

AspectJ [Asp] is an increasingly popular aspect-oriented extension of the Java programming language. In order to study the behaviour of AspectJ programs, Dufour *et al*. [DGH+04] have developed a modified version of the AspectJ compiler which annotates the generated bytecode according to several kinds of overhead. **J* was then extended to make use of these code annotations and compute new, AspectJ-specfic dynamic metrics based on them. This required two different kinds of extensions to **J*. First, the new class file attributes that are generated by the modified compiler had to be correctly decoded. Next, new dynamic metrics had to be designed and implemented.

Making **J* recognize custom class attributes requires that a custom reader class along with its factory be implemented for each new attribute. Then, an instance of the factory class simply has to be associated with the attribute name. This can be accomplished by issuing a simple call to the current `ClassFileFactory` instance. At this point, all further management of the new attributes is handled by **J*.

Computing the new metrics is a more complex task because the tags need to be dynamically propagated along call edges in the dynamic control flow graph (CFG) of the application in order to provide a more accurate measure of each class of overhead. The generic propagation framework discussed in the previous example can be used for this task as well, allowing the implementor of the new analysis to almost exclusively focus on details of the propagation algorithm. In this example, the propagation rules are complex; nonetheless, the implementation of the propagation algorithm itself is very compact.

Once the progation analysis was implemented, implementing the new metrics was an almost trivial task, and only required very little work in order to implement a total of nine new metrics.

Therefore, none of the modifications that were required for this relatively complex task required modifying the core of the *J* framework. All modifications were also easy to implement because of the built-in support for propagation operations that *J* provides.

Chapter 5
Experiences

Chapter 3 has presented a set of dynamic metrics, along with empirical results demonstrating the potential usefulness of individual metrics in understanding program behaviour. This chapter discusses practical experiences with dynamic metrics, and shows not only the extent of the information that can be obtained from them, but more importantly how combined information from various metrics can be used to paint a better portrait of the overall behaviour of an application. Section 5.1 describes the benchmark programs that were used in this study. Section 5.2 presents specific case studies which demonstrate the usefulness of dynamic metrics for common program understanding tasks.

5.1 Benchmarks

This section lists the benchmarks that were used in this study, along with a short description of each of them.

AUTOMATA

> AUTOMATA is a simple one-dimensional, two-state cellular automaton simulator. This instance of the benchmark simulates an automaton with 100 cells for 20 time steps. Two additional computation sizes have also been used: 200 cells for 40 time steps, and 1000 cells for 100 time steps.

COEFFICIENTS

COEFFICIENTS computes the coefficients of the least square polynomial curves of degrees 0 to 20 for a set of 114 points in a real-valued coordinate system using a matrix pseudoinverse. An alternative problem size consists of the same computation for polynomials of degree 0 to 24 using a set of 162 points.

EMPTY

EMPTY is the empty program, i.e., the program which simply returns from its main method.

HELLOWORLD

HELLOWORLD is the well-known program that prints the string `"Hello, world"` to its standard output.

JLEX

JLEX is a lexical analyzer generator. This benchmark takes as input the syntax describing the commands of a simple media server, and generates a Java implementation of a lexical analyzer program which tokenizes character streams according to the specification.

LINPACK

LINPACK is a numerically intensive program which is commonly used to measure floating-point performance. It solves a dense 500x500 system of linear equations in the form $A\vec{x} = \vec{b}$ by performing a Gaussian elimination on the matrix. The matrix A is generated randomly.

ROLLERCOASTER

ROLLERCOASTER is an implementation of the classical “Roller Coaster” concurrency problem. A cart waits for passengers and goes for a ride, and repeats the process. This instance of the benchmark has 7 passenger threads and one cart thread.

The concept of “seat” in the benchmark is implemented using a non-blocking mutex for each seat. Passenger threads continuously try to acquire a free seat in the roller coaster. The cart thread continuously checks to see if it is filled with passengers. Once it is, it takes them for a ride, then kicks them off. This process is repeated 50 times.

TELECOM

A phone billing simulation program. The benchmark simulates a phone system with 15 users, and 5 threads trying to place calls between pairs of users. Users are then charged according to the duration of their calls. Apart from being multithreaded, the TELECOM benchmark has the particularity of being implemented using AspectJ.

VOLANO-CLIENT

The VOLANO benchmark [Volano] is a client-server chat room simulation program. This benchmark is the client side of the VOLANO benchmark executed for two iterations 3 chat rooms, depth 4.

VOLANO-SERVER

The server side of the VOLANO-CLIENT benchmark.

Ashes Suite Collection Benchmarks [Ashes]

SABLECC

An object-oriented framework compiler generator. The SableCC framework uses object-oriented techniques to automatically build a strictly typed abstract syntax tree (AST), and generates tree-walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance. This benchmark processes a grammar for version 1.1 of the Java programming language. The WIG grammar was also used as an alternative input for this benchmark.

SOOT

A Java bytecode transformation and optimization framework. The framework provides a number of static program transformations that are applied to a 3-address representation of Java bytecode called *Jimple*. This benchmark reads classes from its own `jimple` subpackage and emits them in the form of Jimple code. An alternative input consits of a class from SOOT's `coffi` subpackage.

JOlden benchmarks [CM01]

BARNES-HUT

Simulates the the motion of particles in space using a $O(n \log n)$ algorithm for computing the respective accelerations. This benchmark uses an oct-tree representation of 4K bodies in space.

BISORT

Sorts an array of 128K randomly-generated integers by creating two bitonic sequences and merging them. This benchmark performs both a forward sort and a backward sort.

EM3D

Simulates the propagation of electromagnetic waves through a 3D object. This benchmark uses an irregular bipartite graph which contains 2000 nodes of out-degree 100 representing electric and magnetic field values.

HEALTH

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. This benchmark performs the simulation for 500 time steps using 5 levels for health centers.

MST

Computes the Minimum Spanning Tree (MST) of a graph composed of 1K nodes using Bentley's algorithm.

PERIMETER

Computes the total perimeter of a region in a 64K binary image represented by a quad-tree. The benchmark creates an image, counts the number of leaves in the quad-tree and then computes the perimeter of the image using Samet's algorithm.

POWER

Solves the Power System Optimization Problem for 10000 customers, where the price of each customer's power consumption is set so that the economic efficiency of the whole community is maximized.

TSP

Computes an estimate of the best Hamiltonian circuit for the Travelling Salesman Problem using 10000 cities.

VORONOI

Computes the Voronoi Diagram for a set of 20000 points by using a divide-and-conquer method. Points are stored in a binary tree sorted by x-coordinate.

SPECjvm98 benchmark suite [Spec]

All benchmarks from this suite can be executed with three different input sizes (sizes 1, 10 and 100). The default configuration used for these benchmarks excludes the harness, which automatically selects the largest input size.

COMPRESS

Modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built.

JESS

JESS is the Java Expert Shell System is based on NASA's CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase run time the benchmark problem iteratively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus the inference engine must search through progressively larger rule sets as execution proceeds.

RAYTRACE

A raytracer that works on a scene depicting a dinosaur.

DB

Performs multiple database functions on memory resident database. This benchmark reads in a 1 MB file which contains records with names, addresses and phone numbers of entities and a 19KB file called `scr6` which contains a stream of operations to perform on the records in the file. The program loops and reads commands till it hits the 'q' command. The commands performed on the file include, among others:

- add an address
- delete and address
- find an address
- sort addresses

JAVAC

> This is the Java compiler from the JDK 1.0.2. [As this is a commercial application, no source code and no further information are provided.]

MPEGAUDIO

> This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. As this is a commercial application only obfuscated class files are available. The workload consists of about 4MB of audio data.

MTRT

> This is a variant of RAYTRACE, a raytracer that works on a scene depicting a dinosaur, where two threads each render the scene in the input file time-test model, which is 340KB in size.

JACK

> A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. The workload consists of a file named `jack.jack`, which contains instructions for the generation of JACK itself. This is fed to JACK so that the parser generates itself multiple times. Because this is a commercial application, no source code is provided.

5.2 Case Studies

5.2.1 Program Understanding

In this section, four benchmarks are analyzed and compared in terms of various dynamic metrics. This analysis is meant to demonstrate how dynamic metrics can be used to obtain

	Metric	COEFFICIENTS	COMPRESS	JAVAC	EMPTY
All	size.run.value	12887	14514	37831	7354
	data.arrayDensity.value	150.880	52.152	37.932	73.311
	data.floatDensity.value	202.808	0.000	0.072	1.987
	polymorphism.receiverCacheMissRate.value	0.1%	0.0%	8.8%	8.4%
	concurrency.lockDensity.value	0.352	0.000	7.897	1.528
Application	size.run.value	975	5084	26267	0
	data.arrayDensity.value	160.404	52.150	15.471	N/A
	data.refArrayDensity.value	80.713	0.000	3.543	N/A
	data.numArrayDensity.value	79.486	52.150	0.359	N/A
	data.floatDensity.value	217.956	0.000	0.000	N/A
	polymorphism.receiverArityCalls.bin(1)	100.0%	100.0%	72.6%	N/A
	polymorphism.receiverArityCalls.bin(2)	0.0%	0.0%	15.1%	N/A
	polymorphism.receiverArityCalls.bin(3+)	0.0%	0.0%	12.3%	N/A
	polymorphism.invokeDensity.value	65.973	16.532	71.771	N/A
	polymorphism.receiverCacheMissRate.value	0.0%	0.0%	7.2%	N/A
	memory.averageObjectSize.value	18.7052	29.02564	29.93472	N/A
	memory.objectAllocationDensity.value	0.013	0.000	2.320	N/A
	memory.objectSize.bin(8)	0.0%	0.0%	0.0%	N/A
	memory.objectSize.bin(16)	66.8%	42.0%	14.8%	N/A
	memory.objectSize.bin(24)	32.9%	41.7%	22.7%	N/A
	memory.objectSize.bin(32)	0.0%	0.0%	45.1%	N/A
	memory.objectSize.bin(40)	0.3%	0.3%	11.3%	N/A
	memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	N/A
	concurrency.lockDensity.value	0.000	0.000	0.670	N/A

Table 5.1: Metrics for benchmark analysis

a high-level overview of the behaviour of applications. The benchmarks that are considered are COEFFICIENTS, COMPRESS, JAVAC, and for startup cost comparison, the EMPTY benchmark. A table of metrics for each is shown in Table 5.1.

Program Size

As discussed in Section 3.4.1, these benchmarks form a progression from small to large. What is surprising is the relative impact of startup and library code. Considering the whole program version of the size.run.value metric, the relative sizes of these programs is far less apparent; e.g., COMPRESS is only 1.12 times larger than COEFFICIENTS, whereas in the application-only version of size.run.value metric the ratio is more than 5.2. Startup code, in fact, accounts for the majority of touched bytecode instructions in all but JAVAC. 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 application-only version of the data.-arrayDensity.value metric; the EMPTY program has no arrays, JAVAC has some array accesses (15.471), COMPRESS a significant but not large number (52.15), and COEFFICIENTS has the highest (160.404). These numbers correspond to a reasonable perception of the relative importance of array usage in these benchmarks. In the case of COMPRESS, there are enough application bytecode instructions executed to almost eliminate the relative effect of startup and other (infrequent) library calls; the whole-program and application-only versions of the metric thus obtain very similar values.

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. COEFFICIENTS has an almost equal density of accesses to numerical (primitive types) arrays as reference arrays (all object and array types), whereas the array density of compress comes entirely from numerical arrays; this is shown through the application-only versions of the data.numArrayDensity.value and data.refArrayDensity.value metrics. In fact, from an inspection of the source both benchmarks actually use almost exclusively numerical arrays. The significant difference between their specific array densities can be explained by the way arrays are represented in Java; in the case of COMPRESS, 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 COEFFICIENTS, 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, as shown in Table 5.2.

	Metric	Original	Loop. Inv.
All	size.run.value	12887	12894
	data.arrayDensity.value	150.880	122.153
	data.floatDensity.value	202.808	228.643
Application	size.run.value	975	989
	data.arrayDensity.value	160.404	129.877
	data.numArrayDensity.value	79.486	90.443
	data.refArrayDensity.value	80.713	39.200
	data.floatDensity.value	217.956	248.730
	polymorphism.invokeDensity.value	65.973	75.390

Table 5.2: Effects of loop invariant removal on COEFFICIENTS benchmark

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. Optimizations specialized to one-dimensional or just numerical array accesses may therefore not realize as much benefit as the general array density would imply. Alternatively, an optimization that recognizes rectangular two-dimensional matrices and converts them to one-dimensional matrices would likely be quite effective in COEFFICIENTS, but much less so in COMPRESS.

Use of floating point is relatively uncomplicated. The EMPTY program uses a small amount of floating point in startup, and JAVAC and COMPRESS use none in their application code — data.floatingPoint.value is 0 for both benchmarks. COEFFICIENTS, 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-orientation, 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). COEFFICIENTS 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.receiverArityCalls.bin metric, for instance shows that 100% of both COEFFICIENTS's and COMPRESS's `invokevirtual` or `invokeinterface` call sites reach exactly one class type; in other words, they are completely monomorphic. The lack of inheritance in COEFFICIENTS 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. Its non-zero polymorphism.receiverCacheMissRate.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, COEFFICIENTS 144.603 bytes, and COMPRESS 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 COMPRESS 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. COEFFICIENTS 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. COMPRESS allocates a few large arrays to use as buffers and tables, but otherwise does little allocation. COEFFICIENTS iteratively allocates two-dimensional arrays of increasing size, and this aggregate effect also shows up in a larger proportion of larger objects (≥ 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 EMPTY have the highest lock density (7.897 and 1.528 respectively). COEFFICIENTS has a low lock density of 0.352 and this rises only slightly for the loop invariant version (due to the decreased number of (invariant) calculations). COMPRESS unsurprisingly has a very low lock density (below 0.001), owing to its long internal calculations. All of these program except `javac` are thus minimally lock intensive.

5.2.2 Manual Optimizations

Understanding a program's behaviour helps to understand its performance bottlenecks. In this section, a case study is presented which aims to demonstrate how dynamic metrics can be used to guide manual code refactoring. A new benchmark, TWELVE, will be introduced in order to achieve this goal. While the benchmark itself is tiny, the techniques that will be described apply equally well to larger programs.

Understanding the benchmark

The TWELVE benchmark is a naive Java reimplementation of a famous obfuscated C program which outputs to its standard output the complete lyrics of song "The Twelve Days of Christmas". The C version of this benchmark has been studied by Ball in [Bal99], where the program was progressively "unobfuscated" using dynamic analysis techniques in order to understand its operation. The program uses a substitution cipher to encode all of the lyrics, and implements a selector algorithm to print verses of the song in the correct order. All of the lyrics of the song are encoded using two strings: one stores the encoded lyrics, and the other stores the cipher that is used to decode them. The original implementation is also entirely recursive. The TWELVE benchmark was given as a challenge problem for the Dagstuhl Seminar on "Understanding Program Dynamics".

