On the Representativeness of Embedded Java Benchmarks

Ciji Isen and Lizy John
Electrical and Computer Engineering Department
University of Texas at Austin
{isen, ljohn}@ece.utexas.edu

Jung Pil Choi and Hyo Jung Song
Samsung Electronics
{jungpil.choi, hjsong}@samsung.com

***Abstract*— Java has become one of the predominant languages for embedded and mobile platforms due to its architecturally neutral design, portability, and security. But Java execution in the embedded world encompasses Java Virtual Machines (JVMs) specially tuned for the embedded world, with stripped-down capabilities, and configurations for memory-limited environments. While there have been some studies on desktop and server Java, there have been very few studies on embedded Java. The non proliferation of embedded Java benchmarks and the lack of widespread profiling tools and simulators have only exacerbated the problem. While the industry uses some benchmarks such as **MorphMark, MIDPMark,** and **EEMBC Java Grinder Bench,** their representativeness in comparison to actual embedded Java applications has not been studied. In order to conduct such a study, we gathered an **actual mobile phone application suite** and characterized it in detail. We measure several properties of the various applications and benchmarks, perform similarity/dissimilarity analysis and shed light on the representativeness of current industry standard embedded benchmarks against actual mobile Java applications. It was observed that for many characteristics, the applications had a broader range, indicating that the benchmarks were under representing the range of characteristics in the real world. Furthermore, we find that the applications exhibit less code reuse/hotness compared to the benchmarks. We also draw comparisons of the embedded benchmarks against popular desktop/client Java benchmarks, such as the SPECjvm98 and DaCapo. Interestingly, embedded applications spend a significant amount of time in standard library code, on average 65%, suggesting to the usefulness of software and hardware techniques to facilitate pre-compilation with out the real time resource overhead of JIT.**

I. INTRODUCTION

Devices such as MP3 players, PDAs, and cellular phones have increased in popularity over the past few years, and this is only expected to grow. According to Rob Shaddock, chief technology officer for Motorola mobile devices, the dramatic size of the cell phone market, with its 2 billion subscribers "makes it the largest consumer electronics business on the planet, bar none" [16]. A 2006 Gartner study expects mobile application deployment by enterprises to grow by 30% per year till 2011 with over 8% of the IT budgets spent on mobile applications.

As mobile devices grow in complexity and time-to-market windows shrink, high-level languages have been increasingly adopted to enhance productivity. Issues relating to development time, platform security and portability, as well as the traditional concerns of performance and cost have been pushed to the forefront. More importantly, the rise of malicious software has increased the need for ensuring the security of executable code. Java [25] has become one of the predominant development languages for mobile systems due to its architecturally neutral design, rich Application Program Interfaces (APIs) and ability to ensure executable security. Java support on embedded platforms has experienced significant development and growth from major industry groups in recent years. According to a Gartner study, the sale of Java enabled cell phones is expected to be nearly 789 million units in 2007, and 1.17 billion in 2010 [14].

There are many reasons for the popularity of Java despite the challenges to run Java in a resource-restricted embedded environment. The embedded space is populated by a very large variety of architectures (ARM, MIPS, x86, PowerPC etc) and operating systems(flavors of Linux, Windows CE, Palm etc) making portability a key issue. This coupled with the short life span of devices and rapid pace of evolution of applications exacerbates the need to be able to churn out high quality, portable code rapidly. A 2006 Gartner study [14] expects that, 50% of the large corporations will employ at least five mobile architectures and a third of current application will be discarded by 2009. Under such circumstances, the managed runtime based model of Java coupled with a virtual machine environment provides considerable advantages to an embedded software developer including code portability and ease of development. The presence of garbage collection in the language runtime, frees the programmers from the burden of memory management, by avoiding memory leaks and accidental memory overwrites ("dangling pointers") [21]. Additionally, Java is designed to deploy code dynamically over a network in a secure fashion. Furthermore, the bytecode structure of Java also ensures that the class files are compact compared to a full native binary, allowing for optimal use of limited network bandwidth.

Figure 1 illustrates the various layers in the typical embedded Java platform. Embedded Java environments consist of two important elements – *configurations* and *profiles*. Configurations provide a set of libraries and a virtual machine. Profiles are APIs built on top of configurations to provide a runtime environment for a specific device such as PDA or cell phone. Profile manages the application, user interface, networking and I/O. In our experiments, the configuration is CLDC, and the profile is MIDP.

