Characterizing the Design and Performance of Interactive Java Applications

Dmitrijs Zaparanuks
Faculty of Informatics
University of Lugano
Email: zaparand@usi.ch

Matthias Hauswirth
Faculty of Informatics
University of Lugano
Email: Matthias.Hauswirth@unisi.ch

Abstract

When designers *of Java runtime systems evaluate the performance of their systems for the purpose of running client-side Java applications, they normally use the Dacapo and SPEC JVM benchmark suites. However, when* users *of those Java runtime systems run client applications, they usually run interactive applications such as Eclipse or NetBeans.*

In this paper we study whether this mismatch is a problem: Do the prevalent Java client-side benchmark suites faithfully represent the characteristics of real-world Java client applications? To answer this question we characterize benchmarks and applications using three kinds of metrics: static metrics, architecture-independent dynamic metrics, and hardware performance counters.

We find that real-world applications significantly differ from existing benchmarks. Our finding indicates that the current benchmark suites should be augmented to more faithfully represent the large segment of interactive applications.

1. Introduction

System designers and researchers use benchmarks to evaluate the benefit of their innovations. If an innovation, such as a new micro-architectural feature, a new compiler optimization, or a new garbage collection algorithm, speeds up a given set of benchmarks, the innovation is considered beneficial. Such an evaluation is only meaningful if the benchmarks used are representative of the applications used in the real world. This paper studies how the two prevalent client-side Java benchmark suites, SPEC JVM 2008 [1] and Dacapo [2], compare to real-world interactive Java applications such as Eclipse and NetBeans.

Figures 1 and 2 contrast the hottest code of a SPEC JVM 2008 benchmark (compress) with the hottest method in a real-world run of a commonly used interactive application (Eclipse). The two pieces of code look significantly different: the compress code is dominated by arithmetic operations and relatively complex control flow, while the eclipse code is centered around object-oriented aspects such as virtual method calls and field accesses. These significant differences in the hottest regions of code raise the question whether the current client-side Java benchmarks indeed are representative of the rich-client Java applications used in the real world.

```
next_byte: while ((c = input.readByte()) != -1) {
  inCount++;
  fcode = (((int) c << maxBits) + ent);
  i = ((c << hshift) ^ ent);
  int temphtab = htab.of(i);
  if (temphtab == fcode) {
    ent = codetab.of(i);
    continue next_byte;
  }
  if (temphtab >= 0) {
    disp = hsizeReg - i;
    if (i == 0) {
      disp = 1;
    }
    do {
      if ((i -= disp) < 0) {
        i += hsizeReg;
      }
      temphtab = htab.of(i);
      if (temphtab == fcode) {
        ent = codetab.of(i);
        continue next_byte;
      }
    } while (temphtab > 0);
  }
  output(ent);
  outCount++;
  ent = c;
  if (freeEntry < maxMaxCode) {
    codetab.set(i, freeEntry++);
    htab.set(i, fcode);
  } else if (inCount>=checkpoint&&blockCompress!=0) {
    clBlock();
  }
}
```

Figure 1. Hottest code of SPEC JVM 2008 compress

This paper compares benchmarks to real-world applications along four dimensions: using (1) program size, (2) static design metrics, (3) dynamic platform-independent metrics, and (4) hardware-performance metrics. Our characterization strives to prove that current benchmark suites should be extended to include code more similar to interactive applications.

The remainder of this paper is structured as follows: Section 2 discusses the particularities of interactive GUI

```
void addIndexEntry
(char[] category, char[] key, String documentName) {
 HashtableOfObject referenceTable;
 if (documentName.equals(this.lastDocumentName))
  referenceTable = this.lastReferenceTable;
 else {
  referenceTable = (HashtableOfObject)this.
  docsToReferences.get(documentName);
  if (referenceTable == null)
   this.docsToReferences.put
   (documentName, referenceTable =
   new HashtableOfObject(3));
  this.lastDocumentName = documentName;
  this.lastReferenceTable = referenceTable;
 }
 SimpleWordSet existingWords =
 (SimpleWordSet) referenceTable.get(category);
 if (existingWords == null)
  referenceTable.put
  (category, existingWords = new SimpleWordSet(1));
 existingWords.add(this.allWords.add(key));
}
```

Figure 2. Hottest code of interactive Eclipse IDE

applications and introduces the applications we used for this study. Section 3 describes our experimental methodology. Sections 4, 5, 6, and 7 characterize and compare benchmarks to applications on the level of code size, static design metrics, dynamic platform-independent metrics, and dynamic hardware performance metrics. Section 8 compares the programs using a combination of all the metrics. Section 9 discusses related work, and Section 10 concludes.

2. Interactive Applications

Interactive applications with graphical user interfaces are based on GUI toolkits. Many GUI toolkits are implemented as frameworks, meaning that they provide the core of the application and require application code to extend certain predefined extension points. The application usually is responsible for two aspects: GUI creation (setting up the user interface by creating windows containing GUI widgets), and event handling (respond to user events it receives through the GUI toolkit). These two aspects usually are intertwined: while some of the GUI creation happens at startup time (e.g. creating the main window), many GUI creation activities happen throughout the application's runtime as a response to user events (e.g. creating a dialog to save a file).

In object-oriented GUI toolkits, such as the ones available for Java, application code for GUI creation usually instantiates GUI widget classes provided by the GUI toolkit. However, application code often also overrides existing toolkit widgets by providing subclasses with specialized functionality. Application code for event handling generally has to implement interfaces provided by the GUI toolkit, so that the toolkit can call back into the application to deliver user events. Today's object-oriented GUI toolkits implement this call-back mechanism by using the observer design pattern [3].

Given the object-oriented design imposed upon the application by the GUI toolkit, we suspect that many GUI applications maintain a more object-oriented design throughout their code, including the parts of the applications that are not related to the GUI. Thus their design may lead to a more liberal use of reference-based data structures, dynamic memory allocation, and dynamic dispatch than what we observe in some of the more traditional batch-oriented applications in standard benchmark suites.