The Java implementation of the benchmark is a trivial reimplementation of the same algorithm. It uses the same two strings as the original C version. It also allocates a single

	Metric	TWELVE	HELLOWORLD	LINPACK	JAVAC
All	base.instructions.value	7 670 314	529 290	8 613 320	2 018 819 228
	size.run.value	8 137	7 804	10 708	37 831
App	base.instructions.value	900 508	4	8 026 544	912 643 681
	size.run.value	117	4	749	26 267

Table 5.3: Size metrics for TWELVE

object correspoding to the application class, which stores references to both strings using non-static fields. While the original version of the program makes heavy use of pointer gymnastics, the Java version uses `substring` operations to implement the same functionality.

Table 5.3 list several size-related metric values for the TWELVE benchmark. In order to facilitate the analysis of the data, the values are presented along with some context from other benchmarks. From this data, it is obvious that the TWELVE benchmark is tiny. It touches only 117 different bytecode instructions, and does not even execute a million bytecode instructions in its single class (from the value of the base.instructions.value metric[1]). However, looking at the whole program paints a slightly different portrait; in this context TWELVE is much closer in size to LINPACK. These results show that most of TWELVE's computations occur inside the standard libraries. Because of the nature of the benchmark, a reasonable assumption seems to be that it spends the quasi-totality of its time doing string manipulations.

Table 5.4 presents polymorphism-related metrics for the TWELVE benchmark. The next immediately obvious fact about TWELVE is that it has a very high method invocation density. Its polymorphism.invokeDensity.value of 187.100 is the largest that has been recorded for any benchmark except HELLOWORLD, which has an artificially high invoke density due to the fact that it only executes 4 bytecode instructions, one of which is an `invokevirtual`. Its polymorphism.invokeDensity.value value for the whole program is also very high, and ranks TWELVE within the top 10 benchmarks with respect to that

[1]The base metric category contains metrics that failed to meet the basic requirements for the inclusion of dynamic metrics in the set (as discussed in Section 3.1) but which are still useful in some situations.

	Metric	TWELVE	HELLOWORLD	LINPACK	JAVAC
All	polymorphism.invokeDensity.value	52.908	15.132	2.251	39.481
	memory.byteAllocationDensity.value	403.980	1 730.292	150.079	131.824
	memory.objectAllocationDensity.value	11.554	9.084	0.623	3.181
	pointer.refFieldAccessDensity.value	1.067	6.326	0.428	49.778
	pointer.nonrefFieldAccessDensity.value	56.450	47.561	3.266	114.120
	data.arrayDensity.value	20.146	73.196	152.085	37.919
	data.charArrayDensity.value	14.508	32.817	2.123	17.566
All	polymorphism.invokeDensity.value	187.100	250.000	1.325	72.305
	pointer.fieldAccessDensity.value	2.619	250.000	0.001	208.349

Table 5.4: Polymorphism metrics for TWELVE

	Metric	TWELVE	HELLOWORLD	LINPACK	JAVAC
All	memory.byteAllocationDensity.value	359.269	1684.801	149.410	131.804
	memory.objectAllocationDensity.value	10.287	8.937	0.625	3.180
	memory.objectSize.bin(16)	3.9%	14.3%	14.1%	14.6%
	memory.objectSize.bin(24)	89.9%	32.1%	31.1%	41.9%
	memory.objectSize.bin(48-72)	4.3%	23.3%	22.4%	11.4%
Application	memory.byteAllocationDensity.value	0.009	0.000	0.002	69.435
	memory.objectAllocationDensity.value	0.001	0.000	0.000	2.320
	memory.objectSize.bin(16)	0.0%	N/A	100.0%	14.8%
	memory.objectSize.bin(24)	0.0%	N/A	0.0%	22.7%
	memory.objectSize.bin(48-72)	0.0%	N/A	0.0%	6.1%

Table 5.5: Memory metrics for TWELVE

particular metric. TWELVE's recursive nature is an important factor in the measured densities. It is also clear that TWELVE is really not a polymorphic application. In fact, its application class is completely monomorphic (polymorphism.receiverArityCalls.bin(1) = 100%). The whole program version of the same metric reveals only a very low amount of polymorphism; this is most likely due to the effect of the JVM startup.

Table 5.5 shows memory-related metric results. The application part of TWELVE has values of almost zero for both memory.byteAllocationDensity.value and memory.objectAllocationDensity.value In fact, TWELVE itself allocates only one object: an instance

	Metric	TWELVE	HELLOWORLD	LINPACK	JAVAC
All	data.arrayDensity.value	18.012	73.120	151.953	37.932
	data.charArrayDensity.value	12.960	33.169	2.197	17.577
	data.numArrayDensity.value	2.780	34.038	140.602	6.785
	data.refArrayDensity.value	0.131	1.787	8.867	5.608
Application	data.arrayDensity.value	0.000	0.000	157.775	15.471
	data.charArrayDensity.value	0.000	0.000	0.000	3.488
	data.numArrayDensity.value	0.000	0.000	148.385	0.359
	data.refArrayDensity.value	0.000	0.000	9.389	3.543

Table 5.6: Array metrics for TWELVE

of the `Twelve` class. However, including the standard libraries shows that the TWELVE benchmark is relatively memory intensive. It frequently allocates objects—its memory.-objectAllocationDensity.value is the highest of the four benchmarks from Table 5.5, and makes TWELVE rank second of all benchmarks. The memory.byteAllocationDensity.-value value for TWELVE is also high, but clearly not among the high values. This is due to the fact that while TWELVE frequently allocates objects, it allocates fairly small ones. This can be observed by looking at the memory.objectSize.bin(24) metric, which indicates that 24-byte objects account for nearly 90% of all object allocations for this benchmark. A reasonable hypothesis at this time is that `java.lang.String` objects fall under that category; this hypothesis indeed holds under the platform used to conduct this study[2].

This hypothesis is further supported by looking at the density of array operations in the benchmark, shown in Table 5.6. From this data, TWELVE makes a moderate use of character arrays, similar to JAVAC, and ranks among the top 10 benchmarks of the suite for use of character arrays. Table 5.6 also shows that character arrays are by far the most commonly used kind of array in TWELVE ; data.numArrayDensity.value and data.-refArrayDensity.value obtain much lower values of 2.780 and 0.131 respectively.

Finally, from Table 5.7, it can be observed that the locking density in TWELVE is surprisingly high (1.857), and that the locking operations are performed by a relatively small

[2]The `java.lang.String` class has four 4-byte non-static fields (instance variables), combined with a header size of 8 bytes for this particular platform and JVM combination, for a total of 24 bytes.

	Metric	TWELVE	HELLOWORLD	LINPACK	JAVAC
All	concurrency.lockDensity.value	1.857	0.178	0.017	0.229
	concurrency.lock.percentile	25.0%	65.0%	70.0%	6.2%
App	concurrency.lockDensity.value	0.000	0.000	0.000	0.002
	concurrency.lock.percentile	N/A	N/A	N/A	100.0%

Table 5.7: Synchronization metrics for TWELVE

	Metric	TWELVE	TWELVE2	% Change	Ratio
All	base.instructions.value	7 670 314	4 732 160	-38.3%	0.617
	data.charArrayDensity.value	12.960	21.006	62.1%	1.621
	concurrency.lockDensity.value	1.857	3.008	62.0%	1.620
	memory.byteAllocationDensity.value	359.984	243.978	-32.2%	0.678
	memory.objectAllocationDensity.value	11.554	2.884	-75.3%	0.247
	memory.objectSize.bin(16)	3.89	25.472	554.8%	6.548
	memory.objectSize.bin(24)	87.749	32.866	-62.6%	0.375
	memory.objectSize.bin(48-72)	4.463	29.227	554.9%	6.549
App	base.instructions.value	900 508	974 979	8.2%	1.082
	polymorphism.invokeDensity.value	187.100	104.150	-44.3%	0.557

Table 5.8: Change in metrics due to optimization of TWELVE

proportion of the locks (25%).

Improving TWELVE

The most significant findings from the analysis of the benchmark were its very high frequency of method calls, as well as its high memory requirements. In order to address this problem, a new version of the benchmark, TWELVE2, has been written that gets rid of `substring` operations by keeping track of the current position in the array by introducing an additional parameter to the recursive methods. The significant differences in the metrics are shown in Table 5.8.

From Table 5.8, the memory requirements of the TWELVE benchmark have been greatly reduced; its object allocation density has dropped from 11.554 to 2.884, a reduction of 75%. The byte allocation density has also been reduced, from 360 bytes per kbc to 244, which represents a decrease of over 38%. The distribution of objects has also shifted. The

	TWELVE	TWELVE2	% Difference
Real time (msec)	463	433	-6.5%

Table 5.9: Differences in runtime between TWELVE and TWELVE2

distribution of allocated objects in TWELVE2 is more more uniform, as can be observed from the three most important bins from memory.objectSize.bin. Finally, a reduction of over 44% in the density of method invocations in the application code only can also be observed. Thus, the transformation has successfully addressed the original issues.

However, the transformation had inverse effects on two relevant metrics, namely data.-charArrayDensity.value and concurrency.lockDensity.value; both metric values increased by roughly 62%. This apparent change in the metric value was however caused by a change in the number of executed bytecode instructions, and not in the attributes that these metrics measure. Looking at the ratios between the values from the original and transformed versions of TWELVE, one can note that

$$\frac{1}{0.617} \approx 1.621$$

which is incidentally the ratio between the affected metric values. Thus, while relative metrics are more robust than absolute counts with respect to program input, they may vary due to changes in the executed bytecode instructions which are irrelevant to the attributes that they measure. Such variations can be misleading; on the other hand, they only occur when comparing different versions of a benchmark.

Table 5.9 shows the execution time for both the TWELVE and TWELVE2 benchmarks. The measurements correspond to the average execution time for five runs. The optimization had a positive effect on the total running time of the benchmark, reducing the original time by approximately 6.5%.

Improving TWELVE2

Both TWELVE and TWELVE2 had a surprisingly high density of lock operations. Further investigation can reveal that the locking is caused by calling I/O routines from the standard libraries (see Section 3.4.5 for a previous discussion of the matter). A new version of the benchmark, TWELVE3, has been designed which improves upon TWELVE2 by reducing the

	Metric	TWELVE2	TWELVE3	% Change	Ratio
All	base.instructions.value	4 732 160	2 845 436	-39.9%	0.601
	data.charArrayDensity.value	21.006	34.920	66.2%	1.662
	concurrency.lockDensity.value	3.008	0.311	-89.7%	0.103

Table 5.10: Change in metrics due to optimization of TWELVE2

	TWELVE2	TWELVE3	% Difference
Real time (msec)	433	279	-35.6%

Table 5.11: Differences in runtime between TWELVE2 and TWELVE3

number of calls to `System.out.print`. This is accomplished by collecting the output in a `java.lang.StringBuffer` object. The `StringBuffer` class is itself synchronized to support concurrent modifications; however, every call to `System.out.print` is very heavyweight and touches six distinct `monitorenter` instructions, rather than a single one in the case of `StringBuffer`. Therefore, buffering the output is still expected to have a beneficial effect overall. Table 5.10 shows the changes in the metric values which were caused by this optimization.

From Table 5.10, the optimization led to the desired reduction of the lock density—the value of concurrency.lockDensity.value dropped by almost 90% to reach the very low value of 0.311 lock operations per kbc. The total number of executed bytecodes was also reduced in the process, indicating calls to the I/O library were also expensive because of their deep call chains. As with TWELVE2, various densities were again skewed because of the dramatic reduction in the number of executed bytecodes. Table 5.10 shows the effect on the value of the data.charArrayDensity.value metric, which exhibits a variation that is inversely proportional to the reduction in the number of executed bytecodes.

Table 5.11 shows the execution time for both the TWELVE2 and TWELVE3 benchmarks. The improvement is even more noticeable for this optimization, since it reduced the running time of the already optimized TWELVE2 program by another 35.6%. This represents a total improvement of 39.7% over the unoptimized TWELVE.

	Metric	Orig.	Inline	-O	PT+CSE
All	base.instructions.value	445.13 M	287.86 M	280.41 M	282.08 M
Application	size.run.value	1008	1449	1417	1425
	size.hot.value	330	683	662	663
	polymorphism.invokeDensity.value	116.17	10.84	11.13	11.06
	polymorphism.receiverArityCalls..bin(1)	100%	100%	100%	100%
	polymorphism.targetArity.bin(1)	100%	100%	100%	100%
	pointer.fieldAccessDensity.value	126.7	196.1	200.9	177.8

Table 5.12: Dynamic Metrics for the VORONOI Benchmark

5.2.3 Compiler Optimizations

The previous subsection has demonstrated that dynamic metrics can be used to guide manual program transformations or refactorings, as well as to quantify their respective effects on the behaviour of the program. However, the same approach can be applied to compiler optimizations, where dynamic metrics can be used to reveal potential optimization opportunities, and evaluate the effect of applying the transformations. In order to demonstrate this, the effect of compiler transformations will be studied using the VORONOI benchmark.

Table 5.12 shows the relevant dynamic metrics for four variations of the VORONOI benchmark. Note that because only the application code can be optimized, all metrics except base.instructions.value are reported in their application-only version. A detailed look at each variation will be provided next, followed by a discussion of the impact of the transformations on the running time of the benchmark.

Effect of Transformations on Dynamic Metrics

In Table 5.12, the column named *Orig.* corresponds to the metric values for the original version of the VORONOI benchmark[3]. While the benchmark executes about 445 million bytecode instructions (base.instructions.value), it only touched 1008 distinct bytecodes in its own classes (size.run.value). The size of the VORONOI application is thus relatively small. Of these 1008 bytecodes, only 330 of them account for 90% of the total bytecode executions (size.hot.value).

The most interesting metrics for the VORONOI benchmark have to do with the density

[3]For consistency, all four variations of the benchmark have been run through SOOT, a Java bytecode transformation framework.

and polymorphism of the virtual method calls. The benchmark has a very high density of calls to to virtual methods, as the polymorphism.invokeDensity.value metric is 116.17. This means that this benchmark executes a virtual invocation about 1 out of every 11 bytecode instructions. Moreover, the fact that the both the polymorphism.receiverArityCalls.-bin(1) and the polymorphism.targetArityCalls.bin(1) metrics have a value of 100% indicates that all virtual calls originate from completely monomorphic call sites. The high density of virtual invocations indicates that method inlining is likely to have a positive impact of the performance of the application; the low polymorphism suggests that compiler techniques could likely resolve each call site to exactly one method, and are this likely to be very effective at performing inlining.