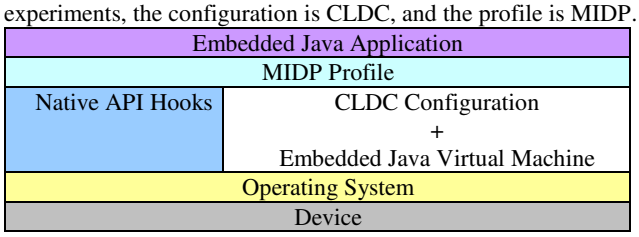


Fig 1. Typical Embedded Java Platform

CLDC (Connected Limited Device Configuration) [29] includes some new classes designed specifically to fit the needs of small-footprint devices with an intermittent (limited) network connection. It specifies a stripped-down JVM (e.g.: KVM) as well as several APIs for fundamental application services. Short for Mobile Information Device Profile, **MIDP** is a set of embedded Java APIs that define how software applications interface with cellular phones and two-way pagers.

Embedded Java has become important and critical; however embedded Java benchmarking is yet to mature. A notable effort comes from the Embedded Microprocessor Benchmark Consortium (EEMBC), a consortium of 58 member companies, who have interest in a variety of areas like microprocessors (MPU's) microcontrollers (MCU's), and digital signal processors (DSP's). To address the emergence of Java as a language of choice, in 2002, EEMBC released a Java benchmark suite, the

EEMBC Java Grinderbench [12]. EEMBC Grinderbench contains kernels extracted from some real world applications built on the CLDC framework. There are also some other benchmarks available like **Morphmark** [22], **MIDPMark** [7] and **CaffeineMark** [23]. These stress some aspects of the embedded Java framework on the target test platform and are mostly synthetic. For this reason, some of the benchmarks like CaffeineMark have been criticized to be the LLLs of embedded Java benchmarks. We use all of these benchmarks in our study.

Given a set of benchmarks, it would be immensely useful to system designers to know, if the benchmarks provide adequate representation of their application space. Such concerns are understandably acute in emerging areas like embedded Java. Although suites such as Morphmark, MIDPmark and Caffeinmark exist, nothing is known about their representativeness. Furthermore, the loop nature of many of these benchmarks has been of concern to designers and many often doubt their representativeness. We undertake a study examining the embedded benchmarks in the light of a suite of embedded applications from a leading cell phone manufacturer, Samsung Electronics. The mobile application suite consists of 50+ real world applications, deployed in various models of mobile phones and devices. These applications include interactive graphical games, board games, image rendering, web browsing, photo gallery, and navigation systems (maps).

There have been very few studies on embedded Java in the past. Desktop Java has been understood a little more and hence we also compare and contrast embedded Java with desktop Java.

Table 1. Embedded Java Benchmarks Used in this Study

Benchmark	*Source*
EmbeddedCaffeineMark 3.0	Pendragon Software
MIDPMark	Digital Bridges
Morpmark	Morpheme
GrinderBench	EEMBC

Specifically, we address the following questions:

a) Are popular embedded benchmarks representative of the real world embedded applications?

A primary concern in benchmarking is whether the benchmarks are representative of actual applications. Embedded benchmarks are traditionally miniature compared to their desktop counterparts. They are loop intensive and often are created by stripping out a lot of I/O functionality from applications. Many are often concerned about the representativensss and usefulness of these miniature embedded benchmarks. We study a cellular phone application suite consisting of 50+ applications in use within an actual cellular phone, and analyze the similarity between these real world embedded applications and embedded Java benchmarks to understand the representativeness of the benchmarks. We use Principal Component Analysis and clustering techniques [9, 10, 11] to study similarity.

b) Are embedded Java benchmarks fundamentally different from client/desktop Java benchmarks?

Most embedded benchmarks are traditionally miniature. Does the small size mean inherent reduction of complexity compared to the desktop benchmarks? How different are embedded Java benchmarks from desktop Java benchmarks? We compare the embedded Java benchmarks against desktop benchmarks like SPECjvm98 [26], and a recently proposed desktop suite, the DaCapo benchmark [3].

II. Embedded benchmarks and applications

This section describes the embedded benchmarks and applications that we used.

A. Embedded CaffeineMark (CaffMark)

The EmbeddedCaffeineMark benchmark suite from PenDragon Software [23] is a simple synthetic benchmark suite composed of several key Java language features. As illustrated in Table 2, the EmbeddedCaffeineMark suite is composed of a variety of tests involving kernels targeting basic language features such as method invocations, object manipulation, and basic logical, string and mathematical operations.