2.1. Applications used in this study

Table 1 shows the interactive applications we selected for this study. We focused exclusively on open-source applications written in Java. We selected rich-client applications, that is, programs installed on a local machine, which can make full use of all the features of Java's extensive class library. We included applications based on both prevalent Java GUI toolkits, Swing [4], which is part of the Java platform, and SWT [5], which is part of IBM's Eclipse rich-client platform [6].

Application	Version	Description
argouml	0.28	UML CASE tool
crossword	0.3.5	CrosswordSage puzzle editor
gantt	2.0.9	GanttProject Gantt chart editor
jedit	4.3pre16	Programmer's text editor
jfreechart	1.0.13	Chart library showing temporal data
jhotdraw	7.1	Vector graphics editor
netbeans	6.5.1	Integrated development environment
eclipse	3.4.2	Integrated development environment

Table 1. Interactive Java applications used in this study

We selected a total of 8 interactive applications, 7 based on Swing and 1 based on SWT[1]. With the exception of CrosswordSage, a small application focused on a narrow task, which we included because of its high all-time activity score on SourceForge, all the selected applications represent feature-rich programs. In terms of functionality, applications such as NetBeans and Eclipse are on par or even ahead of comparable native (non-Java) applications, and they are used by a large number of users (for example, the Galileo release of Eclipse was downloaded over one million times in its first 27 days [7]).

3. Experimental methodology

In this section we describe the aspects of our experimental methodology that apply to all subsequent sections.

3.1. Sessions

We compared two benchmark suites, Dacapo 2006-10-MR2 (D) and SPEC JVM 2008 (S), with a set of interactive

1. Outside its original use in the Eclipse IDE, the adoption of the SWT GUI toolkit in open-source applications is still relatively rare.

applications (I). To collect the dynamic and hardware metrics, we used the *large* inputs of the Dacapo suite, and we set the SPEC JVM 2008 iteration time to 27 seconds, to match the average duration of the Dacapo benchmarks. We reran all these experiments 10 times and averaged the metrics over the ten runs.

Type	Description
SS start-stop	Open and immediately close the application
EE exploration	Explore all toolbar buttons and menu items
RR realistic run	Productively use the application

Table 2. Types of interactive sessions

Running an interactive application requires user interactions. We thus planned three types of sessions (**Table 2**) for each of our eight applications. We manually ran each application for each type of session three times, and averaged the metrics over the three runs.

3.2. Infrastructure

We ran all programs on a Sun HotSpot virtual machine (build 1.6.0_13-b03) in server mode with a heap size of 1024 MB, running on Linux 2.6.25 with the perform2 kernel patch, on top of a 1.60 GHz 8-core Intel Xeon E5310 machine.

3.3. Metrics Collection

We collected the static design metrics using Jdepend [8]. We developed a bytecode instrumentation agent, using the ASM toolkit [9], to dynamically instrument loaded classes with the counting instrumentation we needed to capture our architecture-independent dynamic metrics. To get access to hardware performance counters we used perfmon2 [10].

To eliminate the perturbation due to our dynamic bytecode instrumentation approach, we collected hardware performance counts in runs without our instrumentation framework. Moreover, to gather all the hardware events we targeted, we had to perform multiple runs, each time counting a different subset of events.

Given these conditions, we had to manually run each interactive application 72 times: we performed 3 equivalent runs for the 3 different types of sessions, and we had to repeat this for collecting the 1 set of platform-independent metrics and the 7 sets of hardware performance metrics (we had to collect 14 hardware metrics but only could collect two metrics per run).

3.4. Analysis approach

In Sections 5, 6, and 7 we characterize the programs using three different kinds of metrics (static design, dynamic platform-independent, and hardware performance) by following the same approach. We first perform hierarchical agglomerative clustering using Ward's linkage method to compute a *dendrogram*. Then we use *principal component analysis* (PCA) to reduce the dimensionality of the metrics space down to two principal components, and we visualize the programs in this two-dimensional space to show their similarities and differences. We support this PCA-based visualization with a *scree plot* to show the importance of the principal components and with a *correlation circle* to highlight the principal components' loadings and the correlations between metrics. Finally, we also show the distributions of selected metrics using *boxplots* to inform our discussion.

4. Code Size

Large programs can put considerable stress on the dynamic class loading, linking, and just-in-time compilation subsystems, and thus exhibit potentially different overall execution characteristics. For this reason we compare the size of the different types of programs in this section.

Table 3 provides an overview of the program sizes. We show the name of the benchmark or interactive application and its size in number of classes. We also show the number and percentage of GUI classes. A GUI class is a normal Java class which either is part of the GUI toolkit, is a subclass of another GUI class, implements an interface that is a GUI class, or is an inner class inside another GUI class. Given the design imposed by the GUI toolkit, GUI classes may significantly differ in their design from other classes in an application. The non-interactive benchmarks do not intentionally make use of GUI classes, as they do not have graphical user interfaces. The last column in the table explicitly lists the classes we considered in our static analyses of each benchmark and application. Selecting the classes that constitute an application is non-trivial, and we decided to include all classes in all JAR files that were required for successfully running the application.

Memon [11] reports that up to 60% of application code can correspond to GUI code. Our data shows that while some specialized applications (CrosswordSage and JHotDraw Draw) can even exceed that fraction, the biggest applications such as NetBeans and Eclipse contain a significantly smaller fraction of GUI classes. The data also shows that even some benchmarks (e.g. sunflow or fop) contain a few classes we categorize as GUI classes. However, they do not use these classes for presenting a user interface.