The column labelled *Inline* gives the dynamic metrics for the same benchmark after having applied inlining using the SOOT framework. Note that this had a dramatic effect on the benchmark as the transformed version executes about 288 million instructions, a reduction of approximately 35%. The transformation also had the desired effect on the invocation density, which was reduced from 116.17 to 10.84. However, inlining also introduced 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 having additionally 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, another opportunity for optimization is apparent from the metric values: the density of field accesses is very high in the transformed version of the program. The pointer.fieldAccessDensity.value metric has a value of 200.9, which means that about 1 in 5 bytecode instructions is an access (either a write or a read operation) to a field.

In the case where a single field is read multiple times, such as in the following code:

```
...
a = p.x;
```

```
...
b = p.x;
...
```

it is possible to transform the program so that the value of the field will be stored in a local variable, and then reuse the value of the variable to avoid performing repeated, more expensive field accesses. The transformed example would thus become the following:

```
...
temp = p.x;
a = temp;
...
b = temp;
...
```

Note that this is only possible when the value of the field does not change between the two accesses. Determining this information is further complicated by the fact that multiple references can be *aliased*, i.e., point to the same object. The following example illustrates such a case where the transformation would be prohibited:

```
...
p.x = 0;
...
q = p; // Alias p and q
a = p.x; // a = 0
...
q.x = 4; // p.x = 4
...
b = p.x; // b = 4, not b = 0
...
```

In order to reduce the number of field accesses, a whole-program points-to (PT) analysis has been performed in order to determine which references are possibly aliased, followed by a common-sub-expression (CSE) elimination of field accesses. 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. The metric values for this transformation are provided in the column labelled *PT+CSE*.

Mode		Orig.	Inline	-O	PT+CSE
Interpreter	runtime	48.43	30.50	30.08	30.10
JIT–No Inlining	runtime	12.23	9.87	9.89	9.88
	compile-time	0.048	0.046	0.045	0.044
	compiled-bytes	2711	3683	3519	3400
JIT	runtime	9.80	9.72	9.76	9.75
	compile-time	0.094	0.052	0.055	0.046
	compiled-bytes	2743	3435	3193	3150

Table 5.13: Running time measurements for the VORONOI benchmark

Indeed, the metrics show that after applying the transformation the number of executed instructions increased from 280M to 282M and the size of the application went up from 1417 instructions to 1425 instructions. However, it did have the desired effect on the field accesses; the value of the data.fieldAcessDensity.value metric been reduced from 200.9 to 177.8.

Effect of Optimizations on Runtime Performance

As it has been demonstrated in the previous subsection, the dynamic metrics help to identify opportunities for optimizations and transformations, and can also help in understanding the effect, both positive and negative.

Table 5.13 presents 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 JVM. In order to get reliable and repeatable results, a slightly larger problem size was used for the runtime experiments than when collecting the metrics (metric computations were performed using 20 000 nodes, while runtime experiements used 100 000 nodes). The runtime numbers are the average of five runs, reporting the total time as reported 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.

In Table 5.13, runtime measurements are given for three configurations of the VM. The first configuration, labelled *Interpreter*, runs only in interpretive mode (`java -Xint`). The second configuration, labelled *JIT–No Inlining*, 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 runs in 48.43 seconds, whereas the *Inline* version is much faster, executing in 30.50 seconds. The *-O* version shows a small improvement, with an execution time of 30.08 seconds, which corresponds quite well with the small improvement in executed instructions that was visible in the metrics. The *PTR+CSE* shows a very slight increase in running time, executing in 30.10 seconds. This shows that the additional instructions that were introduced by the transformation could not all be eliminated.

The result from the *JIT–No Inlining* configuration again show that the statically-inlined version of VORONOI (column *Inline*) executes much faster, 9.87 versus 12.23 seconds for the original version. 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, it is worth noting that the amount of compiled code does go down slightly.

Running the benchmark with the ordinary *JIT* configuration shows that the JIT inliner is quite effective, giving an execution time of 9.80 versus 12.23 when the JIT inliner is turned off. This indicates that two different inliners (SOOT's 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 has to compile slightly more code (2743 bytes vs 2711 bytes for when the inliner is disabled). 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 (9.72 seconds), the JIT compile time is very reasonable (0.051 seconds), and the amount of compiled code is quite small (3435 bytes).

Chapter 6
Conclusions and Future Work

6.1 Conclusions

This thesis has presented dynamic metrics as a means of assessing the actual runtime behaviour of a program by providing a high-level overview of several of its key aspects. This dynamic information can be more relevant than the more common static measures, in particular for compiler and runtime system developers.

Five categories of dynamic metrics have been defined and empirically validated using data from a representative set of benchmarks programs. The metrics characterize a program's runtime behaviour in terms of size, data structures, polymorphism, memory use, and concurrency and synchronization. These metrics were designed with the goals of being unambiguous, dynamic, robust, discriminating, and platform-independent.

A flexible and extensible dynamic analysis framework, **J*, has been built around the JVMPI, and allows the defined metrics to be computed for Java programs, as well as to distill a concise subset that characterizes a program.

Empirical validation of the metrics was performed by computing the defined metrics for a representative set of well-known Java benchmarks, and by evaluating the metrics in terms of how well they corresponded to the commonly accepted qualitative appraisal of the benchmarks. The discussion also considered how well each of the metrics satisfied the set of desirable metric properties.

The utility of the metrics was evaluated by applying them to three specific problems, and determining to which extent they provided useful information for each task.

6.2 Future Work

The work on dynamic metrics can be extended in several ways. This section presents some possible directions for future research.

6.2.1 Alternate Data Sources

In order to continue the investigation of dynamic metrics, it would be necessary to find alternate sources of execution data in order to overcome the limitations that are imposed by using the JVMPI. In particular, the addition of a general-purpose profiling framework within SableVM [Gag02], a free, open-source and portable Java virtual machine, would offer many potential advantages. Such a framework could provide a more versatile profiling interface than what the JVMPI currently offers, and allow for a greater flexibility in terms of the information that can be recorded. Such a profiling framework may also be useful to other researchers who also often need to instrument a Java virtual machine in order to obtain dynamic information.

6.2.2 Additional Metrics

Several new metrics could be added to our set. In particular, the following metrics are believed to be among the most promising for compiler optimization and runtime development.

Program Structure

Based on the ideas that were proposed as part of the research done on static complexity metrics, the decision structure of a program seems to be closely associated with its complexity. Dynamic metrics based on measuring various aspects of the control flow of an application could therefore provide an informative measure of one aspect of its complexity. Such metrics could be obtained by measuring instructions that change control flow (`if`, `switch`, `goto`, method invocations, . . .). For example, a program with a single large loop is considered simple, whereas a program with multiple loops and/or many control flow changes within a loop is considered more complex. Simple metrics include measuring the density of execution of control bytecodes. A more dynamic measure of the decision structure would involve, for instance, measuring the rate at which control bytecodes change direction. Percentile metrics measuring the "hotness" of control bytecodes, as well as bin metrics that measure the proportion of such bytecodes that always take a particular direction, or that take different ones at different points in the execution, are also possible.

Concurrency

Identifying concurrent benchmarks involves determining whether more than one thread[1] 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 amount of concurrency in a program could, however, be approximated by monitoring the threads which are simultaneously in the *active* or *runnable* states, i.e., threads that are either running, or at least capable of being run (but which are not currently scheduled). In this case, the requirement on the number of processors that are needed to show concurrent execution is lifted.

[1]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.

Unfortunately, this would 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), this amount of extra imprecision should not overly obscure the actual concurrency of an application.

Memory

Another interesting program behaviour is to determine if there exists an allocation hotspot for a given program execution, i.e., small fraction of allocation sites which account for a large fraction of total bytes or objects allocated. Value and percentile metrics would be good candidates for measuring such behaviour.

Researchers interested in garbage collection are often interested in the liveness of dynamically-allocated objects. For example, generational garbage collection is potentially a good idea if a large proportion of objects have short lifetimes. For liveness metrics, time is often reported in terms of intervals of allocated bytes. For example, for an interval size of 10000 bytes, interval 1 ends after 10000 bytes have been allocated, interval 2 ends after 20000 bytes have been allocated and so on.

Object lifetimes could be estimated by identifying objects which become unreachable during each interval. Based on the the amount of live memory at the end of each interval, a simple metric such as the average or maximum number of live bytes or object could be computed. More interestingly, more dynamic metrics such as the rate at which objects or bytes become unreachable, or a bin metric reporting the proportion of objects or allocated bytes which survive predetermined numbers of intervals (e.g., 0, 1, 2, 3–5, 6–10, 11 and more) could also be reported.

By defining the *birth time* of an object as the number of the interval in which it was allocated and conversely its *death time* as the number of the interval in which it becomes unreachable, it is possible to define metrics which measure various aspects of the *lifetime* of objects. Bin metrics would constitute good candidates for such measurements.

Another concept that is appearing in the garbage collection literature is the notion of dragged objects [RR96, SKS02]. Dragged objects are those that are still reachable, but are never touched for the remainder of the execution. These objects therefore represent unnecessary heap overhead. Dynamic metrics could measure quantities such as the average drag time of an object, or the proportion of the heap which is occupied by dragged objects.

Pointers

Pointer polymorphism is typically measured as an average or maximum number of target addresses per pointer, and symmetrically the number of pointers per target address (Cheng and Hwu argue that both are required for a more accurate measurement [CH00]). Value and bin metrics could be defined to measure the number of distinct objects that are referenced by each object reference, as well as the number of distinct object references directed at each object.

Continuous Extensions

Because of the large amount of information that is now available in software visualization tools, an approach to program understanding based on a refinement process has obvious advantages. In order to bridge the large gap that exists between dynamic software metrics and complete visualization, continuous versions of the metrics need to be defined. Such metrics would be computed at specific intervals rather than once at the end of the execution, and would thus show the evolution of the metric value over time. One particularly interesting potential application of such a technique is metric-based program phase detection.

6.2.3 Objectivity of Measures

Most metrics are affected by variations in the program's behaviour which should not impact the characteristics being measured. Different kinds of metrics react differently to such changes in the behaviour. While it may not be possible to completely eliminate such noise, any reduction of its effects on the metrics would have a positive impact on the objectivity of the measures.

The informally-defined desirable qualities of dynamic metrics that were presented in Section 3.1 could also benefit from a formal definition. This would allow further studies to take place, such as investigating a possible correlation between the informative nature of a metric and its robustness with respect to unrelated behavioural changes.

6.2.4 Efficiency of the Computations

Due to the very large amount of information being processed, the current method for computing dynamic metrics is very slow — in the range of several hours for simple benchmark programs. There are many aspects of the computation that could be optimized for speed. In particular, better compression techniques, instruction predicting strategies, alternate profiling techniques and data representations could result in significant speed improvements. The integration of STEP and **J* is also planned. Investigating sample-based techniques, as well as their impact of the measurement values would also be valuable. Most of the research in this area would also potentially applicable to the field of software visualization.

Appendix A
Metric Data

A.1 Whole program metrics

A.1. Whole program metrics

Table A.1: Whole program metrics for small benchmarks

Metric	EMPTY	HELLOWORLD	AUTOMATA	COEFFICIENTS	LINPACK
base.bytes.value	940008	941048	1119784	3070920	1281376
base.classes.value	277	277	275	286	280
base.instructions.value	558128	561842	2551019	28264280	8613320
base.methods.value	16202	16391	161835	1818630	27738
base.objects.value	4906	4922	10933	21216	5287
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000
concurrency.lock.percentile	40.9%	38.9%	15.1%	16.7%	47.3%
concurrency.lockDensity.value	1.528	1.557	6.677	0.352	0.111
data.arrayDensity.value	73.311	73.120	31.059	150.880	151.953
data.charArrayDensity.value	33.186	33.169	10.430	1.562	2.197
data.floatDensity.value	1.987	1.974	0.438	202.808	285.427
data.numArrayDensity.value	34.213	34.038	14.581	74.986	140.602
data.refArrayDensity.value	1.795	1.787	0.396	73.708	8.867
memory.averageObjectSize.value	191.60375	191.1922	102.42239	144.74548	242.36353
memory.byteAllocationDensity.value	1684.216	1674.934	438.956	108.650	148.767
memory.objectAllocationDensity.value	8.790	8.760	4.286	0.751	0.614
memory.objectSize.bin(8)	0.6%	0.6%	0.3%	0.1%	0.5%
memory.objectSize.bin(16)	14.7%	14.6%	24.9%	14.4%	14.4%
memory.objectSize.bin(24)	32.8%	32.8%	33.1%	38.8%	31.7%
memory.objectSize.bin(32)	8.7%	8.7%	3.9%	3.7%	8.3%
memory.objectSize.bin(40)	9.5%	9.4%	4.2%	4.1%	8.9%
memory.objectSize.bin(48-72)	21.7%	21.8%	28.3%	15.6%	20.9%
memory.objectSize.bin(80-136)	7.2%	7.3%	3.3%	8.1%	6.9%
memory.objectSize.bin(144-392)	3.9%	3.8%	1.6%	7.6%	3.6%
memory.objectSize.bin(400+)	0.9%	1.0%	0.4%	7.6%	4.8%
pointer.fieldAccessDensity.value	53.844	54.179	148.044	97.686	3.799
pointer.nonrefFieldAccessDensity.value	30.054	30.268	92.139	34.807	2.120
pointer.refFieldAccessDensity.value	23.790	23.911	55.905	62.880	1.679

Table A.1: Whole program metrics for small benchmarks (continued)