Table 2. EMBEDDED CAFFEINEMARK TESTS

TestName	*Description*
Sieve	prime number 'sieve'
Loop	Fibonacci 'loop'
Logic	Boolean 'logic' tests
String	'String' concatenation and conversion
Float	'float' operations (add,sub,mul,div)
Method	Repeated method invocation (non-recursive)

B. MIDPMark

The MIDPMark benchmark suite by DigitalBridges [7], is a suite that tests the different media features of a mobile handset. A set of non-interactive and non-graphical MIDPMark tests as described in Table 3 were used in the study. These tests include method and thread creation, integer math and synchronization calls.

Table 3. MIDPMark Non-Graphical Tests

TestName	*Description*
Creation	New object creation
Logic	Integer Math
Method	Method Invocation (private, public, static, final, etc)
Synchro	Synchronized method calls (use of monitor)
Thread	Thread creation

C. MorphMark

MorphMark, by Morpheme Ltd [22], is another common benchmark suite that tests the performance of applications like video games. The benchmark is primarily designed to test the overall performance of embedded Java Virtual Machines when running an entertainment gaming application. However, the benchmark suite also has tests targeting the virtual machine performance itself without interacting with the graphics subsystem. The set of tests that were used in our study are listed in Table 4.

Table 4. MorphMark Non-Graphical Tests

TestName	*Description*
Forward Loop	Method loop with a incrementing counter
Reverse Loop	Method loop with a decrementing counter
Integer Math	Simple arithmetic tests
System Array Copy	Array copy using the System.ArrayCopy functionality
User Array Copy	Array copy using user level copying

D. EEMBC Java GrinderBench

EEMBC, the industry embedded benchmarking consortium,

provides a set of Java benchmarks known as Java GrinderBench[12]. These are composed of a set of six embedded Java applications. The applications are non-interactive, non-graphical and only require the support of the CLDC libraries. All the six applications were used in the study and are described in more detail in Table 5.

Table 5. EEMBC Java GrinderBench Applications

TestName	*Description*
Chess	Chess playing solver (3 games, 10 moves)
Crypto	Encrypts/Decrypts a small text document with a set of crypto algorithms (DES, IDEA, Blowfish, Twofish)
XML	Parsing and manipulation of a small XML document
Parallel	Mergesort, Matrix multiply using multiple threads for execution
PNG	Decodes a PNG graphic image (doesn't use graphical display, just the decoding only)
RegEx	Parses a file using regular expressions

E. Mobile phone Java Applications

We obtained access to a suite of embedded Java applications for a mobile phone. The suite consists of approximately 50 different applications. These applications are from various applications domains like interactive graphical games, board games, image rendering, web browsing, photo gallery, and navigation systems (maps).

Table 6. Summary of Dynamic Bytecode Count for Studied Benchmarks

Benchmark Suite	*Dynamic Bytecodes (Million of Bytecodes)*	
Embedded Caffeine Mark	50	
MIDPMark	70.9	
Morphmark	4.8	
EEMBC Java Grinder	Chess	9.9
	Crypto	24.7
	XML	10.03
	Parallel	155.4
	PNG	7
	RegEx	5.7

In order to give an indication of the length of the benchmarks, we provide the dynamic size of the various benchmarks used in this study. Table 6 shows the number of bytecodes executed for each given benchmark. As seen in Table 6, the individual tests and suites in the study have varying dynamic bytecode counts, ranging from 5 to 155 million bytecodes. In contrast, the SPECjvm98 benchmarks execute 2 to 954 million bytecodes with the –s1 data set and approximately 100 million to 1 billion bytecodes with the –s100 data set. Thus the embedded benchmarks are miniature compared with the desktop Java benchmarks. How representative are these embedded benchmarks? That is the question we set off to answer.