Figure 3 visualizes the class counts and GUI class counts of Table 3, and it augments this information with the number of classes that were actually loaded during our interactive sessions. Each mark in the graph represents a benchmark or an application. The axes represent the number of classes (x: non-GUI, y: GUI) each program is made of. Due to the strong skew in the distribution of program sizes we use log scales for both axes. The plot clearly shows two significant differences between benchmarks (represented as x and + marks) and interactive applications: (1) benchmarks contain far fewer

	Classes	GUI		JAR files
Runtime Library	19104	4741	25%	charsets, classes, dt, jce, jconsole, jsse, laf, ui, deploy, plugin, sa-jdi, apple_provider, sunjce_provider, sunpkcs11, dnsns, localedata, indicim, thaiim
SPECjvm2008 1.01 20090519				
compiler.compiler	627	0	0%	SPECjvm2008 (spec.benchmarks.compiler.* spec.benchmarks.compiler.compiler.*), javac
compiler.sunflow	627	0	0%	SPECjvm2008 (spec.benchmarks.compiler.* spec.benchmarks.compiler.sunflow.*), javac
compress	13	0	0%	SPECjvm2008 (spec.benchmarks.compress.**)
crypto.aes	2	0	0%	SPECjvm2008 (spec.benchmarks.crypto.* spec.benchmarks.crypto.aes.*)
crypto.rsa	2	0	0%	SPECjvm2008 (spec.benchmarks.crypto.* spec.benchmarks.crypto.rsa.*)
crypto.signverify	2	0	0%	SPECjvm2008 (spec.benchmarks.crypto.* spec.benchmarks.crypto.signverify.*)
derby	1319	0	0%	SPECjvm2008 (spec.benchmarks.derby.**), derby
mpegaudio	73	1	1%	SPECjvm2008 (spec.benchmarks.mpegaudio.**), jl1.0
scimark.fft.small	7	0	0%	SPECjvm2008 (spec.benchmarks.scimark.utils.* spec.benchmarks.scimark.fft.*)
scimark.lu.small	6	0	0%	SPECjvm2008 (spec.benchmarks.scimark.utils.* spec.benchmarks.scimark.lu.*)
scimark.monte_carlo	6	0	0%	SPECjvm2008 (spec.benchmarks.scimark.utils.* spec.benchmarks.scimark.monte_carlo.*)
scimark.sor.small	7	0	0%	SPECjvm2008 (spec.benchmarks.scimark.utils.* spec.benchmarks.scimark.sor.*)
scimark.sparse.small	6	0	0%	SPECjvm2008 (spec.benchmarks.scimark.utils.* spec.benchmarks.scimark.sparse.*)
serial	26	0	0%	SPECjvm2008 (spec.benchmarks.serial.**)
sunflow	650	43	7%	SPECjvm2008 (spec.benchmarks.sunflow.**), sunflow, janino
xml.transform	414	0	0%	SPECjvm2008 (spec.benchmarks.xml.* spec.benchmarks.xml.transform.**), xom-1.1, Tidy
xml.validation	3	0	0%	SPECjvm2008 (spec.benchmarks.xml.* spec.benchmarks.xml.validation.**)
Dacapo 2006-10-MR2 (xdeps)				
antlr	228	5	2%	antlr, antlr-deps
bloat	360	0	0%	bloat, bloat-deps
chart	1246	110	9%	chart, chart-deps
eclipse	5074	52	1%	eclipse, eclipse-deps, including unpacked plug-in jar files
fop	4387	241	5%	fop, fop-deps
hsqldb	577	88	15%	hsqldb, hsqldb-deps
jython	1498	0	0%	jython, jython-deps
luindex	349	0	0%	luindex, luindex-deps
lusearch	349	0	0%	lusearch, lusearch-deps
pmd	2775	79	3%	pmd, pmd-deps
xalan	1827	2	0%	xalan, xalan-deps
AWT/Swing GUI Applications				
ArgoUML	5349	1532	29%	antlr-2.7.7, argouml-mdr, argouml-model, argouml, commons-logging-1.0.2, anarres-cpp-no-dependencies-1.2.3, antlr-2.7.7, antlr-runtime-3.1.1, argo_cpp, argo_idl, argo_java, argo_php, argouml-csharp, argouml-diagrams-sequence, argouml-sql, gef-0.13, java-interfaces, jmi, jmiutils, log4j-1.2.6, mdrapi, mof, nbmdr, ocl-argo-1.1, openide-util, swidgets-0.1.4, toolbar-1.4.1-20071227
CrosswordSage	34	26	76%	CrosswordSage
GanttProject	5288	899	17%	eclipsito, ganttproject, AppleJavaExtensions, commons-httpclient-contrib, commons-httpclient, commons-logging, commons-transaction-1.0, helpgui-1.1, jakarta-poi-2.5, jakarta-slide-webdavlib-2.1, jdnc-modifBen, jdom-1.0, jgoodies-looks-1.2.2, xml-im-exporter1.1, pert, ganttproject-htmlpdf, fop-font-metrics, fop, ganttproject-avalon, ganttproject-batik, ganttproject-msproject, jax-qname, jaxb-api, jaxb-impl, jaxb-libs, mpxj, namespace, relaxngDatatype, xsdlib
jEdit	1150	588	51%	jedit, LatestVersion, MacOS, QuickNotepad
JFreeChart Demo	1667	469	28%	jfreechart-1.0.13-demo, jfreechart-1.0.13, iText-2.1.5, jcommon-1.0.16, jfreechart-1.0.13-experimental
JHotDraw Draw	1146	797	70%	JHotDraw Draw, quaqua, MRJAdapter
NetBeans Java SE	45367	10064	22%	(all 1466 JAR files in the distribution)
SWT GUI Application				
Eclipse SDK Classic	33252	4118	12%	(all 344 JAR files in the distribution)

Table 3. Program size characteristics

classes, and (2) benchmarks contain almost no GUI classes. This difference holds for the static size of the programs as well as the number of classes actually loaded at runtime.

5. Static Design Metrics

In this section we study how interactive Java applications differ in their object-oriented design from Java benchmarks. This study is interesting because differences in design, such as the liberal use of inheritance and dynamic dispatch, can lead to different execution characteristics.

Metrics. Martin [8] introduced a set of design quality metrics for Java. For each of our programs we compute each of Martin's seven metrics shown in **Table 4**. We compute the metric averages across all application packages and exclude the packages of the Java runtime library.