Metric	Empty	HelloWorld	Automata	Coefficients	Linpack
polymorphism.callSites.value	437	472	482	677	530
polymorphism.calls.value	8356	8485	111511	1731915	19559
polymorphism.invokeDensity.value	14.971	15.102	43.712	61.276	2.271
polymorphism.receiverArity.bin(1)	97.3%	97.5%	97.5%	98.1%	97.7%
polymorphism.receiverArity.bin(2)	2.5%	2.3%	2.3%	1.8%	2.1%
polymorphism.receiverArity.bin(3+)	0.2%	0.2%	0.2%	0.1%	0.2%
polymorphism.receiverArityCalls.bin(1)	95.2%	95.2%	96.0%	100.0%	97.9%
polymorphism.receiverArityCalls.bin(2)	2.1%	2.1%	3.8%	0.0%	1.0%
polymorphism.receiverArityCalls.bin(3+)	2.6%	2.6%	0.2%	0.0%	1.1%
polymorphism.receiverCacheMissRate.value	8.4%	8.7%	0.7%	0.1%	4.1%
polymorphism.receiverPolyDensity.value	0.02746	0.02542	0.0249	0.0192	0.02264
polymorphism.receiverPolyDensityCalls.value	0.04751	0.04773	0.03994	0.00045	0.02147
polymorphism.targetArity.bin(1)	98.4%	98.5%	98.5%	98.8%	98.7%
polymorphism.targetArity.bin(2)	1.4%	1.3%	1.2%	1.0%	1.1%
polymorphism.targetArity.bin(3+)	0.2%	0.2%	0.2%	0.1%	0.2%
polymorphism.targetArityCalls.bin(1)	96.7%	96.8%	99.8%	100.0%	98.6%
polymorphism.targetArityCalls.bin(2)	0.6%	0.6%	0.0%	0.0%	0.3%
polymorphism.targetArityCalls.bin(3+)	2.6%	2.6%	0.2%	0.0%	1.1%
polymorphism.targetCacheMissRate.value	6.9%	7.2%	0.6%	0.0%	3.4%
polymorphism.targetPolyDensity.value	0.01602	0.01483	0.01452	0.01182	0.01321
polymorphism.targetPolyDensityCalls.value	0.03267	0.03241	0.00249	0.00021	0.01411
size.codeCoverage.value	10.2%	10.8%	11.4%	16.1%	13.7%
size.deadCode.value	64705	64258	63926	67405	67465
size.hot.value	985	1009	924	116	119
size.hot.percentile	13.4%	12.9%	11.2%	0.9%	1.1%
size.hotClasses.value	10	10	15	3	1
size.hotClasses.percentile	8.1%	7.6%	11.4%	1.9%	0.7%
size.hotMethods.value	34	35	53	6	4
size.hotMethods.percentile	7.7%	7.4%	11.0%	1.0%	0.8%
size.load.value	72059	72062	72158	80292	78173
size.loadedClasses.value	277	277	275	286	280
size.loadedMethods.value	3603	3603	3599	3735	3650
size.run.value	7354	7804	8232	12887	10708

Table A.2: Whole program metrics for small mulithreaded benchmarks

Metric	ROLLERCOASTER	TELECOM	VOLANO-CLIENT	VOLANO-SERVER
base.bytes.value	1047536	25403664	2253960	3388888
base.classes.value	282	322	396	547
base.instructions.value	2029566	219682923	9202666	14772826
base.methods.value	133495	12605381	387753	608208
base.objects.value	7418	358398	33013	50143
concurrency.contendedLock.percentile	50.0%	30.0%	28.6%	50.0%
concurrency.contendedLockDensity.value	0.271	0.088	0.017	0.017
concurrency.lock.percentile	14.1%	20.6%	10.8%	11.7%
concurrency.lockDensity.value	18.142	4.259	8.730	5.593
data.arrayDensity.value	43.999	53.058	60.954	63.117
data.charArrayDensity.value	16.331	10.783	21.765	24.963
data.floatDensity.value	0.568	0.015	0.278	0.409
data.numArrayDensity.value	12.625	5.066	22.318	21.098
data.refArrayDensity.value	11.489	34.848	0.561	2.475
memory.averageObjectSize.value	141.21542	70.88115	68.27492	67.58447
memory.byteAllocationDensity.value	516.138	115.638	244.925	229.400
memory.objectAllocationDensity.value	3.655	1.631	3.587	3.394
memory.objectSize.bin(8)	0.4%	0.0%	0.2%	0.2%
memory.objectSize.bin(16)	10.3%	4.7%	13.6%	13.5%
memory.objectSize.bin(24)	34.6%	47.4%	45.9%	44.5%
memory.objectSize.bin(32)	6.2%	8.7%	10.2%	10.5%
memory.objectSize.bin(40)	11.9%	8.6%	7.1%	6.7%
memory.objectSize.bin(48-72)	25.3%	21.7%	11.5%	14.9%
memory.objectSize.bin(80-136)	8.0%	4.4%	7.2%	6.6%
memory.objectSize.bin(144-392)	2.6%	4.3%	3.6%	2.2%
memory.objectSize.bin(400+)	0.7%	0.3%	0.7%	0.9%
pointer.fieldAccessDensity.value	109.555	111.066	108.634	103.315
pointer.nonrefFieldAccessDensity.value	64.621	61.084	63.174	59.002
pointer.refFieldAccessDensity.value	44.934	49.982	45.459	44.313

Table A.2: Whole program metrics for small mulithreaded benchmarks (continued)

Metric	RollerCoaster	Telecom	Volano-Client	Volano-Server
polymorphism.callSites.value	536	967	1577	2588
polymorphism.calls.value	87421	10913528	300399	484130
polymorphism.invokeDensity.value	43.074	49.679	32.643	32.772
polymorphism.receiverArity.bin(1)	97.4%	95.6%	98.2%	97.0%
polymorphism.receiverArity.bin(2)	2.1%	3.5%	1.4%	2.0%
polymorphism.receiverArity.bin(3+)	0.6%	0.9%	0.4%	1.0%
polymorphism.receiverArityCalls.bin(1)	98.2%	98.2%	97.3%	94.6%
polymorphism.receiverArityCalls.bin(2)	1.5%	1.3%	1.7%	2.5%
polymorphism.receiverArityCalls.bin(3+)	0.3%	0.6%	1.0%	2.9%
polymorphism.receiverCacheMissRate.value	1.0%	0.8%	1.1%	2.8%
polymorphism.receiverPolyDensity.value	0.02612	0.04447	0.01839	0.02975
polymorphism.receiverPolyDensityCalls.value	0.01764	0.0182	0.02705	0.05435
polymorphism.targetArity.bin(1)	98.7%	97.6%	98.7%	98.0%
polymorphism.targetArity.bin(2)	1.1%	1.7%	1.0%	1.5%
polymorphism.targetArity.bin(3+)	0.2%	0.7%	0.3%	0.5%
polymorphism.targetArityCalls.bin(1)	99.6%	99.3%	98.1%	96.4%
polymorphism.targetArityCalls.bin(2)	0.1%	0.1%	1.4%	1.8%
polymorphism.targetArityCalls.bin(3+)	0.3%	0.6%	0.5%	1.8%
polymorphism.targetCacheMissRate.value	0.8%	0.6%	0.8%	1.4%
polymorphism.targetPolyDensity.value	0.01306	0.02378	0.01268	0.02009
polymorphism.targetPolyDensityCalls.value	0.00363	0.00704	0.0192	0.03645
size.codeCoverage.value	11.8%	24.4%	22.2%	34.7%
size.deadCode.value	63923	69155	68853	97886
size.hot.value	1219	713	1273	2418
size.hot.percentile	14.3%	3.2%	6.5%	4.6%
size.hotClasses.value	20	10	13	23
size.hotClasses.percentile	14.6%	4.9%	5.5%	6.6%
size.hotMethods.value	59	32	56	96
size.hotMethods.percentile	11.8%	3.5%	5.2%	5.5%
size.load.value	72460	91469	88494	149940
size.loadedClasses.value	282	322	396	547
size.loadedMethods.value	3620	4095	4526	6132
size.run.value	8537	22314	19641	52054

Table A.3: Whole program metrics for medium and large benchmarks

Metric	JLEX	SOOT (JIMPLE)	SOOT (COFFI)	SABLECC (JAVA)	SABLECC (WIG)
base.bytes.value	22126432	616113752	235391416	217111616	238632360
base.classes.value	310	819	809	663	666
base.instructions.value	72940156	2124320307	1401704070	632803039	1155005238
base.methods.value	6112124	152128245	63999476	44361168	68825232
base.objects.value	441908	17493261	5668203	3503081	4243762
concurrency.contendedLock.percentile	N/A	100.0%	100.0%	33.3%	66.7%
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000
concurrency.lock.percentile	4.8%	10.0%	7.5%	5.6%	4.6%
concurrency.lockDensity.value	23.313	5.769	4.251	12.764	9.825
data.arrayDensity.value	24.583	42.821	84.290	43.635	52.068
data.charArrayDensity.value	1.254	7.316	4.168	6.638	4.885
data.floatDensity.value	0.021	1.109	0.066	0.032	0.019
data.numArrayDensity.value	9.522	10.915	52.585	9.450	18.125
data.refArrayDensity.value	12.459	13.612	11.032	15.613	13.917
memory.averageObjectSize.value	50.07022	35.22006	41.5284	61.97733	56.23132
memory.byteAllocationDensity.value	303.350	290.029	167.932	343.095	206.607
memory.objectAllocationDensity.value	6.059	8.235	4.044	5.536	3.674
memory.objectSize.bin(8)	0.0%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	0.6%	46.3%	46.5%	22.3%	26.0%
memory.objectSize.bin(24)	2.6%	33.9%	33.7%	40.8%	37.5%
memory.objectSize.bin(32)	93.9%	1.0%	1.0%	7.2%	10.0%
memory.objectSize.bin(40)	0.5%	6.5%	5.5%	12.8%	11.8%
memory.objectSize.bin(48-72)	1.9%	10.6%	10.4%	6.5%	5.7%
memory.objectSize.bin(80-136)	0.3%	1.0%	1.2%	5.3%	4.5%
memory.objectSize.bin(144-392)	0.2%	0.5%	0.9%	5.0%	4.3%
memory.objectSize.bin(400+)	0.0%	0.2%	0.9%	0.2%	0.2%
pointer.fieldAccessDensity.value	167.756	159.668	167.253	157.043	147.967
pointer.nonrefFieldAccessDensity.value	104.479	67.313	53.710	57.370	45.607
pointer.refFieldAccessDensity.value	63.278	92.355	113.543	99.673	102.360

Table A.3: Whole program metrics for medium and large benchmarks (continued)

Metric	JLEX	SOOT (JIMPLE)	SOOT (COFFI)	SABLECC (JAVA)	SABLECC (WIG)
polymorphism.callSites.value	1263	3802	3453	3984	3876
polymorphism.calls.value	3510504	109808788	47599767	32127993	51805162
polymorphism.invokeDensity.value	48.129	51.691	33.958	50.771	44.853
polymorphism.receiverArity.bin(1)	98.6%	93.6%	93.4%	94.3%	93.9%
polymorphism.receiverArity.bin(2)	1.0%	3.0%	3.2%	2.4%	2.7%
polymorphism.receiverArity.bin(3+)	0.4%	3.5%	3.4%	3.3%	3.4%
polymorphism.receiverArityCalls.bin(1)	90.5%	68.3%	63.4%	94.3%	94.7%
polymorphism.receiverArityCalls.bin(2)	0.1%	13.2%	12.7%	0.5%	0.5%
polymorphism.receiverArityCalls.bin(3+)	9.5%	18.5%	23.9%	5.2%	4.8%
polymorphism.receiverCacheMissRate.value	0.2%	8.9%	7.6%	0.2%	0.2%
polymorphism.receiverPolyDensity.value	0.01425	0.06444	0.06632	0.05748	0.0614
polymorphism.receiverPolyDensityCalls.value	0.09532	0.31709	0.36567	0.05733	0.05318
polymorphism.targetArity.bin(1)	99.1%	95.2%	95.2%	97.3%	97.3%
polymorphism.targetArity.bin(2)	0.6%	2.1%	2.3%	2.0%	2.1%
polymorphism.targetArity.bin(3+)	0.3%	2.6%	2.5%	0.6%	0.6%
polymorphism.targetArityCalls.bin(1)	90.5%	72.8%	74.6%	95.5%	95.9%
polymorphism.targetArityCalls.bin(2)	9.3%	11.9%	11.0%	0.2%	0.2%
polymorphism.targetArityCalls.bin(3+)	0.2%	15.3%	14.3%	4.2%	3.9%
polymorphism.targetCacheMissRate.value	0.1%	4.3%	3.6%	0.1%	0.2%
polymorphism.targetPolyDensity.value	0.0095	0.04761	0.04778	0.02661	0.02657
polymorphism.targetPolyDensityCalls.value	0.09488	0.27167	0.25351	0.04454	0.04096
size.codeCoverage.value	24.4%	30.3%	28.3%	33.9%	33.1%
size.deadCode.value	66072	88497	90901	96726	97914
size.hot.value	422	3323	1859	1760	1434
size.hot.percentile	2.0%	8.7%	5.2%	3.6%	3.0%
size.hotClasses.value	8	35	23	19	17
size.hotClasses.percentile	4.5%	6.5%	4.6%	4.0%	3.5%
size.hotMethods.value	25	134	86	78	62
size.hotMethods.percentile	3.3%	7.5%	5.0%	3.7%	2.9%
size.load.value	87405	126901	126733	146253	146337
size.loadedClasses.value	310	819	809	663	666
size.loadedMethods.value	3848	6013	5983	7162	7173
size.run.value	21333	38404	35832	49527	48423

Table A.4: Whole program metrics for JOlden benchmarks–part 1

Metric	BARNES-HUT	BISORT	EM3D	HEALTH	MST
base.bytes.value	703254112	4763424	9646152	38481896	50504088
base.classes.value	285	277	280	283	281
base.instructions.value	1936887268	534219368	267434575	385920250	203111553
base.methods.value	44963711	15550676	2099777	23415523	17297868
base.objects.value	15521638	136396	21749	1232200	2105058
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	100.0%
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000
concurrency.lock.percentile	51.7%	48.3%	1.6%	49.2%	51.7%
concurrency.lockDensity.value	0.001	0.002	4.521	0.004	0.006
data.arrayDensity.value	105.891	0.081	89.253	4.564	19.585
data.charArrayDensity.value	0.013	0.037	0.080	0.065	0.112
data.floatDensity.value	245.665	0.002	13.511	5.988	0.006
data.numArrayDensity.value	97.512	0.037	3.067	0.056	0.102
data.refArrayDensity.value	4.380	0.002	81.534	3.992	19.352
memory.averageObjectSize.value	45.30798	34.92349	443.52163	31.23024	23.99178
memory.byteAllocationDensity.value	363.085	8.917	36.069	99.715	248.652
memory.objectAllocationDensity.value	8.014	0.255	0.081	3.193	10.364
memory.objectSize.bin(8)	0.0%	0.0%	0.1%	0.0%	0.0%
memory.objectSize.bin(16)	49.6%	0.5%	3.6%	69.4%	49.9%
memory.objectSize.bin(24)	0.0%	97.3%	8.1%	16.0%	49.9%
memory.objectSize.bin(32)	0.4%	0.3%	2.1%	14.0%	0.0%
memory.objectSize.bin(40)	49.6%	0.3%	20.6%	0.1%	0.0%
memory.objectSize.bin(48-72)	0.3%	0.9%	6.1%	0.2%	0.1%
memory.objectSize.bin(80-136)	0.0%	0.3%	2.7%	0.1%	0.0%
memory.objectSize.bin(144-392)	0.0%	0.2%	7.1%	0.2%	0.0%
memory.objectSize.bin(400+)	0.0%	0.1%	49.7%	0.1%	0.1%
pointer.fieldAccessDensity.value	111.141	120.798	110.669	249.321	80.125
pointer.nonrefFieldAccessDensity.value	5.781	47.485	21.167	79.331	22.096
pointer.refFieldAccessDensity.value	105.359	73.313	89.503	169.990	58.029