III. Methodology

Characterizing embedded Java and comparing it to desktop Java is very challenging from the perspective of tools and methodology. Embedded and desktop VMs are different and the stripped down embedded VMs will not run the desktop applications. However, we use a methodology that consists of analyzing object-oriented metrics in the various programs at a VM-independent level. We use the **IBM J9 VM** [15], and a **Sprint Wireless Tool Kit** [28] in order to perform the various experiments. Embedded VMs are stripped down to be resource efficient which in turn makes them deficient in some of the debugging and profiling abilities. Many of the embedded VMs have no support for profiling interfaces like **JVMPI** [25] or **JVMTI** [25]**.** We used the IBM J9 VM, which supports JVMPI, enabling us to measuring several of the VM level metrics. Real embedded Java applications often use some of the device specific characters adding to the dependencies and complexity of execution. Some of these dependencies seem to interfere with the limited JVMPI support IBM J9 has and had to be dealt at a case by case basis for the suite of 50+ applications. By working around these limitations, we perform a study of embedded Java benchmarks at both the bytecode and the native embedded processor level.

A. Simulation/Execution framework

The *embedded virtual machine* used in this study is an IBM J9. The IBM J9 VM, is a high-end, feature-heavy, versatile VM implementation for embedded Java. It supports JVMPI enabling profiling.

B. Principal Component Analysis and Clustering

In order to understand the similarity/dissimilarity between embedded Java benchmarks, other Java benchmarks and actual embedded Java applications, we use Principal Component Analysis (PCA) and clustering [10, 11]. PCA is a multivariate statistical technique that reduces a large N-dimensional space into a lower dimensional uncorrelated space with very little loss of information. In order to isolate the effect of varying ranges of each parameter, the data is first normalized to a unit normal distribution, *i.e.* a normal distribution with mean equal to zero and standard deviation equal to 1, for each variable. PCA helps to reduce the dimensionality of a data set while retaining most of the original information. PCA computes new variables, so-called principal components, which are linear combinations of the original variables, such that all the principal components are uncorrelated. PCA transforms p variables X_1, X_2,...., X_p into p principal components (PC) $Z_1,Z_2,\ldots,Z_p$ such that:

$$Z_i = \sum_{j=0}^{p} a_{ij} X_j$$

This transformation has the property $Var\ [Z_1] \geq \text{Var}\ [Z_2] \geq \ldots \geq \text{Var}\ [Z_p]$ which means that Z_1 contains the most information and Z_p the least. Given this property of decreasing variance of the PCs, we can remove the components with the lower values of variance from the analysis. This reduces the dimensionality of the data set while controlling the amount of information that is lost. We use a standard technique (Kaiser Criterion) to choose PCs where only the top few PCs which have eigenvalues greater than or equal to one are retained. For details on PCA please refer to [9]. After PCA, the workload space is projected using the most important principal components, or linkage distance between the programs is computed.

IV. Measuring Representativeness of Embedded Benchmarks

We study characteristics of the aforementioned embedded Java benchmarks and applications. Some issues with the instrumentation and profiling setup prevented a few of the applications from functioning correctly; but still we were able to correctly execute 40+ real world applications. We first compare the benchmarks and applications for **code complexity**, code structure, object-orientedness features, class hierarchies, etc using the

popular **Chidamber and Kemerer (C-K) metrics** [4]. Then the applications and benchmarks are compared for **object behavior** as measured by execution-time object allocation/deallocation and liveness characteristics using methodology used by Diekman-Holzle, and Blackburn et al [8, 3].

A. Code Complexity Metrics

The complexity of the code is one of the aspects to compare embedded benchmarks and real world mobile phone application programs. Chidamber and Kemerrer [4] proposed several object oriented programming metrics in order to quantify code complexity. These metrics include Depth of Inheritance tree, number of children, coupling between classes, etc. We use the software package **ckjm** [27] to measure these metrics. As in prior work [3], these metrics are measured for classes that load at runtime. The libraries are excluded from the analysis as they are heavily duplicated across the benchmarks. These metrics are described in short as follows.

WMC (Weighted Methods per Class): WMC for a given program is measured by adding *complexity* of a program's methods. Ckjm assigns a complexity value of 1 to each method, and therefore the WMC value is equal to the number of declared methods in the loaded classes. Large numbers thus show that a class provides a variety of different behaviors in the form of different methods/functions.
DIT (Depth of Inheritance Tree): DIT provides for each class a measure of the inheritance levels from the top of the object hierarchy. In Java where all classes inherit object the minimum value of DIT is 1.
NOC (Number of Children): NOC measures the number of immediate subclasses of the class.
CBO (Coupling Between Objects): For a given class CBO measures the number of classes coupled to a given class. Classes may be coupled via method calls, field accesses, inheritance, arguments, return types, and exceptions. The metric measures code complexity in terms of interactions between objects and classes.
RFC (Response for a Class): RFC measures the number of different methods that may execute when a method is invoked. Ckjm calculates a rough approximation to the response set by inspecting method calls within the class's method bodies.
LCOM (Lack of Cohesion): LOC counts methods in a class that are not related through the sharing of some of the class's fields.