The first two metrics show the average package size in terms of concrete or abstract classes. The next two metrics characterize coupling between packages. *Afferent Coupling* (Ca) shows the responsibility of the package. It is a measure

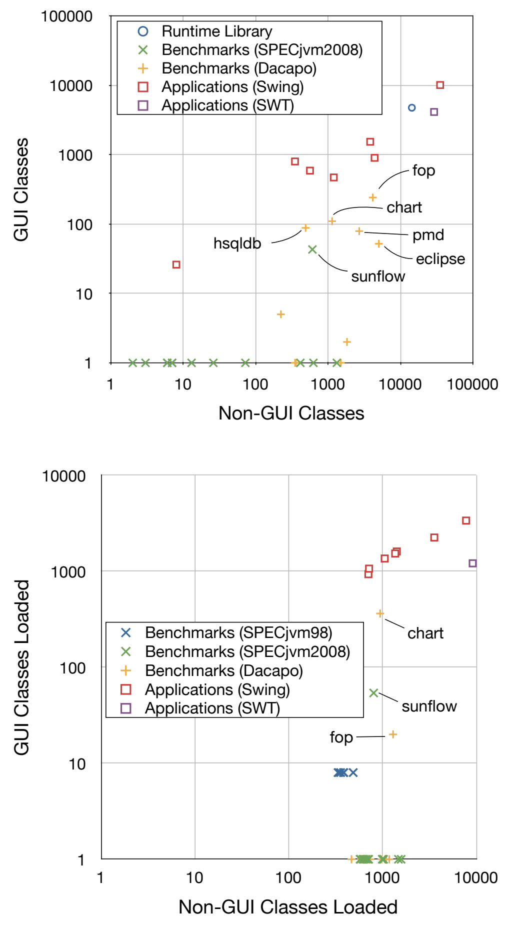

Figure 3. Program size (static or loaded classes)

Name	Description
CC	Concrete Class Count / Package
AC	Abstract Class (and Interface) Count / Package
Ca	Afferent Couplings
Ce	Efferent Couplings
A	Abstractness (0-1)
I	Instability (0-1)
D	Distance from the Main Sequence (0-1)

Table 4. Static design metrics

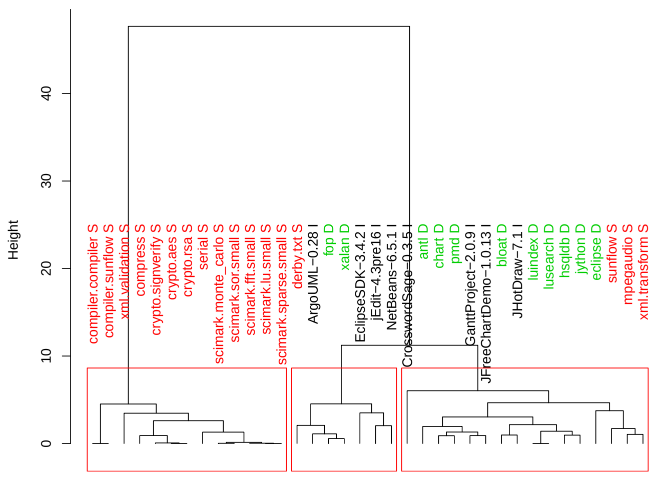

Figure 4. Dendrogram for static design metrics

of how many other classes depend on this package. This metric can be indicative of the chance that a given package is going to be used at runtime. *Efferent Coupling* (Ce) provides a notion of the package dependence. If classes of a package highly depend on other packages, that package may behave as assembling mechanism to glue all the details provided by other packages. *Abstractness* (A) of the packages represents the fraction of abstract classes and interfaces. It is indicative of the level of polymorphism and virtual method dispatch. *Instability* (I) is based on coupling and quantifies the potential for future changes in a package. Finally, *Distance from the Main Sequence* (D) represents a package's balance between abstractness and stability. It is based on the finding that ideally abstract packages are stable, and only concrete packages show instability. This metric measures the deviation from these ideals, producing a value between 0 (good) and 1 (bad).

Clusters. Figure 4 presents a dendrogram that clusters the benchmarks according to the above design metrics. We use hierarchical agglomerative clustering. This bottom-up clustering process starts out by considering each benchmark to be a singleton cluster. It then iteratively combines the two clusters that are most similar into a new cluster. It thus grows a tree bottom-up, by combining small clusters into ever larger clusters, until it ends up with a single cluster containing all benchmarks. Each new cluster is represented as a horizontal line in the dendrogram, connecting the two clusters it combines. The y-coordinate (height) of the connecting line represents the dissimilarity of the two combined clusters: the smaller the y-coordinate of the connection, the more similar the two connected clusters.

The tree structure of the dendrogram in Figure 4 thus shows how interactive applications differ from benchmark suites in terms of design characteristics, and it also visualizes the magnitudes of those differences. We cut the dendrogram at a height that leads to three clusters, and we surround the corresponding clusters with rectangles. We overlay the names of each program over the dendrogram, colored by whether the program is a SPEC (S) or Dacapo (D) benchmark, or an interactive application (I). Most of the SPEC benchmarks are located close to each other, forming the big separate cluster on the left. We believe that the relatively small size and highly specialized aim of these benchmarks lead to non-modular designs that clearly differ from the more object-oriented designs in some of the other programs. The second cluster contains 7 programs, among them 4 of our 8 interactive applications. Eclipse, NetBeans, and ArgoUML are the biggest applications of our suite, and their appearance in the same cluster indicates that they follow similar design principles. The last cluster contains a mixed set of applications belonging to different suites but dominated by Dacapo.