Table A.4: Whole program metrics for JOlden benchmarks–part 1 (continued)

Metric	BARNES-HUT	BISORT	EM3D	HEALTH	MST
polymorphism.callSites.value	667	546	577	616	598
polymorphism.calls.value	36149267	6159976	2070608	22017320	9426469
polymorphism.invokeDensity.value	18.664	11.531	7.742	57.051	46.410
polymorphism.receiverArity.bin(1)	97.6%	97.8%	97.9%	98.1%	98.0%
polymorphism.receiverArity.bin(2)	2.2%	2.0%	1.9%	1.8%	1.8%
polymorphism.receiverArity.bin(3+)	0.1%	0.2%	0.2%	0.2%	0.2%
polymorphism.receiverArityCalls.bin(1)	86.7%	100.0%	100.0%	100.0%	100.0%
polymorphism.receiverArityCalls.bin(2)	13.3%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverCacheMissRate.value	3.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverPolyDensity.value	0.02399	0.02198	0.0208	0.01948	0.02007
polymorphism.receiverPolyDensityCalls.value	0.13257	8e-05	0.00026	3e-05	6e-05
polymorphism.targetArity.bin(1)	98.4%	98.7%	98.8%	98.9%	98.8%
polymorphism.targetArity.bin(2)	1.5%	1.1%	1.0%	1.0%	1.0%
polymorphism.targetArity.bin(3+)	0.1%	0.2%	0.2%	0.2%	0.2%
polymorphism.targetArityCalls.bin(1)	86.7%	100.0%	100.0%	100.0%	100.0%
polymorphism.targetArityCalls.bin(2)	13.3%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	3.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetPolyDensity.value	0.01649	0.01282	0.01213	0.01136	0.01171
polymorphism.targetPolyDensityCalls.value	0.13256	5e-05	0.00015	2e-05	4e-05
size.codeCoverage.value	15.1%	13.2%	13.2%	14.9%	14.0%
size.deadCode.value	65866	66118	66389	65131	65592
size.hot.value	160	138	76	66	186
size.hot.percentile	1.4%	1.4%	0.8%	0.6%	1.7%
size.hotClasses.value	3	1	2	3	4
size.hotClasses.percentile	1.9%	0.7%	1.4%	2.0%	2.6%
size.hotMethods.value	7	4	3	5	11
size.hotMethods.percentile	1.1%	0.8%	0.6%	0.8%	1.9%
size.load.value	77594	76129	76518	76566	76295
size.loadedClasses.value	285	277	280	283	281
size.loadedMethods.value	3683	3623	3642	3637	3644
size.run.value	11728	10011	10129	11435	10703

Table A.5: Whole program metrics for JOlden benchmarks–part 2

Metric	PERIMETER	POWER	TSP	VORONOI
base.bytes.value	18251376	49747904	2792344	48128808
base.classes.value	285	281	278	281
base.instructions.value	169521668	1290322805	53740828	467194279
base.methods.value	12219921	24030991	1583529	54588717
base.objects.value	459021	789299	54250	1441905
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	48.3%	51.7%	3.2%	51.7%
concurrency.lockDensity.value	0.009	0.001	1.849	0.003
data.arrayDensity.value	0.308	93.417	0.793	28.351
data.charArrayDensity.value	0.156	0.017	0.364	0.049
data.floatDensity.value	0.008	474.910	471.566	226.352
data.numArrayDensity.value	0.128	92.490	0.364	0.044
data.refArrayDensity.value	0.007	0.156	0.019	27.675
memory.averageObjectSize.value	39.76153	63.02796	51.47178	33.37863
memory.byteAllocationDensity.value	107.664	38.555	51.959	103.017
memory.objectAllocationDensity.value	2.708	0.612	1.009	3.086
memory.objectSize.bin(8)	0.0%	0.0%	0.1%	0.0%
memory.objectSize.bin(16)	0.2%	0.1%	1.4%	3.5%
memory.objectSize.bin(24)	0.4%	1.6%	3.1%	75.0%
memory.objectSize.bin(32)	98.8%	97.6%	61.2%	18.8%
memory.objectSize.bin(40)	0.1%	0.1%	0.9%	2.3%
memory.objectSize.bin(48-72)	0.3%	0.4%	32.3%	0.1%
memory.objectSize.bin(80-136)	0.1%	0.1%	0.7%	0.1%
memory.objectSize.bin(144-392)	0.1%	0.0%	0.3%	0.1%
memory.objectSize.bin(400+)	0.0%	0.0%	0.1%	0.1%
pointer.fieldAccessDensity.value	102.844	31.544	212.953	120.546
pointer.nonrefFieldAccessDensity.value	21.786	30.320	184.858	55.719
pointer.refFieldAccessDensity.value	81.058	1.224	28.096	64.827

Table A.5: Whole program metrics for JOlden benchmarks–part 2 (continued)

Metric	PERIMETER	POWER	TSP	VORONOI
polymorphism.callSites.value	559	540	569	685
polymorphism.calls.value	7758325	642003	1361998	51624887
polymorphism.invokeDensity.value	45.766	0.498	25.344	110.500
polymorphism.receiverArity.bin(1)	95.2%	97.8%	97.9%	98.2%
polymorphism.receiverArity.bin(2)	2.0%	2.0%	1.9%	1.6%
polymorphism.receiverArity.bin(3+)	2.9%	0.2%	0.2%	0.1%
polymorphism.receiverArityCalls.bin(1)	37.0%	99.9%	100.0%	100.0%
polymorphism.receiverArityCalls.bin(2)	0.0%	0.1%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(3+)	63.0%	0.0%	0.0%	0.0%
polymorphism.receiverCacheMissRate.value	41.0%	0.1%	0.1%	0.0%
polymorphism.receiverPolyDensity.value	0.0483	0.02222	0.02109	0.01752
polymorphism.receiverPolyDensityCalls.value	0.62977	0.00088	0.00035	1e-05
polymorphism.targetArity.bin(1)	96.8%	98.7%	98.8%	99.0%
polymorphism.targetArity.bin(2)	1.1%	1.1%	1.1%	0.9%
polymorphism.targetArity.bin(3+)	2.1%	0.2%	0.2%	0.1%
polymorphism.targetArityCalls.bin(1)	42.9%	99.9%	100.0%	100.0%
polymorphism.targetArityCalls.bin(2)	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(3+)	57.1%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	37.4%	0.1%	0.1%	0.0%
polymorphism.targetPolyDensity.value	0.0322	0.01296	0.0123	0.01022
polymorphism.targetPolyDensityCalls.value	0.57136	0.00051	0.00022	1e-05
size.codeCoverage.value	13.4%	14.4%	13.6%	13.7%
size.deadCode.value	66151	66323	66323	66783
size.hot.value	398	409	110	354
size.hot.percentile	3.9%	3.7%	1.1%	3.3%
size.hotClasses.value	5	1	1	3
size.hotClasses.percentile	3.4%	0.7%	0.7%	2.1%
size.hotMethods.value	16	7	4	15
size.hotMethods.percentile	2.9%	1.3%	0.8%	2.7%
size.load.value	76430	77503	76751	77358
size.loadedClasses.value	285	281	278	281
size.loadedMethods.value	3658	3640	3636	3681
size.run.value	10279	11180	10428	10575

Table A.6: Whole program metrics for SPECjvm98 benchmarks–part 1

Metric	DB	JAVAC	JACK	JESS
base.bytes.value	95257968	266083104	322520528	529596000
base.classes.value	304	471	356	458
base.instructions.value	3837235350	2018819228	1027673717	1800201377
base.methods.value	117421985	104976930	62969582	112635997
base.objects.value	3212477	6419563	5992438	7947181
concurrency.contendedLock.percentile	N/A	50.0%	100.0%	100.0%
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	2.9%	7.1%	19.2%	2.6%
concurrency.lockDensity.value	14.673	7.897	15.748	2.746
data.arrayDensity.value	73.544	37.932	32.776	55.183
data.charArrayDensity.value	33.810	17.577	6.341	0.231
data.floatDensity.value	0.000	0.072	0.224	12.098
data.numArrayDensity.value	0.995	6.785	4.737	0.766
data.refArrayDensity.value	38.625	5.608	13.016	48.596
memory.averageObjectSize.value	29.6525	41.44879	53.82125	66.63948
memory.byteAllocationDensity.value	24.825	131.801	313.836	294.187
memory.objectAllocationDensity.value	0.837	3.180	5.831	4.415
memory.objectSize.bin(8)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	91.4%	14.6%	34.6%	17.8%
memory.objectSize.bin(24)	5.9%	41.9%	39.8%	16.6%
memory.objectSize.bin(32)	0.9%	20.7%	4.2%	24.8%
memory.objectSize.bin(40)	0.9%	6.2%	7.1%	0.9%
memory.objectSize.bin(48-72)	0.9%	11.4%	12.7%	39.6%
memory.objectSize.bin(80-136)	0.0%	4.2%	0.7%	0.2%
memory.objectSize.bin(144-392)	0.0%	0.8%	0.5%	0.1%
memory.objectSize.bin(400+)	0.0%	0.2%	0.4%	0.1%
pointer.fieldAccessDensity.value	123.279	163.866	156.614	151.145
pointer.nonrefFieldAccessDensity.value	50.430	93.849	95.091	89.639
pointer.refFieldAccessDensity.value	72.849	70.017	61.523	61.506

Table A.6: Whole program metrics for SPECjvm98 benchmarks–part 1 (continued)

Metric	DB	JAVAC	JACK	JESS
polymorphism.callSites.value	695	3233	1685	1309
polymorphism.calls.value	91503087	79053857	43790538	99520115
polymorphism.invokeDensity.value	23.846	39.158	42.611	55.283
polymorphism.receiverArity.bin(1)	97.8%	81.8%	98.2%	97.7%
polymorphism.receiverArity.bin(2)	2.0%	8.3%	1.4%	1.6%
polymorphism.receiverArity.bin(3+)	0.1%	9.9%	0.5%	0.7%
polymorphism.receiverArityCalls.bin(1)	100.0%	70.7%	82.5%	93.3%
polymorphism.receiverArityCalls.bin(2)	0.0%	12.6%	17.5%	2.4%
polymorphism.receiverArityCalls.bin(3+)	0.0%	16.6%	0.1%	4.3%
polymorphism.receiverCacheMissRate.value	0.0%	8.8%	1.7%	3.2%
polymorphism.receiverPolyDensity.value	0.02158	0.18187	0.0184	0.02292
polymorphism.receiverPolyDensityCalls.value	0.00024	0.29291	0.17547	0.06651
polymorphism.targetArity.bin(1)	98.6%	91.2%	98.8%	98.5%
polymorphism.targetArity.bin(2)	1.3%	3.4%	0.8%	0.9%
polymorphism.targetArity.bin(3+)	0.1%	5.4%	0.4%	0.6%
polymorphism.targetArityCalls.bin(1)	100.0%	86.8%	90.2%	93.4%
polymorphism.targetArityCalls.bin(2)	0.0%	1.7%	9.7%	2.4%
polymorphism.targetArityCalls.bin(3+)	0.0%	11.5%	0.1%	4.3%
polymorphism.targetCacheMissRate.value	0.0%	4.8%	1.7%	3.2%
polymorphism.targetPolyDensity.value	0.01439	0.08815	0.01246	0.01528
polymorphism.targetPolyDensityCalls.value	0.00022	0.13218	0.09784	0.06633
size.codeCoverage.value	15.9%	28.4%	26.8%	20.3%
size.deadCode.value	77166	95341	78814	89040
size.hot.value	152	2261	1475	564
size.hot.percentile	1.0%	6.0%	5.1%	2.5%
size.hotClasses.value	4	23	16	9
size.hotClasses.percentile	2.6%	7.7%	7.9%	3.1%
size.hotMethods.value	5	103	63	24
size.hotMethods.percentile	0.8%	7.3%	7.0%	2.3%
size.load.value	91730	133172	107697	111661
size.loadedClasses.value	304	471	356	458
size.loadedMethods.value	4299	5435	4559	4962
size.run.value	14564	37831	28883	22621

Table A.7: Whole program metrics for SPECjvm98 benchmarks–part 2

Metric	COMPRESS	MPEGAUDIO	RAYTRACE	MTRT
base.bytes.value	138418824	4965544	195899840	203094952
base.classes.value	310	356	324	324
base.instructions.value	12474432171	11491896285	2125242085	2173460119
base.methods.value	225970199	109746189	277506608	281538532
base.objects.value	10571	15099	6381047	6652175
concurrency.contendedLock.percentile	N/A	100.0%	N/A	100.0%
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	47.0%	21.5%	1.4%	1.4%
concurrency.lockDensity.value	0.000	0.001	0.166	0.323
data.arrayDensity.value	52.152	142.423	39.092	39.347
data.charArrayDensity.value	0.006	0.010	0.402	0.744
data.floatDensity.value	0.000	287.862	313.131	308.502
data.numArrayDensity.value	52.145	110.674	3.833	3.927
data.refArrayDensity.value	0.001	31.247	34.247	34.041
memory.averageObjectSize.value	13094.20339	328.86575	30.70027	30.53061
memory.byteAllocationDensity.value	11.096	0.432	92.178	93.443
memory.objectAllocationDensity.value	0.001	0.001	3.003	3.061
memory.objectSize.bin(8)	0.3%	0.2%	0.0%	0.0%
memory.objectSize.bin(16)	10.2%	14.6%	1.0%	1.9%
memory.objectSize.bin(24)	37.3%	37.5%	91.1%	89.4%
memory.objectSize.bin(32)	6.2%	5.8%	4.3%	4.7%
memory.objectSize.bin(40)	5.4%	3.5%	2.3%	2.5%
memory.objectSize.bin(48-72)	19.2%	15.8%	1.1%	1.2%
memory.objectSize.bin(80-136)	13.9%	15.4%	0.2%	0.3%
memory.objectSize.bin(144-392)	4.3%	6.3%	0.0%	0.0%
memory.objectSize.bin(400+)	3.1%	0.9%	0.0%	0.0%
pointer.fieldAccessDensity.value	189.557	95.682	157.955	157.739
pointer.nonrefFieldAccessDensity.value	95.144	38.897	97.490	97.243
pointer.refFieldAccessDensity.value	94.413	56.785	60.464	60.496