Table 7 presents the C-K metrics for embedded benchmarks and the mobile phone application suite. The benchmark data includes EEMBC, Morphmark, MIDPmark, and Caffeine benchmarks. The applications data is aggregated from 40+ different applications.

Weighted Methods per class ranges from 37 to 191 for the benchmarks, while it ranges from 2 to 410 for the applications. Similarly the depth of inheritance tree ranges from 7 to 49 for benchmarks while it ranges 0 to 68 for the applications. However, the average depth of inheritance tree is 25 for the benchmarks while only 16 for the applications. The number of children range from 0 to 16 for benchmarks and 0 to 22 for applications. In general, the applications exhibit a wider spread than the benchmarks. For all 6 of the metrics, the application has a maximum much higher than the maximum of the benchmarks. Similarly, the minimum for the applications is smaller than the minimum for the benchmarks for 5 out of 6 metrics. However, if you look at the mean or median, for many of the metrics, the benchmarks are slightly more complex than the suite of applications we studied. Although this might suggest that the benchmarks are slightly over designed, it is not unhealthy to have benchmarks slightly overstress application characteristics since benchmarks are made to last longer and also intended to help in studying ruggedness of designs. The more serious weakness is that the benchmark suite lacks the broad spread of the application suite.

In order to compare and visualize the similarity/dissimilarity of the workload space (with all metrics), we rely on Principal Components Analysis (PCA) [9]. Figure 2 shows a scatter plot of two principal components that account for about 86% of the variance in the embedded benchmarks and the embedded applications. From Figure 2 we can observe that the benchmarks are within the typical range exhibited by the applications for the code complexity characteristics. But there are a few real world applications (e.g.: A18, A30, A49, A50) that are very different from all the benchmarks and hence are not represented by the benchmarks. Two of the applications, A49 and A50 have high values for most of the C-k metrics. (Our non-disclosure agreement with Samsung does not allow us to discuss the details of these applications any more.) However, it can be easily concluded that there is room for improvement in the design of the benchmarks. Benchmark suites can be made to include more diverse benchmarks. Although the benchmark suite does not show complete coverage of the application space, it is encouraging to note that the industry standard benchmarks do fall within the range exhibited by the applications.

Another interesting aspect is the comparison of the complexity of embedded benchmarks to SPECjvm98 and the recently developed DaCapo benchmark [3]. DaCapo was an effort to create a benchmark suite that is heavily object-oriented. We compared the C-k metrics of SPECjvm98 and DaCapo benchmarks to embedded programs and observe that DaCapo is far richer and complex in object-orientedness than any other program we studied. Figure 3 presents the principal component analysis with embedded applications, embedded benchmarks, SPECjvm98 and DaCapo. The complexity and richness of DaCapo overshadows the rest of the data. DaCapo_eclipse is far out to the left and is not shown in the figure. The embedded benchmarks, applications, luindex & lusearch from DaCapo suite and most of the SPEC benchmarks (with the exception of javac) gets clustered together on the right side.

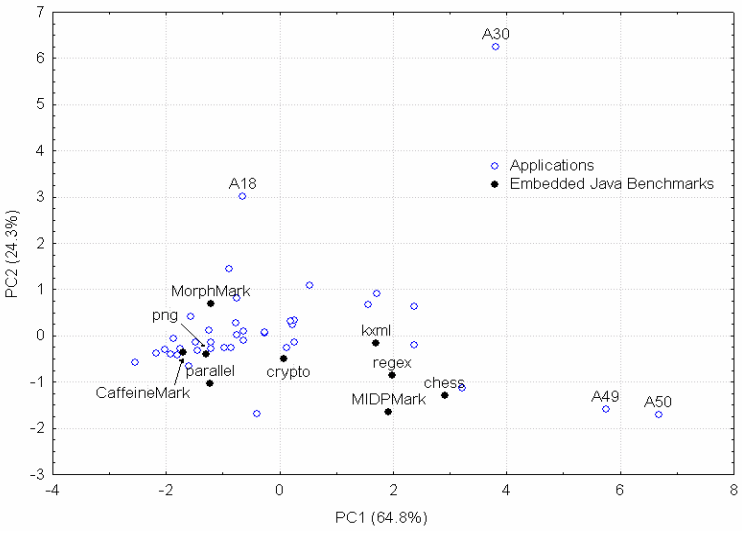


Figure 2. Principal component based analysis using C-k complexity metrics for Embedded benchmarks and mobile phone applications