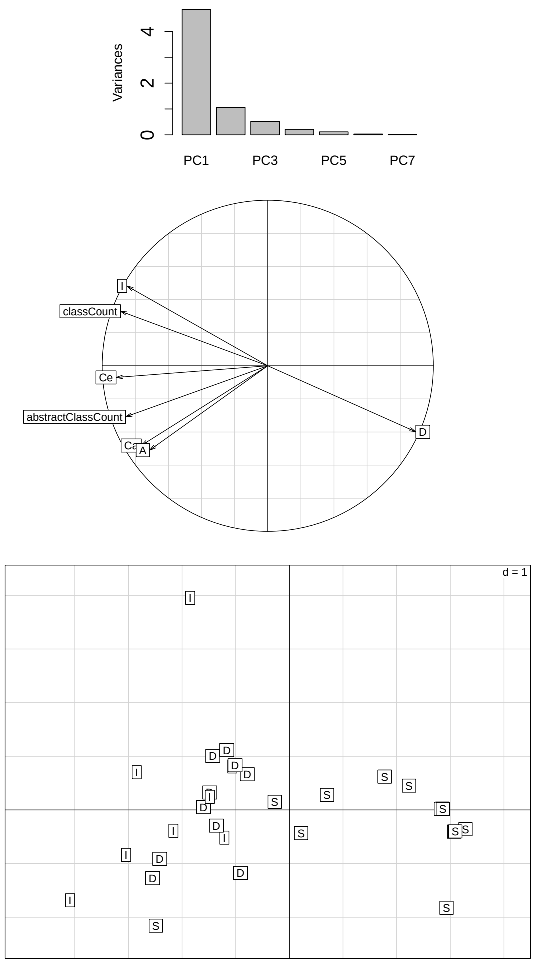

Figure 5. PCA for static design metrics

Principal component analysis. Figure 5 shows the projection of the static metrics onto the top two principal components (PCs). It contains three plots, a scree plot, a correlation circle, and a scatter plot. The *scree plot* shows one bar for each PC, indicating that PC's contribution to the overall variance of the data. The sum of all bars corresponds to 100% of the variance. The plot thus shows that the first component (PC1) absorbs the vast majority of the variance. A PC represents a linear combination of our metrics. The coefficients in that linear combination represent the degree by which each metric affects a given PC. This is called the PC's "loading". We present the loadings of the first two PCs using a *correlation circle*. In that plot the x-axis and y-axis correspond to PC1 and PC2. We project each metric onto this plane and draw it using an arrow. The relative positions of the metrics represent the magnitude of association between metrics: if two metrics are far from the center and close together (such as Ca and A), then they are positively correlated; if they are far from the center but orthogonal (such as A and I), they are uncorrelated, and if they are on the opposite side of the center (such as I and D), they are negatively correlated. If a metric is close to the center, then some information from this metric is carried by other PCs (other than PC1 and PC2). We don't see this case here, because, as the scree plot shows, the first two PCs cover the vast majority of the data's variance. The correlation circle is useful for understanding the factor loading because it visualizes the meaning of the PCs: The coordinate of a metric along an axis corresponds to the correlation of that metric with that PC as measured by Pearson's correlation coefficient (which has an absolute value between 0 and 1). The *scatter plot* shows the programs projected onto the plane defined by PC1 (x-axis) and PC2 (y-axis). It shows that PC1 almost partitions the space into interactive applications (left), Dacapo benchmarks (center), and SPEC benchmarks (right). The loadings of PC1 involve several metrics, with Ce being the one metric that most closely correlates with PC1. Thus, the interactive applications seem to differ from the benchmark suites most strongly in terms of their efferent coupling. Given their location on the left side of the plane, their efferent coupling is higher than that of the benchmarks.

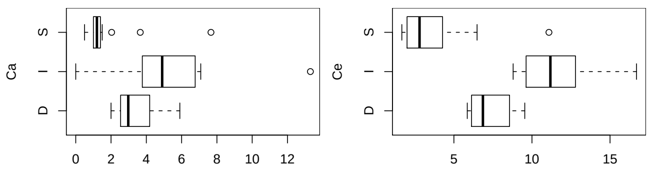

Figure 6. Coupling: applications (I), benchmarks (S,D)

Discussion. The two box plots in **Figure 6** investigate afferent (Ca) and efferent (Ce) coupling in more detail. Each box represents the inter-quartile range (25% of measurements are below the box, 50% inside the box, 25% above the box). The thick line represents the median, and the circles represent the outliers. The figure shows that interactive applications (I) exhibit higher afferent coupling and, particularly, higher efferent coupling. We believe that one reason for this is that programs that are orders of magnitude larger have more classes that a class can depend on (resp. that can depend on a class).

6. Dynamic Platform-Independent Metrics

Dynamic metrics, such as the ones introduced by Dufour et al. [12], characterize the behavior of applications in a platform-independent way. Unfortunately, those metrics are expensive to collect: their collection slows down program execution by orders of magnitude, preventing the realistic use of interactive applications. For this reason we have developed a set of dynamic metrics that we can collect with low overhead instrumentation[2]. In this section we study whether

2. For SPEC JVM 2008 compiler.compiler, we experienced a slowdown of 1800x with Dufour's *J, but only 35x with our approach.

and how interactive applications differ from Java benchmarks along the characteristics covered by these metrics.

Metrics. Table 5 describes our dynamic metrics. The top section describes the basic events we count. We do not use these absolute counts for characterization, because they depend on the duration of the program run. Instead, we define relative metrics based on these counts. The *Conditionals vs. loops* group characterizes control flow. *Calls vs. loops* distinguishes between recursive and iterative code. *Calls vs. branches* compares the amount of inter-procedural and intra-procedural control flow. It distinguishes between object-oriented code (using dynamic dispatch) and procedural code (using conditionals). *Array vs. field access* distinguishes between array-based and object-based code. *Array vs. field vs. static access* extends this to code based on global variables. *Object vs. array allocations* provides a different view on the prevalence of arrays. *Invocation types* breaks down calls based on their cost: interface method invocations require the most expensive implementation, followed by virtual calls, special (non-virtual) calls, and ultimately statically dispatched calls. *Encapsulation vs. direct field access* distinguishes between code that uses accessor methods and code that accesses public fields. *Static field access vs. instance field access* is another way to distinguish between procedural and object-oriented programs.