Table A.7: Whole program metrics for SPECjvm98 benchmarks–part 2 (continued)

Metric	COMPRESS	MPEGAUDIO	RAYTRACE	MTRT
polymorphism.callSites.value	606	847	1506	1510
polymorphism.calls.value	206227172	79265420	266910566	270241701
polymorphism.invokeDensity.value	16.532	6.898	125.591	124.337
polymorphism.receiverArity.bin(1)	97.5%	96.3%	95.3%	95.2%
polymorphism.receiverArity.bin(2)	2.3%	1.7%	3.3%	3.3%
polymorphism.receiverArity.bin(3+)	0.2%	2.0%	1.5%	1.5%
polymorphism.receiverArityCalls.bin(1)	100.0%	91.7%	91.0%	90.9%
polymorphism.receiverArityCalls.bin(2)	0.0%	7.9%	7.5%	7.4%
polymorphism.receiverArityCalls.bin(3+)	0.0%	0.4%	1.5%	1.6%
polymorphism.receiverCacheMissRate.value	0.0%	0.1%	0.2%	0.3%
polymorphism.receiverPolyDensity.value	0.02475	0.0366	0.04714	0.04768
polymorphism.receiverPolyDensityCalls.value	1e-05	0.08273	0.08987	0.09059
polymorphism.targetArity.bin(1)	98.4%	97.5%	99.1%	99.0%
polymorphism.targetArity.bin(2)	1.5%	2.2%	0.9%	0.9%
polymorphism.targetArity.bin(3+)	0.2%	0.2%	0.1%	0.1%
polymorphism.targetArityCalls.bin(1)	100.0%	91.8%	98.9%	99.0%
polymorphism.targetArityCalls.bin(2)	0.0%	8.1%	1.1%	1.0%
polymorphism.targetArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	0.0%	0.0%	0.0%	0.0%
polymorphism.targetPolyDensity.value	0.0165	0.02479	0.0093	0.00993
polymorphism.targetPolyDensityCalls.value	1e-05	0.0816	0.01062	0.01049
size.codeCoverage.value	16.0%	35.6%	20.2%	20.3%
size.deadCode.value	76248	78986	79131	79013
size.hot.value	396	3062	778	832
size.hot.percentile	2.7%	7.0%	3.9%	4.1%
size.hotClasses.value	4	8	7	7
size.hotClasses.percentile	2.5%	4.3%	4.0%	4.0%
size.hotMethods.value	7	18	18	20
size.hotMethods.percentile	1.1%	2.4%	2.4%	2.6%
size.load.value	90762	122698	99149	99161
size.loadedClasses.value	310	356	324	324
size.loadedMethods.value	4276	4555	4427	4427
size.run.value	14514	43712	20018	20148

A.2 Application-only metrics

Table A.8: Application-only metrics for small benchmarks

Metric	EMPTY	HELLOWORLD	AUTOMATA	COEFFICIENTS	LINPACK
base.bytes.value	0	0	32	6472	16
base.classes.value	1	1	1	6	1
base.instructions.value	0	4	135291	25792396	8026544
base.methods.value	0	1	2047	1699017	10619
base.objects.value	0	0	1	346	1
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	N/A	0.000	0.000	0.000	0.000
concurrency.lock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.lockDensity.value	N/A	0.000	0.000	0.000	0.000
data.arrayDensity.value	N/A	0.000	133.667	160.404	157.775
data.charArrayDensity.value	N/A	0.000	14.805	0.000	0.000
data.floatDensity.value	N/A	0.000	0.000	217.956	306.142
data.numArrayDensity.value	N/A	0.000	118.803	79.486	148.385
data.refArrayDensity.value	N/A	0.000	0.015	80.713	9.389
memory.averageObjectSize.value	N/A	N/A	32.0	18.7052	16.0
memory.byteAllocationDensity.value	N/A	0.000	0.237	0.251	0.002
memory.objectAllocationDensity.value	N/A	0.000	0.007	0.013	0.000
memory.objectSize.bin(8)	N/A	N/A	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	N/A	N/A	0.0%	66.8%	100.0%
memory.objectSize.bin(24)	N/A	N/A	0.0%	32.9%	0.0%
memory.objectSize.bin(32)	N/A	N/A	100.0%	0.0%	0.0%
memory.objectSize.bin(40)	N/A	N/A	0.0%	0.3%	0.0%
memory.objectSize.bin(48-72)	N/A	N/A	0.0%	0.0%	0.0%
memory.objectSize.bin(80-136)	N/A	N/A	0.0%	0.0%	0.0%
memory.objectSize.bin(144-392)	N/A	N/A	0.0%	0.0%	0.0%
memory.objectSize.bin(400+)	N/A	N/A	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	N/A	250.000	209.230	99.604	0.001
pointer.nonrefFieldAccessDensity.value	N/A	0.000	75.393	34.174	0.001
pointer.refFieldAccessDensity.value	N/A	250.000	133.837	65.430	0.000

Table A.8: Application-only metrics for small benchmarks (continued)

Metric	EMPTY	HELLOWORLD	AUTOMATA	COEFFICIENTS	LINPACK
polymorphism.callSites.value	0	1	8	85	39
polymorphism.calls.value	0	1	2064	1701599	10634
polymorphism.invokeDensity.value	N/A	250.000	15.256	65.973	1.325
polymorphism.receiverArity.bin(1)	N/A	100.0%	100.0%	100.0%	100.0%
polymorphism.receiverArity.bin(2)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArity.bin(3+)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(1)	N/A	100.0%	100.0%	100.0%	100.0%
polymorphism.receiverArityCalls.bin(2)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(3+)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverCacheMissRate.value	N/A	100.0%	0.4%	0.0%	0.4%
polymorphism.receiverPolyDensity.value	N/A	0.0	0.0	0.0	0.0
polymorphism.receiverPolyDensityCalls.value	N/A	0.0	0.0	0.0	0.0
polymorphism.targetArity.bin(1)	N/A	100.0%	100.0%	100.0%	100.0%
polymorphism.targetArity.bin(2)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArity.bin(3+)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(1)	N/A	100.0%	100.0%	100.0%	100.0%
polymorphism.targetArityCalls.bin(2)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(3+)	N/A	0.0%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	N/A	100.0%	0.4%	0.0%	0.4%
polymorphism.targetPolyDensity.value	N/A	0.0	0.0	0.0	0.0
polymorphism.targetPolyDensityCalls.value	N/A	0.0	0.0	0.0	0.0
size.codeCoverage.value	0.0%	57.1%	85.2%	41.1%	70.9%
size.deadCode.value	4	3	51	1399	307
size.hot.value	0	4	62	57	59
size.hot.percentile	N/A	100.0%	21.2%	5.8%	7.9%
size.hotClasses.value	0	1	1	2	1
size.hotClasses.percentile	N/A	100.0%	100.0%	40.0%	100.0%
size.hotMethods.value	0	1	3	4	2
size.hotMethods.percentile	N/A	100.0%	33.3%	11.4%	15.4%
size.load.value	4	7	344	2374	1056
size.loadedClasses.value	1	1	1	6	1
size.loadedMethods.value	2	2	10	87	14
size.run.value	0	4	293	975	749

Table A.9: Application-only metrics for small mulithreaded benchmarks

Metric	ROLLERCOASTER	TELECOM	VOLANO-CLIENT	VOLANO-SERVER
base.bytes.value	648	1201032	39488	35760
base.classes.value	4	13	19	38
base.instructions.value	318426	7734125	172969	242947
base.methods.value	19565	837629	11965	18080
base.objects.value	12	45024	1142	1045
concurrency.contendedLock.percentile	50.0%	100.0%	100.0%	88.9%
concurrency.contendedLockDensity.value	0.100	0.099	0.006	0.165
concurrency.lock.percentile	14.3%	50.0%	23.1%	25.0%
concurrency.lockDensity.value	32.800	5.503	7.949	9.467
data.arrayDensity.value	36.976	0.005	16.298	25.425
data.charArrayDensity.value	0.000	0.000	0.000	0.420
data.floatDensity.value	0.000	0.000	0.006	0.037
data.numArrayDensity.value	0.000	0.003	0.000	0.000
data.refArrayDensity.value	36.966	0.000	6.735	10.920
memory.averageObjectSize.value	54.0	26.67537	34.57793	34.2201
memory.byteAllocationDensity.value	2.035	155.290	228.295	147.193
memory.objectAllocationDensity.value	0.038	5.821	6.602	4.301
memory.objectSize.bin(8)	0.0%	0.0%	0.1%	0.0%
memory.objectSize.bin(16)	33.3%	0.0%	0.0%	1.7%
memory.objectSize.bin(24)	0.0%	66.6%	9.5%	9.6%
memory.objectSize.bin(32)	0.0%	33.3%	76.1%	75.6%
memory.objectSize.bin(40)	0.0%	0.0%	3.2%	3.4%
memory.objectSize.bin(48-72)	58.3%	0.0%	9.5%	8.7%
memory.objectSize.bin(80-136)	8.3%	0.0%	1.6%	0.9%
memory.objectSize.bin(144-392)	0.0%	0.0%	0.0%	0.1%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	170.350	136.140	169.504	157.952
pointer.nonrefFieldAccessDensity.value	68.663	59.497	79.141	69.933
pointer.refFieldAccessDensity.value	101.688	76.643	90.363	88.019

Table A.9: Application-only metrics for small mulithreaded benchmarks (continued)

Metric	RollerCoaster	Telecom	Volano-Client	Volano-Server
polymorphism.callSites.value	33	82	288	524
polymorphism.calls.value	40143	1007898	21097	28644
polymorphism.invokeDensity.value	126.067	130.318	121.970	117.902
polymorphism.receiverArity.bin(1)	100.0%	93.9%	98.6%	98.5%
polymorphism.receiverArity.bin(2)	0.0%	6.1%	0.0%	0.6%
polymorphism.receiverArity.bin(3+)	0.0%	0.0%	1.4%	1.0%
polymorphism.receiverArityCalls.bin(1)	100.0%	92.6%	89.4%	87.3%
polymorphism.receiverArityCalls.bin(2)	0.0%	7.4%	0.0%	3.7%
polymorphism.receiverArityCalls.bin(3+)	0.0%	0.0%	10.6%	9.0%
polymorphism.receiverCacheMissRate.value	0.1%	2.8%	5.3%	4.7%
polymorphism.receiverPolyDensity.value	0.0	0.06098	0.01389	0.01527
polymorphism.receiverPolyDensityCalls.value	0.0	0.07441	0.10561	0.12673
polymorphism.targetArity.bin(1)	100.0%	98.8%	99.3%	99.2%
polymorphism.targetArity.bin(2)	0.0%	1.2%	0.0%	0.2%
polymorphism.targetArity.bin(3+)	0.0%	0.0%	0.7%	0.6%
polymorphism.targetArityCalls.bin(1)	100.0%	98.5%	94.7%	94.3%
polymorphism.targetArityCalls.bin(2)	0.0%	1.5%	0.0%	0.7%
polymorphism.targetArityCalls.bin(3+)	0.0%	0.0%	5.3%	5.0%
polymorphism.targetCacheMissRate.value	0.1%	0.6%	3.3%	3.4%
polymorphism.targetPolyDensity.value	0.0	0.0122	0.00694	0.00763
polymorphism.targetPolyDensityCalls.value	0.0	0.01488	0.0528	0.05677
size.codeCoverage.value	81.0%	73.4%	57.6%	49.1%
size.deadCode.value	75	243	1759	5018
size.hot.value	33	360	572	1268
size.hot.percentile	10.3%	53.7%	23.9%	26.2%
size.hotClasses.value	3	7	7	12
size.hotClasses.percentile	75.0%	58.3%	43.8%	41.4%
size.hotMethods.value	4	26	29	58
size.hotMethods.percentile	40.0%	42.6%	29.3%	28.3%
size.load.value	395	914	4149	9861
size.loadedClasses.value	4	13	19	38
size.loadedMethods.value	13	92	210	370
size.run.value	320	671	2390	4843

A.2. Application-only metrics

Table A.10: Application-only metrics for medium and large benchmarks

Metric	JLex	Soot (Jimple)	Soot (Coffi)	SableCC (Java)	SableCC (WIG)
base.bytes.value	86688	196809392	64670968	26364016	38976080
base.classes.value	24	532	522	304	307
base.instructions.value	13607703	1291772489	1061844586	411052482	870603855
base.methods.value	414274	105284449	41936959	27041635	46634885
base.objects.value	3406	9329175	3120275	1133840	1652059
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000
concurrency.lock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.lockDensity.value	0.000	0.000	0.000	0.000	0.000
data.arrayDensity.value	53.677	35.553	93.999	38.868	50.287
data.charArrayDensity.value	1.418	0.000	0.000	0.000	0.000
data.floatDensity.value	0.000	0.269	0.008	0.000	0.000
data.numArrayDensity.value	47.001	16.159	68.625	11.209	22.190
data.refArrayDensity.value	0.611	9.696	7.425	13.274	10.251
memory.averageObjectSize.value	25.45156	21.09612	20.72605	23.25197	23.59243
memory.byteAllocationDensity.value	6.371	152.356	60.904	64.138	44.769
memory.objectAllocationDensity.value	0.250	7.222	2.939	2.758	1.898
memory.objectSize.bin(8)	0.1%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	12.9%	58.8%	61.1%	49.9%	48.0%
memory.objectSize.bin(24)	71.9%	30.9%	29.2%	19.8%	18.9%
memory.objectSize.bin(32)	0.1%	0.8%	0.9%	20.2%	23.3%
memory.objectSize.bin(40)	14.9%	8.8%	8.4%	10.1%	9.8%
memory.objectSize.bin(48-72)	0.0%	0.7%	0.5%	0.0%	0.0%
memory.objectSize.bin(80-136)	0.0%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(144-392)	0.1%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	129.197	190.751	185.943	164.895	147.958
pointer.nonrefFieldAccessDensity.value	46.428	70.976	52.264	40.635	30.821
pointer.refFieldAccessDensity.value	82.769	119.776	133.679	124.260	117.136

Table A.10: Application-only metrics for medium and large benchmarks (continued)