Table 7: Complexity Metrics for Embedded Java Benchmarks and Applications

Benchmark	WMC: Weighted methods per class	DIT: Depth of Inheritance Tree	NOC: Number of Children	CBO: Coupling between object classes	RFC: Response for a Class	LCOM: Lack of cohesion in methods
Chess	183	49	8	143	468	362
Crypto	123	19	4	59	248	487
KXML	186	28	8	72	516	922
Parallel	37	15	4	27	85	31
Png	64	11	0	26	192	77
Regex	191	42	13	47	391	691
Caff	64	7	0	14	113	88
MIDP	113	47	16	52	365	450
Morphmark	94	7	0	11	218	1570
MIN	**37**	**7**	**0**	**11**	**85**	**31**
MAX	**191**	**49**	**16**	**143**	**516**	**1570**
Arith.MEAN	**117.22**	**25**	**5.88**	**50.11**	**288.44**	**519.77**
Applications						
A1	36	9	2	15	131	29
A2	126	8	0	14	343	1329
A3	64	8	2	11	116	774
A4	220	44	6	71	627	547
A5	107	13	0	40	271	727
A6	109	13	0	35	322	140
A7	208	42	13	123	562	471
A8	122	9	0	22	332	461
A9	85	13	0	39	249	264
A10	34	2	0	3	86	79
A11	239	24	4	54	601	1019
A12	75	8	0	24	252	206
A13	43	3	0	6	107	175
A14	140	4	1	6	235	2617
A15	128	24	5	34	265	1806
A16	75	4	0	8	172	1163
A17	69	12	0	19	247	60
A18	136	4	0	12	223	5601
A19	239	25	5	47	637	1473
A20	261	36	2	73	765	688
A21	180	14	0	45	578	1375
A22	200	13	4	26	393	544
A23	128	12	1	38	405	240
A24	58	3	0	7	116	485
A25	160	18	0	47	434	557
A26	2	0	8	121	157	2
A27	128	12	1	38	405	240
A28	77	12	1	20	297	67
A29	52	5	0	11	164	92
A30	410	16	0	61	1131	8369
A31	43	5	0	8	125	26
A32	128	22	0	62	373	165
A33	128	12	1	38	405	240
A34	116	13	0	48	406	370
A36	81	6	0	13	297	377
A37	144	21	2	49	393	291
A38	111	12	0	23	327	181
A45	65	8	0	23	175	116
A48	8	1	0	0	10	8
A49	300	62	21	163	764	569
A50	325	68	22	199	835	700
A52	44	6	0	15	128	63
MIN	**2**	**0**	**0**	**0**	**10**	**2**
MAX	**410**	**68**	**22**	**199**	**1131**	**8369**
Arith. MEAN	**128.6667**	**15.38095**	**2.404762**	**40.7381**	**353.8333**	**826.3333**