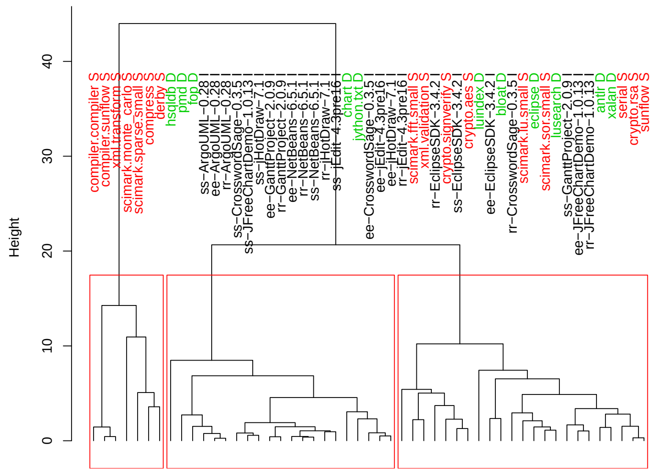

Figure 7. Dendrogram for dynamic platform-independent metrics

Clusters. For clustering and PCA we exclude all redundant metrics and only keep N-1 of the N metrics in each group (metrics marked with **X** in Table 5). **Figure 7** shows the dendrogram based on our dynamic metrics. The first cluster consists of 7 SPEC benchmarks. The second cluster is dominated by interactive applications. Note, that almost all start-stop (SS) sessions fit into this category, except for Eclipse, which is based on a different GUI toolkit and for GantProject. The third cluster again contains a mixture of different program types.

Use	Name	Definition
Basic metrics:		
	FwdBr	forward branch
	BackBr	backward branch
	Branches	BackBr+FwdBr
	Invokes	Static+Special+Virtual+Interface
	MethCall	Interface+Virtual
	ArrayAcc	*LOAD+*STORE+AALOAD+AASTORE
	FieldAcc	GETFIELD+PUTFIELD
	StaticAcc	GETSTATIC+PUTSTATIC
	ArrayAlloc	*NEWARRAY
	ObjectAlloc	NEW
Conditionals vs. Loops:		
X	FwdBr%	FwdBr/(FwdBr+BackBr)
	BackBr%	BackBr/(FwdBr+BackBr)
Calls vs. Loops:		
X	Invokes%	Invokes/(Invokes+BackBr)
	BackBr%	BackBr/(Invokes+BackBr)
Calls vs. Branches:		
X	Invokes%	Invokes/(Invokes+Branches)
	Branches%	Branches/(Invokes+Branches)
Array vs. Field access:		
X	ArrayAcc%	ArrayAcc/(ArrayAcc+FieldAcc)
	FieldAcc%	FieldAcc/(ArrayAcc+FieldAcc)
Array vs. Field vs. Static access:		
X	StaticAcc%	StaticAcc/(ArrayAcc+FieldAcc+StaticAcc)
X	ArrayAcc%	ArrayAcc/(ArrayAcc+FieldAcc+StaticAcc)
	FieldAcc%	FieldAcc/(ArrayAcc+FieldAcc+StaticAcc)
Object vs. Array allocations:		
X	ObjectAlloc%	ObjectAlloc/(ObjectAlloc+ArrayAlloc)
	ArrayAlloc%	ArrayAlloc/(ObjectAlloc+ArrayAlloc)
Invocation types:		
X	Static%	Static/Invokes
X	Special%	Special/Invokes
X	Virtual%	Virtual/Invokes
	Interface%	Interface/Invokes
Encapsulation vs. direct field access:		
X	FieldAcc%	FieldAcc/(FieldAcc+MethCall)
	MethCall%	MethCall/(FieldAcc+MethCall)
Static field access vs. instance field access:		
X	FieldAcc%	FieldAcc/(FieldAcc+StaticAcc)
	StaticAcc%	StaticAcc/(FieldAcc+StaticAcc)

Table 5. Dynamic platform-independent metrics

Principal component analysis. Figure 8 shows the principal component analysis. The scree plot shows that, unlike for the static design metrics, PC2 and even PC3 still cover a major part of the overall variance. This is a result of the richer dynamic metric space. We can also see this in the correlation circle, where the arrows of several metrics (most notably "Obj.", the fraction of objects in all object and array allocations) are relatively short (and thus the variance in these metrics is not fully covered by PC1 and PC2). The correlation circle shows that PC1 is negatively correlated with forward branches and positively correlated with array accesses and method invocations. The benchmarks to the right, with a high fraction of array accesses and a relatively low number of forward branches and method invocations, all are SPEC benchmarks. PC2's loading is dominated by the fraction of field accesses in all field and static accesses. Interactive applications are mostly located in the lower left quadrant, and thus have (besides fewer array accesses, more forward branches, and more method invocations), to our surprise, a

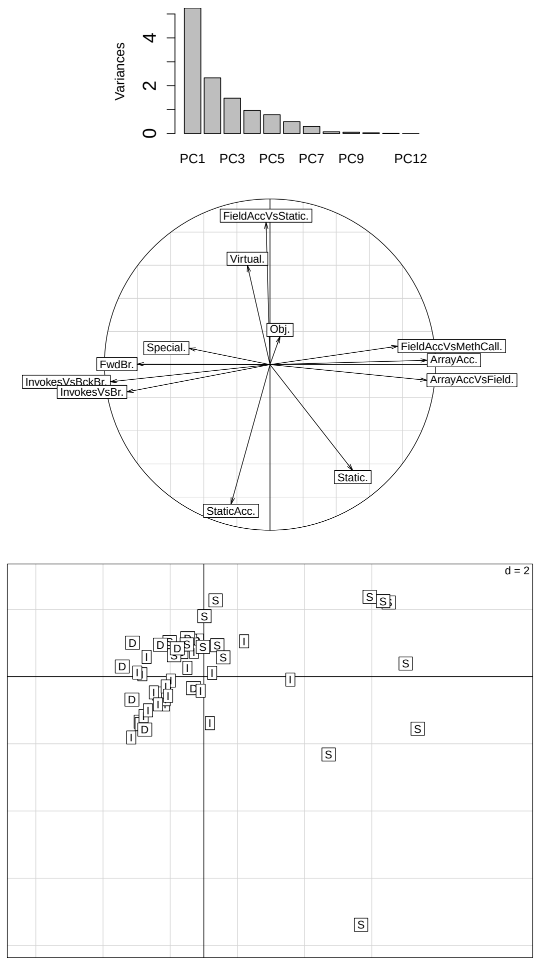

Figure 8. PCA for dynamic platform-independent metrics

higher fraction of *static* field accesses.