Metric	JLex	Soot (Jimple)	Soot (Coffi)	SableCC (Java)	SableCC (WIG)
polymorphism.callSites.value	636	3186	2839	2624	2513
polymorphism.calls.value	375638	90329861	39829368	28335329	47796441
polymorphism.invokeDensity.value	27.605	69.927	37.510	68.934	54.900
polymorphism.receiverArity.bin(1)	100.0%	93.3%	93.1%	92.4%	91.7%
polymorphism.receiverArity.bin(2)	0.0%	3.0%	3.2%	2.8%	3.3%
polymorphism.receiverArity.bin(3+)	0.0%	3.7%	3.7%	4.8%	5.0%
polymorphism.receiverArityCalls.bin(1)	100.0%	78.4%	72.0%	94.3%	94.7%
polymorphism.receiverArityCalls.bin(2)	0.0%	15.4%	14.6%	0.4%	0.5%
polymorphism.receiverArityCalls.bin(3+)	0.0%	6.2%	13.3%	5.3%	4.8%
polymorphism.receiverCacheMissRate.value	0.2%	4.4%	4.6%	0.2%	0.2%
polymorphism.receiverPolyDensity.value	0.0	0.06685	0.06939	0.07584	0.08277
polymorphism.receiverPolyDensityCalls.value	0.0	0.21637	0.27964	0.05712	0.05292
polymorphism.targetArity.bin(1)	100.0%	95.1%	95.1%	96.7%	96.7%
polymorphism.targetArity.bin(2)	0.0%	2.1%	2.3%	2.5%	2.6%
polymorphism.targetArity.bin(3+)	0.0%	2.8%	2.7%	0.8%	0.7%
polymorphism.targetArityCalls.bin(1)	100.0%	83.5%	85.1%	95.0%	95.6%
polymorphism.targetArityCalls.bin(2)	0.0%	13.9%	12.7%	0.2%	0.2%
polymorphism.targetArityCalls.bin(3+)	0.0%	2.6%	2.2%	4.7%	4.2%
polymorphism.targetCacheMissRate.value	0.2%	2.7%	2.1%	0.1%	0.2%
polymorphism.targetPolyDensity.value	0.0	0.04896	0.04931	0.03277	0.03303
polymorphism.targetPolyDensityCalls.value	0.0	0.16504	0.14904	0.04972	0.04386
size.codeCoverage.value	73.5%	57.7%	52.7%	72.1%	69.0%
size.deadCode.value	3778	19207	21428	11877	13228
size.hot.value	758	2759	1191	1099	874
size.hot.percentile	7.2%	10.5%	5.0%	3.6%	3.0%
size.hotClasses.value	5	29	16	12	11
size.hotClasses.percentile	21.7%	7.5%	4.6%	5.0%	4.4%
size.hotMethods.value	15	106	61	45	36
size.hotMethods.percentile	12.0%	9.4%	5.8%	4.3%	3.4%
size.load.value	14243	45446	45278	42606	42690
size.loadedClasses.value	24	532	522	304	307
size.loadedMethods.value	157	2188	2158	2223	2234
size.run.value	10465	26239	23850	30729	29462

A.2. Application-only metrics

Table A.11: Application-only metrics for JOlden benchmarks–part 1

Metric	BARNES-HUT	BISORT	EM3D	HEALTH	MST
base.bytes.value	125298336	3145704	160144	17784024	25198592
base.classes.value	9	2	4	8	6
base.instructions.value	1935371167	533615375	235376026	385173978	191438642
base.methods.value	44058916	15532248	68033	23391029	13086509
base.objects.value	7762308	131071	4009	1025342	1050624
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000	0.000
concurrency.lock.percentile	N/A	N/A	N/A	N/A	N/A
concurrency.lockDensity.value	0.000	0.000	0.000	0.000	0.000
data.arrayDensity.value	105.947	0.000	101.217	4.441	20.532
data.charArrayDensity.value	0.000	0.000	0.000	0.000	0.000
data.floatDensity.value	245.580	0.000	11.913	5.997	0.000
data.numArrayDensity.value	97.577	0.000	3.399	0.000	0.000
data.refArrayDensity.value	4.383	0.000	92.635	3.997	20.527
memory.averageObjectSize.value	16.14189	24.0	39.94612	17.34448	23.98441
memory.byteAllocationDensity.value	64.741	5.895	0.680	46.171	131.628
memory.objectAllocationDensity.value	4.011	0.246	0.017	2.662	5.488
memory.objectSize.bin(8)	0.0%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	99.2%	0.0%	0.2%	83.3%	0.2%
memory.objectSize.bin(24)	0.0%	100.0%	0.0%	16.6%	99.8%
memory.objectSize.bin(32)	0.8%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(40)	0.0%	0.0%	99.8%	0.0%	0.0%
memory.objectSize.bin(48-72)	0.1%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(80-136)	0.0%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(144-392)	0.0%	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	111.206	120.870	115.322	249.684	76.597
pointer.nonrefFieldAccessDensity.value	5.774	47.502	13.697	79.416	15.119
pointer.refFieldAccessDensity.value	105.432	73.367	101.625	170.268	61.478

Table A.11: Application-only metrics for JOlden benchmarks–part 1 (continued)

Metric	Barnes-Hut	BiSort	Em3d	Health	Mst
polymorphism.callSites.value	129	36	63	74	60
polymorphism.calls.value	36136608	6150231	852049	22003629	9414983
polymorphism.invokeDensity.value	18.672	11.526	3.620	57.126	49.180
polymorphism.receiverArity.bin(1)	96.9%	100.0%	100.0%	100.0%	100.0%
polymorphism.receiverArity.bin(2)	3.1%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArity.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(1)	86.7%	100.0%	100.0%	100.0%	100.0%
polymorphism.receiverArityCalls.bin(2)	13.3%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverCacheMissRate.value	3.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverPolyDensity.value	0.03101	0.0	0.0	0.0	0.0
polymorphism.receiverPolyDensityCalls.value	0.1326	0.0	0.0	0.0	0.0
polymorphism.targetArity.bin(1)	96.9%	100.0%	100.0%	100.0%	100.0%
polymorphism.targetArity.bin(2)	3.1%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArity.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(1)	86.7%	100.0%	100.0%	100.0%	100.0%
polymorphism.targetArityCalls.bin(2)	13.3%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	3.0%	0.0%	0.0%	0.0%	0.0%
polymorphism.targetPolyDensity.value	0.03101	0.0	0.0	0.0	0.0
polymorphism.targetPolyDensityCalls.value	0.1326	0.0	0.0	0.0	0.0
size.codeCoverage.value	80.5%	85.4%	79.0%	91.5%	82.4%
size.deadCode.value	392	81	150	84	127
size.hot.value	160	137	42	65	175
size.hot.percentile	9.9%	29.0%	7.5%	7.2%	29.5%
size.hotClasses.value	2	1	1	3	4
size.hotClasses.percentile	22.2%	50.0%	25.0%	37.5%	66.7%
size.hotMethods.value	7	4	1	4	10
size.hotMethods.percentile	13.0%	33.3%	5.6%	15.4%	32.3%
size.load.value	2012	554	713	991	720
size.loadedClasses.value	9	2	4	8	6
size.loadedMethods.value	73	15	22	29	36
size.run.value	1620	473	563	907	593

Table A.12: Application-only metrics for JOlden benchmarks–part 2

Metric	PERIMETER	POWER	TSP	VORONOI
base.bytes.value	14493376	656104	786384	28017056
base.classes.value	10	6	2	6
base.instructions.value	168779823	1236657898	50508412	466327427
base.methods.value	12196762	16939154	1303323	54436748
base.objects.value	452921	22403	16383	1161916
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	N/A	N/A	N/A	N/A
concurrency.lockDensity.value	0.000	0.000	0.000	0.000
data.arrayDensity.value	0.000	97.433	0.000	28.303
data.charArrayDensity.value	0.000	0.000	0.000	0.000
data.floatDensity.value	0.000	461.224	499.775	226.489
data.numArrayDensity.value	0.000	96.487	0.000	0.000
data.refArrayDensity.value	0.000	0.162	0.000	27.724
memory.averageObjectSize.value	31.99979	29.28643	48.0	24.11281
memory.byteAllocationDensity.value	85.871	0.531	15.569	60.080
memory.objectAllocationDensity.value	2.684	0.018	0.324	2.492
memory.objectSize.bin(8)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	0.0%	0.0%	0.0%	4.2%
memory.objectSize.bin(24)	0.0%	50.0%	0.0%	93.0%
memory.objectSize.bin(32)	100.0%	44.6%	0.0%	0.0%
memory.objectSize.bin(40)	0.0%	0.0%	0.0%	2.8%
memory.objectSize.bin(48-72)	0.0%	5.4%	100.0%	0.0%
memory.objectSize.bin(80-136)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(144-392)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	103.041	32.883	221.376	120.687
pointer.nonrefFieldAccessDensity.value	21.738	31.618	191.774	55.776
pointer.refFieldAccessDensity.value	81.303	1.264	29.602	64.911

Table A.12: Application-only metrics for JOlden benchmarks–part 2 (continued)

Metric	Perimeter	Power	Tsp	Voronoi
polymorphism.callSites.value	49	34	56	175
polymorphism.calls.value	7745539	631062	1254194	51613567
polymorphism.invokeDensity.value	45.891	0.510	24.831	110.681
polymorphism.receiverArity.bin(1)	69.4%	100.0%	100.0%	100.0%
polymorphism.receiverArity.bin(2)	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArity.bin(3+)	30.6%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(1)	36.9%	100.0%	100.0%	100.0%
polymorphism.receiverArityCalls.bin(2)	0.0%	0.0%	0.0%	0.0%
polymorphism.receiverArityCalls.bin(3+)	63.1%	0.0%	0.0%	0.0%
polymorphism.receiverCacheMissRate.value	41.1%	0.0%	0.0%	0.0%
polymorphism.receiverPolyDensity.value	0.30612	0.0	0.0	0.0
polymorphism.receiverPolyDensityCalls.value	0.63073	0.0	0.0	0.0
polymorphism.targetArity.bin(1)	77.6%	100.0%	100.0%	100.0%
polymorphism.targetArity.bin(2)	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArity.bin(3+)	22.4%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(1)	42.8%	100.0%	100.0%	100.0%
polymorphism.targetArityCalls.bin(2)	0.0%	0.0%	0.0%	0.0%
polymorphism.targetArityCalls.bin(3+)	57.2%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	37.5%	0.0%	0.0%	0.0%
polymorphism.targetPolyDensity.value	0.22449	0.0	0.0	0.0
polymorphism.targetPolyDensityCalls.value	0.57225	0.0	0.0	0.0
size.codeCoverage.value	90.9%	95.1%	90.8%	59.6%
size.deadCode.value	78	95	87	720
size.hot.value	393	402	54	351
size.hot.percentile	50.6%	21.9%	6.3%	33.0%
size.hotClasses.value	5	1	1	2
size.hotClasses.percentile	50.0%	16.7%	50.0%	33.3%
size.hotMethods.value	16	6	2	15
size.hotMethods.percentile	38.1%	20.7%	15.4%	34.1%
size.load.value	855	1928	946	1783
size.loadedClasses.value	10	6	2	6
size.loadedMethods.value	50	32	16	73
size.run.value	777	1833	859	1063

A.2. Application-only metrics

Table A.13: Application-only metrics for SPECjvm98 benchmarks–part 1

Metric	DB	JAVAC	JACK	JESS
base.bytes.value	250432	63369728	4836304	138894184
base.classes.value	14	175	66	158
base.instructions.value	1116021821	912643681	253793323	1555320922
base.methods.value	1560372	55325842	7380830	91077976
base.objects.value	15646	2116931	154799	3883094
concurrency.contendedLock.percentile	N/A	N/A	N/A	N/A
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	50.0%	20.0%	100.0%	50.0%
concurrency.lockDensity.value	0.017	0.670	1.142	0.009
data.arrayDensity.value	86.161	15.471	22.538	58.064
data.charArrayDensity.value	0.000	3.488	1.275	0.000
data.floatDensity.value	0.000	0.000	0.000	13.998
data.numArrayDensity.value	2.486	0.359	11.476	0.833
data.refArrayDensity.value	83.675	3.543	2.042	53.067
memory.averageObjectSize.value	16.00614	29.93472	31.24248	35.76895
memory.byteAllocationDensity.value	0.224	69.435	19.056	89.303
memory.objectAllocationDensity.value	0.014	2.320	0.610	2.497
memory.objectSize.bin(8)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(16)	99.9%	14.8%	8.4%	0.1%
memory.objectSize.bin(24)	0.0%	22.7%	1.6%	18.3%
memory.objectSize.bin(32)	0.0%	45.1%	81.1%	48.9%
memory.objectSize.bin(40)	0.0%	11.3%	8.9%	0.0%
memory.objectSize.bin(48-72)	0.0%	6.1%	0.0%	32.7%
memory.objectSize.bin(80-136)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(144-392)	0.0%	0.0%	0.0%	0.0%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	129.693	208.349	161.333	155.981
pointer.nonrefFieldAccessDensity.value	0.254	120.080	89.463	91.654
pointer.refFieldAccessDensity.value	129.439	88.269	71.870	64.327

Table A.13: Application-only metrics for SPECjvm98 benchmarks–part 1 (continued)

Metric	DB	JAVAC	JACK	JESS
polymorphism.callSites.value	128	2617	1110	737
polymorphism.calls.value	89942094	65501655	21539366	92066118
polymorphism.invokeDensity.value	80.592	71.771	84.870	59.194
polymorphism.receiverArity.bin(1)	100.0%	78.4%	98.9%	98.4%
polymorphism.receiverArity.bin(2)	0.0%	9.7%	0.5%	0.8%
polymorphism.receiverArity.bin(3+)	0.0%	12.0%	0.5%	0.8%
polymorphism.receiverArityCalls.bin(1)	100.0%	72.6%	90.0%	99.2%
polymorphism.receiverArityCalls.bin(2)	0.0%	15.1%	9.9%	0.1%
polymorphism.receiverArityCalls.bin(3+)	0.0%	12.3%	0.2%	0.8%
polymorphism.receiverCacheMissRate.value	0.0%	7.2%	3.4%	0.4%
polymorphism.receiverPolyDensity.value	0.0	0.21628	0.01081	0.01628
polymorphism.receiverPolyDensityCalls.value	0.0	0.27395	0.10028	0.00834
polymorphism.targetArity.bin(1)	100.0%	89.7%	99.0%	98.9%
polymorphism.targetArity.bin(2)	0.0%	3.8%	0.4%	0.4%
polymorphism.targetArity.bin(3+)	0.0%	6.5%	0.5%	0.7%
polymorphism.targetArityCalls.bin(1)	100.0%	92.0%	90.0%	99.2%
polymorphism.targetArityCalls.bin(2)	0.0%	1.9%	9.9%	0.0%
polymorphism.targetArityCalls.bin(3+)	0.0%	6.1%	0.2%	0.8%
polymorphism.targetCacheMissRate.value	0.0%	3.1%	3.4%	0.4%
polymorphism.targetPolyDensity.value	0.0	0.10279	0.00991	0.01085
polymorphism.targetPolyDensityCalls.value	0.0	0.08007	0.10026	0.00818
size.codeCoverage.value	70.6%	58.8%	79.9%	52.0%
size.deadCode.value	1890	18397	4703	10736
size.hot.value	67	2759	802	476
size.hot.percentile	1.5%	10.5%	4.3%	4.1%
size.hotClasses.value	1	20	5	6
size.hotClasses.percentile	12.5%	13.8%	9.6%	4.3%
size.hotMethods.value	2	124	17	19
size.hotMethods.percentile	3.8%	15.9%	5.8%	4.3%
size.load.value	6436	44664	23424	22370
size.loadedClasses.value	14	175	66	158
size.loadedMethods.value	162	1259	440	781
size.run.value	4546	26267	18721	11634