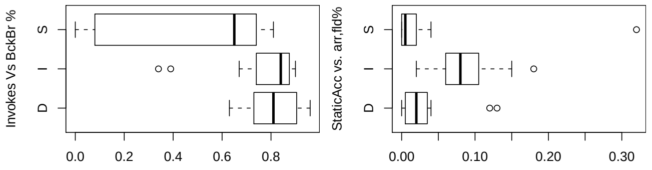

Figure 9. Dynamic metrics: applications (I) vs. benchmarks (S,D)

Discussion. The left side of **Figure 9** shows that Dacapo benchmarks (D) are similar to interactive applications (I) with respect to method invocations vs. backward branches: in both cases, the fraction of method invocations is higher than it is for the loop-oriented SPEC benchmarks (S). The right side of Figure 9 confirms the surprising finding that static accesses are more prevalent in interactive applications than in the benchmarks. We believe that this could be due to the abundance of constants that are used in communicating between the Java GUI toolkit and the (non-object-oriented) native GUI library.

7. Hardware Performance Metrics

While platform-independent dynamic metrics and static design metrics characterize the coding style and design of programs, hardware performance counters concentrate on the performance aspects of the execution on a real system.

Metrics. We focus our selection of metrics on central aspects of single-threaded performance, because most interactive applications are dominated by the single GUI thread that handles user events and renders to the screen. **Table 6** describes our selection of hardware metrics, including the L2 cache, DTLB, and ITLB miss ratios as well as branch missprediction ratios (overall, for conditional, and for indirect branches).

Name	Description
L2-miss%	L2_LINES_IN/L2_ST+L2_LD+L2_IFETCH
DTLB-miss%	DTLB_MISSES/L1D_ALLREF
ITLB-miss%	ITLB/L1I_READS
BR_MISSP%	BR_MISSP_EXEC/BR_INST_EXEC
BR_CND_MISSP%	BR_CND_MISSP_EXEC/BR_CND_EXEC
BR_IND_MISSP%	BR_IND_MISSP_EXEC/BR_IND_EXEC

Table 6. Hardware performance metrics

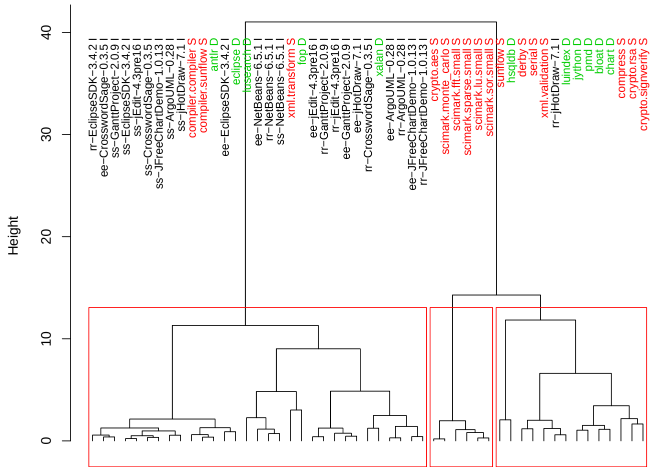

Figure 10. Dendrogram for hardware metrics

Clusters. The dendrogram in **Figure 10** shows three big clusters. The first one is dominated by interactive applications. The second one is dominated by SPEC benchmarks, and the third contains mostly SPEC and Dacapo benchmarks. The location of the cluster connections (height along the y-axis) shows that the first cluster (interactive applications) is significantly different from the second and third clusters (benchmarks), and that the second and third clusters are quite similar.

Principal component analysis. The scree plot and correlation circle in **Figure 11** show that PC1 and PC2 cover a significant fraction of the overall hardware performance variability, with the notable exception of the L2 cache misses (short arrow). The scatter plot shows most interactive applications in the lower left quadrant, which the correlation circle

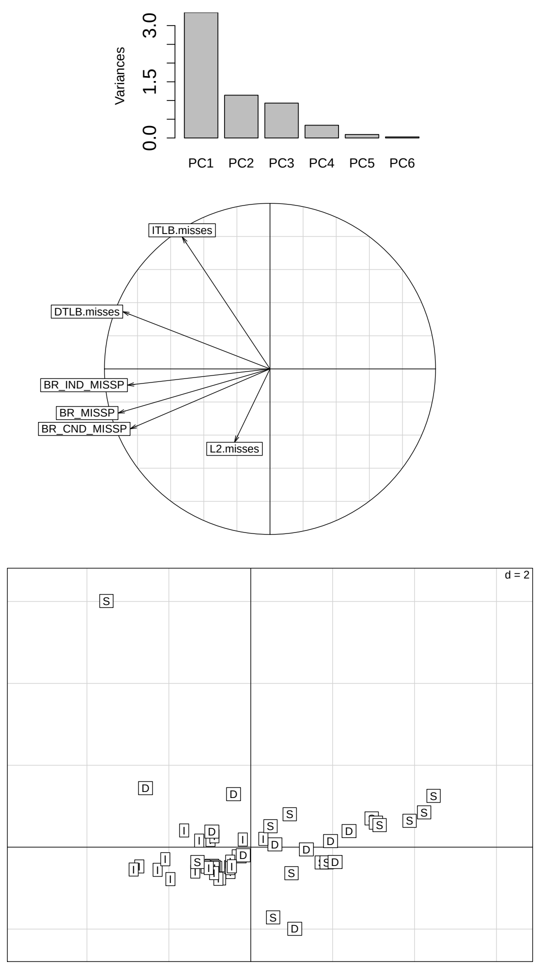

Figure 11. PCA for hardware performance metrics

characterizes as an area with a high branch missprediction rate.

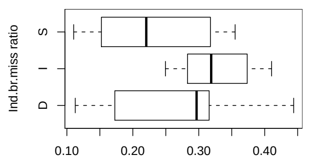

Figure 12. Indirect branch mispredictions: applications (I) vs. benchmarks (S,D)

Discussion. A closer look at indirect branch mispredictions in **Figure 12** shows that interactive applications (I) indeed have a higher fraction of misses than benchmarks (D and S). Indirect branch mispredictions can be the result of polymorphism: virtual method calls are often compiled down to indirect calls. This correlates with our intuition that interactive applications make more use of polymorphism due to the design imposed by the GUI frameworks they are based upon.