Table A.14: Application-only metrics for SPECjvm98 benchmarks–part 2

Metric	Compress	Mpegaudio	Raytrace	Mtrt
base.bytes.value	9056	17736	122551072	125843104
base.classes.value	22	62	35	35
base.instructions.value	12472768728	11488941053	2097923733	2121758239
base.methods.value	225926502	108336917	276304326	279242302
base.objects.value	312	992	5039690	5189653
concurrency.contendedLock.percentile	N/A	N/A	N/A	100.0%
concurrency.contendedLockDensity.value	0.000	0.000	0.000	0.000
concurrency.lock.percentile	100.0%	100.0%	50.0%	50.0%
concurrency.lockDensity.value	0.000	0.000	0.000	0.000
data.arrayDensity.value	52.150	142.445	38.974	39.129
data.charArrayDensity.value	0.000	0.000	0.000	0.000
data.floatDensity.value	0.000	287.936	316.809	315.536
data.numArrayDensity.value	52.150	110.699	3.694	3.665
data.refArrayDensity.value	0.000	31.254	34.690	34.867
memory.averageObjectSize.value	29.02564	17.87903	24.31718	24.24885
memory.byteAllocationDensity.value	0.001	0.002	58.415	59.311
memory.objectAllocationDensity.value	0.000	0.000	2.402	2.446
memory.objectSize.bin(8)	0.0%	0.4%	0.0%	0.0%
memory.objectSize.bin(16)	42.0%	94.5%	1.0%	1.9%
memory.objectSize.bin(24)	41.7%	1.0%	94.2%	93.3%
memory.objectSize.bin(32)	0.0%	0.6%	4.8%	4.8%
memory.objectSize.bin(40)	0.3%	0.2%	0.0%	0.0%
memory.objectSize.bin(48-72)	8.0%	2.5%	0.0%	0.1%
memory.objectSize.bin(80-136)	8.0%	0.7%	0.0%	0.0%
memory.objectSize.bin(144-392)	0.0%	0.1%	0.0%	0.0%
memory.objectSize.bin(400+)	0.0%	0.0%	0.0%	0.0%
pointer.fieldAccessDensity.value	189.575	95.693	158.432	158.522
pointer.nonrefFieldAccessDensity.value	95.152	38.900	97.729	97.611
pointer.refFieldAccessDensity.value	94.422	56.793	60.703	60.910

Table A.14: Application-only metrics for SPECjvm98 benchmarks–part 2 (continued)

Metric	COMPRESS	MPEGAUDIO	RAYTRACE	MTRT
polymorphism.callSites.value	54	326	936	939
polymorphism.calls.value	206202050	79209661	266335521	269129344
polymorphism.invokeDensity.value	16.532	6.894	126.952	126.843
polymorphism.receiverArity.bin(1)	98.1%	94.8%	94.2%	94.1%
polymorphism.receiverArity.bin(2)	1.9%	0.3%	3.5%	3.6%
polymorphism.receiverArity.bin(3+)	0.0%	4.9%	2.2%	2.2%
polymorphism.receiverArityCalls.bin(1)	100.0%	91.7%	91.0%	90.9%
polymorphism.receiverArityCalls.bin(2)	0.0%	7.9%	7.5%	7.4%
polymorphism.receiverArityCalls.bin(3+)	0.0%	0.4%	1.5%	1.6%
polymorphism.receiverCacheMissRate.value	0.0%	0.1%	0.2%	0.3%
polymorphism.receiverPolyDensity.value	0.01852	0.05215	0.05769	0.05857
polymorphism.receiverPolyDensityCalls.value	0.0	0.08272	0.09005	0.09096
polymorphism.targetArity.bin(1)	98.1%	96.3%	99.6%	99.5%
polymorphism.targetArity.bin(2)	1.9%	3.4%	0.4%	0.5%
polymorphism.targetArity.bin(3+)	0.0%	0.3%	0.0%	0.0%
polymorphism.targetArityCalls.bin(1)	100.0%	91.8%	98.9%	98.9%
polymorphism.targetArityCalls.bin(2)	0.0%	8.1%	1.1%	1.1%
polymorphism.targetArityCalls.bin(3+)	0.0%	0.0%	0.0%	0.0%
polymorphism.targetCacheMissRate.value	0.0%	0.0%	0.0%	0.0%
polymorphism.targetPolyDensity.value	0.01852	0.03681	0.00427	0.00532
polymorphism.targetPolyDensityCalls.value	0.0	0.08161	0.01064	0.01053
size.codeCoverage.value	77.6%	90.9%	83.7%	84.5%
size.deadCode.value	1471	3509	1817	1733
size.hot.value	396	3058	731	754
size.hot.percentile	7.8%	8.7%	7.8%	8.0%
size.hotClasses.value	4	8	7	7
size.hotClasses.percentile	23.5%	16.7%	23.3%	23.3%
size.hotMethods.value	7	17	16	17
size.hotMethods.percentile	12.3%	7.6%	8.8%	9.3%
size.load.value	6555	38484	11181	11193
size.loadedClasses.value	22	62	35	35
size.loadedMethods.value	162	439	294	294
size.run.value	5084	34975	9364	9460

Appendix B
On-line Metrics Resources

B.1 Dynamic Metrics

An online database of dynamic metrics is available for public consultation at:

`http://www.sable.mcgill.ca/metrics/`

B.2 **J*

**J* is made publicly available, at no cost, under the terms of the GNU Lesser General Public License (LGPL). The source code of **J*, and all relevant documentation can be obtained from McGill University's Sable Research Group at:

`http://www.sable.mcgill.ca/starj/`

Bibliography

[AAB$^+$00] Bowen Alpern, Dick Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio J. Serrano, Janicc Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ADG$^+$99] 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 Inc., 1999.

[AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, USA, 2001, pages 168–179. ACM Press.

[ASC03] 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, Mar. 2003.

[Ashes] Ashes suite collection.
<http://www.sable.mcgill.ca/ashes/> .

[Asp] Xerox Corporation Palo Alto Research Center (PARC). Aspectj. <http://www.eclipse.org/aspectj/> .

[Bal99] Thomas Ball. The concept of dynamic analysis. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulouse, France, 1999, pages 216–234. Springer-Verlag.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan./Feb. 1999.

[BJ03] Martin Burtscher and Metha Jeeradit. Compressing extended program traces using value predictors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003, pages 159–169.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montréal, Québec, Canada, 1998, pages 258–268. ACM Press.

[BPW01] Siobhán Byrne, James Power, and John Waldron. A dynamic comparison of the SPEC98 and Java Grande benchmark suites. In *Workshop on Intermediate Presentation Engineering for the Java Virtual Machine*, 2001.

[Bro03] Rhodes H. F. Brown. STEP: A framework for the efficient encoding of general trace data. Master's thesis, McGill University, Montréal, Québec, Canada, 2003.

[CH00] 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 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, Canada, 2000, pages 57–69. ACM Press.

[CK94] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun. 1994.

[CM01] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java controller. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, Sep. 2001, pages 280–291.

[Cor89] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[CS95] Neville I. Churcher and Martin J. Shepperd. Comments on "a metrics suite for object oriented design". *IEEE Transactions on Software Engineering*, 21(3):263–265, Mar. 1995.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1995, pages 77–101. Springer-Verlag.

[DGH$^+$04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Victoria, British Columbia, Canada, Oct. 2004. (To appear).

[DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999, pages 92–115.

[DHPW01] 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*, Palo Alto, California, USA, 2001, pages 106–115. ACM Press.

[Dri01] Karel Driesen. *Efficient Polymorphic Calls*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Salt Lake City, Utah, USA, 1984, pages 297–302. ACM Press.

[EGD03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Anaheim, California, USA, 2003, pages 169–186. ACM Press.

[Els78] James L. Elshoff. An investigation into the effects of the counting method used on software science measurements. *ACM SIGPLAN Notices*, 13(2):30–45, 1978.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, May 9, 2003, pages 24–27.

[Eva84] Michael Evangelist. An analysis of control flow complexity. In *Proceedings of the IEEE Computer Software and Applications Conference (COMPSAC)*, 1984, pages 388–396. IEEE Press.

[Fen94] Norman E. Fenton. Software measurement: a necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar. 1994.

[FN99] Norman E. Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2–3):149–157, Jul. 1999.

[FN00] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, 2000, pages 357–370. ACM Press.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach*. PWS Publishing Company, 1997.

[Gag02] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montréal, Québec, Canada, Dec. 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[GR01] Neelam Gupta and Praveen Rao. Program execution based module cohesion measurement. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, California, USA, Nov. 2001, pages 144–153.

[GRS00] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, Canada, 2000, pages 334–344. ACM Press.

[GZIP] Jean-loup Gailly, Mark Adler, and the Free Software Fondation, Inc. The gzip (GNU zip) compression tool.
<http://www.gzip.org> .

[Hal72] M. H. Halstead. Natural laws controlling algorithm structure? *ACM SIGPLAN Notices*, 7(2):19–26, 1972.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[HF82] Peter G. Hamer and Gillian D. Frewin. M.h. halstead's software science - a critical examination. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Tokyo, Japan, 1982, pages 197–206. IEEE Computer Society Press.

[HM81] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *ACM SIGPLAN Notices*, 16(3):63–74, 1981.

[HM96] Martin Hitz and Behzad Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.

[JGF] EPCC. The Java Grande Forum benchmark suite.
<http://www.epcc.ed.ac.uk/javagrande/javag.html> .

[JVMPI] Sun Microsystems, Inc. Java virtual machine profiler interface (JVMPI).
<http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html> .

[KC01] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, USA, 2001, pages 156–167. ACM Press.

[KKO02] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Seattle, Washington, USA, Nov. 2002, pages 142–160.

[Kob97] Makoto Kobayashi. Memory reference metrics and instruction trace sampling. In *Proceedings of the IEEE International Performance, Computing and Communications Conference*, Feb. 1997, pages 301–307.

[KST$^+$86] Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, 1986.

[LJN$^+$00] 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*, Santa Fe, New Mexico, USA, 2000, pages 22–33. ACM Press.

[LY99] Time Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[Mit02] Aine Mitchell. Dynamic coupling and cohesion metrics for Java programs. Master's thesis, National University of Ireland, Maynooth, Co. Kildare, Ireland, 2002.

[MP03a] Aine Mitchell and James Power. Run-time cohesion metrics for the analysis of Java programs - preliminary results from the SPEC and Grande suites. Technical Report NUIM-CS-TR-2003-08, National University of Ireland, Maynooth, Co. Kildare, Ireland, Apr. 2003.

[MP03b] Aine Mitchell and James Power. Run-time coupling metrics for the analysis of Java programs - preliminary results from the SPEC and Grande suites. Technical Report NUIM-CS-TR-2003-07, National University of Ireland, Maynooth, Co. Kildare, Ireland, Apr. 2003.

[Mye77] Glenford J. Myers. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notices*, 12(10):61–64, 1977.

[Nej88] Brian A. Nejmeh. NPATH: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[OT89] Linda M. Ott and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Pittsburgh, Pennsylvania, USA, 1989, pages 198–204. ACM Press.

[OT93] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the First International Software Metrics Symposium*, May 1993, pages 71–81.

[Piw82] Paul Piwowarski. A nesting level complexity measure. *ACM SIGPLAN Notices*, 17(9):44–50, 1982.

[PW02] James F. Power and John T. Waldron. A method-level analysis of object-oriented techniques in Java applications. Technical Report NUIM-CS-TR-2002-07, National University of Ireland, Maynooth, Co. Kildare, Ireland, Sep. 2002.

[QH04] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in JVMs. In *Proceedings of the USENIX-ACM SIGPLAN Virtual Machine*

Research & Technology Symposium (VM), San Jose, California, USA, May 2004.

[QHV02] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction*, 2002, number 2304 in Lecture Notes in Computer Science, pages 325–341.

[Rig96] Fabrizio Riguzzi. A survey of software metrics. Technical Report DEIS-LIA-96-010, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, 1996. LIA Series no. 17.

[RR96] Niklas Röjemo and Colin Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvania, USA, 1996, pages 34–41. ACM Press.

[RR00] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, San Francisco, California, USA, 2000, pages 71–77. ACM Press.

[RR01] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, 2001, pages 221–230.

[Sal82] Norman F. Salt. Defining software science counting strategies. *ACM SIGPLAN Notices*, 17(3):58–67, 1982.

[SHR^{+}00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Valle-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, USA, 2000, pages 264–280. ACM Press.

[SKS02] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, Berlin, Germany, 2002, pages 64–75. ACM Press.

[Spec] Standard Performance Evaluation Corp. SPEC JVM98 benchmarks. <http://www.specbench.org/osg/jvm98/> .

[SSGS01] 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*, Cambridge, Massachusetts, USA, 2001, pages 194–205. ACM Press.

[Volano] Volano benchmark. <http://www.volano.com/benchmarks.html> .

[VRGH$^+$00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, 2000, pages 18–34.

[Wey88] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sep. 1988.

[YAR99] S.M. Yacoub, H.H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *Sixth IEEE International Symposium on Software Metrics*, Nov. 1999, pages 50–61.

[YAR00] S.M. Yacoub, H.H. Ammar, and T. Robinson. A methodology for architectural-level risk assessment using dynamic metrics. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'2000)*, San Jose, California, USA, Oct. 2000, pages 210–221.

[ZLAV00] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, Canada, 2000, pages 134–144. ACM Press.

[Zus92] Horst Zuse. Support of experimentation by measurement theory. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, 1992, pages 137–140. Springer-Verlag.