8. Clustering based on all Metrics

To see the full picture we performed a clustering based on the combination of the static design metrics, dynamic platform-independent metrics, and hardware performance metrics. **Figure 13** shows that indeed most of the programs are clustered according to their type: the SPEC benchmarks occupy two of the three clusters, and the large remaining cluster consists of several subclusters. One isolated Dacapo benchmark (lusearch) lies in the middle of a large interactive cluster, and the smallest interactive application (CrosswordSage) is located in a Dacapo cluster. Except for those two cases, the interactive applications are tightly clustered together.

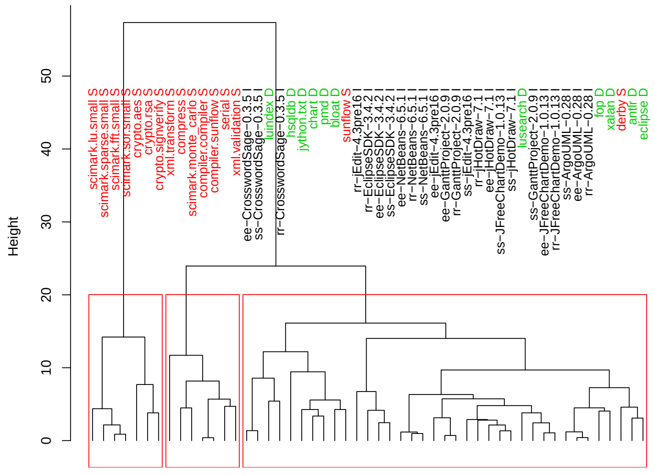

Figure 13. All metric characterization

9. Related Work

Blackburn et al. [2] put a great effort into the creation of the Dacapo benchmark suite, with the goal of providing a more realistic suite of client-side benchmarks. Their goal was to create a diverse suite of real world applications. They show that their new benchmark suite significantly differs in many respects from the SPEC Java benchmarks commonly used at that time. They explicitly exclude GUI applications because of the complexities involved in turning them into benchmarks. In this paper we show that Java GUI applications significantly differ from both the Dacapo benchmark suite as well as the most recent version of the SPEC JVM suite. For showing this, we had to devise dynamic metrics that can be collected while a user interacts realistically with a GUI application. Moreover, unlike for non-interactive benchmarks, we had to manually repeat equivalent realistic interactive sessions with each application we studied, in order to collect all the required hardware and software performance metrics.

Dufour et al. [12] propose dynamic metrics for characterizing the behavior of Java applications in a platform-independent way. Their metrics provide a relevant characterization of Java program behavior. Unfortunately, they are too expensive to collect for interactive applications: Metrics collection slows down program execution by several orders of magnitude, precluding any reasonable interactions between user and application. For this reason we designed dynamic metrics for our study such that we can collect them at speeds that still allow meaningful user interaction.

Eeckhout et al. [13] characterize the interactions between the Java virtual machine, the benchmark application, and the application's inputs. Their characterization is based on hardware performance metrics. Their focus is less on characterizing the difference between benchmarks, and more on determining to what degree performance is affected by the virtual machine, the benchmark, or the input size. The use of principal component analysis and hierarchical agglomerative clustering on the collected hardware performance metrics allows them to draw conclusions useful for experimental performance studies with virtual machines. Unlike their work, the goal of our paper is to study whether real-world interactive applications have different characteristics from existing benchmarks, and we use dynamic and static platform-independent metrics in addition to the microarchitectural metrics provided by hardware performance counters.

10. Conclusions

In this paper we show that the client-side benchmarks used for evaluating new research ideas are not necessarily representative of interactive Java applications. We show this by comparing the benchmarks to widely used real-world client-side Java applications. Our comparison shows that the benchmarks differ from the interactive applications with respect to static as well as dynamic characteristics, including platform-independent as well hardware-dependent performance metrics. We found that interactive applications exhibit higher indirect branch misprediction rates, probably caused by a larger amount of polymorphic method calls. We were surprised to find that interactive applications exhibit higher coupling and a larger fraction of static field accesses, two characteristics that we did not expect from applications we a presumably more object-oriented design.

Our results indicate that future versions of Java benchmark suites either should consider including automated versions of complete interactive applications, or that they should extract the frequently used parts of such applications (similar to the approach of the Dacapo eclipse benchmark), and turn them into automatically executable benchmarks. At the very least, performance experiments using client-side benchmarks should stress the importance of those benchmarks that are similar to the dominant cluster of interactive applications.

While turning interactive applications into benchmarks would require a significant effort, we believe that this effort is outweighed by the resulting benefits. After all, a meaningful evaluation of new features or optimizations in runtime systems, compilers, or computer architecture critically depends on benchmarks that are representative of the applications used in the real world.

References

[1] SPEC, "SPECjvm2008 (Java Virtual Machine Benchmark)," http://www.spec.org/jvm2008/.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 169–190.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] K. Walrath, M. Campione, A. Huml, and S. Zakhour, *The JFC Swing Tutorial: A Guide to Constructing GUIs, 2nd Edition*. Addison-Wesley, 2004.

[5] S. Northover and M. Wilson, *SWT: The Standard Widget Toolkit, Volume 1*. Addison-Wesley, 2004.

[6] J. McAffer and J.-M. Lemieux, *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java Applications*. Addison-Wesley, 2005.

[7] I. Skerrett, "One million Galileo downloads," http://eclipse.dzone.com/articles/one-million-galileo-downloads, July 2009.

[8] R. Martin, "Object oriented design quality metrics: An analysis of dependencies," http://clarkware.com/software/JDepend.html, 1994.

[9] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems*, November 2002.

[10] S. Eranian, "perfmon2: A flexible performance monitoring interface for linux," in *Proceedings of the Linux Symposium*, July 2006, pp. 269–288.

[11] A. M. Memon, "Gui testing: Pitfalls and process," *Computer*, vol. 35, no. 8, pp. 87–88, 2002.

[12] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 149–168, Nov. 2003.

[13] L. Eeckhout, A. Georges, and K. D. Bosschere, "How Java programs interact with virtual machines at the microarchitectural level," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003, pp. 169–186.