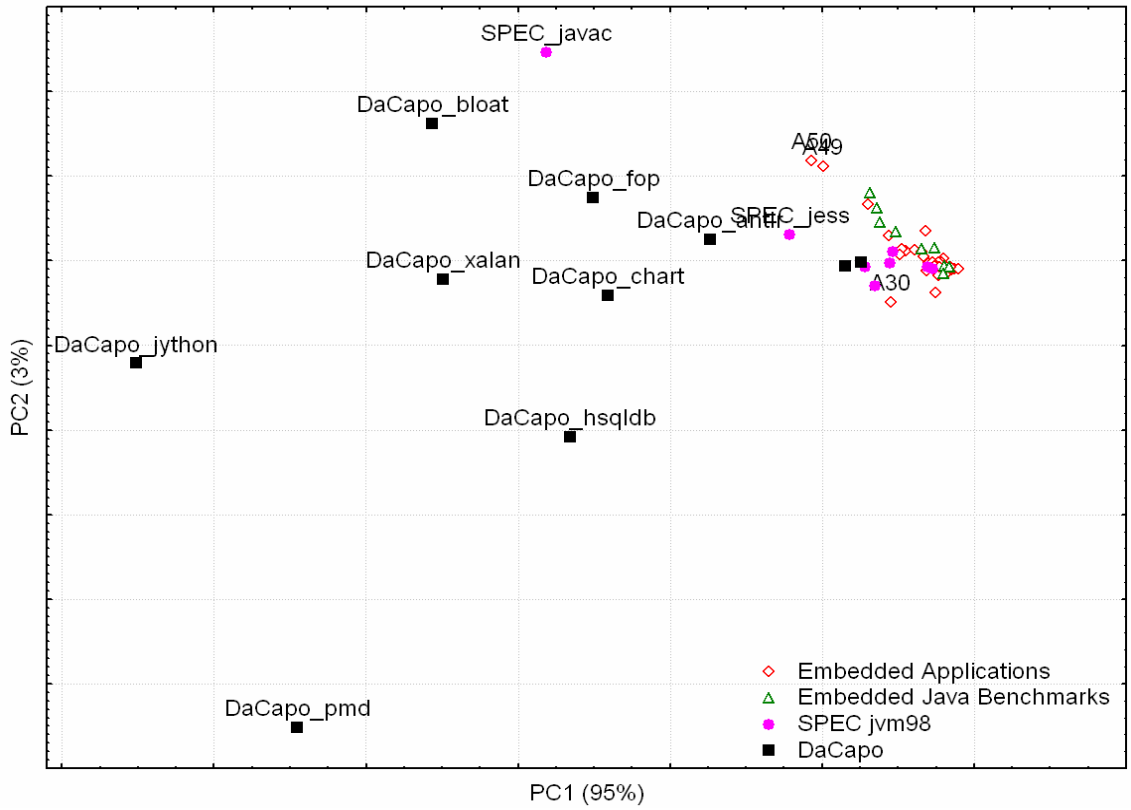


Figure 3. Principal component based analysis using C-k metrics for Embedded, SPECjvm98 and DaCapo benchmarks with mobile phone application suite

B. Object allocation/liveness/locality analysis

This section presents two important program properties namely 1) memory allocation and management, and 2) code reuse. The memory allocation and management properties are of greatest concern to virtual machine designers. The code reuse information is useful to VM and hardware designers. The memory behavior of objects is gathered by using the limited support for JVMPI that IBM J9 VM offers. Due to some of the stability and dependency issues with JVMPI in IBM J9, the code reuse was done using the Sprint Wireless Toolkit 3.1[28]. The Sprint Wireless Toolkit is an emulation of the embedded Java platform with support for some device and vendor specific libraries, which enables us to execute most of our application suite.

1) Dynamic Object Memory behavior

To capture the dynamic memory profile of the programs, we measure some of the metrics suggested by Dieckmann et al [8]. The VM was modified using a plug-in written using JVMPI. This allowed for data collection and regular (controlled) invocation of the GC. We invoked the GC once in every 10K of allocation and measured both object allocation and live object statistics. A brief discussion on each of the metrics follows.

Allocated bytes: measures the total number of bytes allocated by this program.

Live bytes: measures the total number of bytes alive at the end of a 10k allocation cycle, allowing one to compare the data survival behavior. The amount of the bytes that stay alive can determine the size of the second level of the heap and the algorithm used to compact the heap. Since data compaction tends to perform bulk copy, support for such operations in the architecture could be helpful; but the level of support needed would be partly determined by the amount of live bytes.

Allocated to Live bytes ratio*:* measures the ratio of allocated to live bytes. This gives an idea of the pressure on the GC in terms of that data allocation and de-allocation rate. A high ratio would signal that allocation and de-allocation are happening at a high rate (i.e. low survival rate with most of the objects being de-allocated quickly). A high ratio would support the weak generation hypothesis [20], the hypothesis that most objects die young, making it a valuable information to the GC designers. The usefulness of this metric extends to computer architects too. A high ratio is likely to imply that large amount of data is being allocated and stored in the high level memory and caches and then being discarded at a very high rate. This could lead to a low utilization of the cache hierarchy.

Allocated objects: measures the total number of objects allocated by this program. Allocation of a large number of objects requires an appropriately tuned heap management, a useful input to the Virtual Machine designers. A large number of allocations can pressure the TLB.

Live objects: measures the total number of objects alive at the end of a 10k allocation cycle. The larger the number of objects that survive, the more difficult could be the GC operations and possibly more number of data copy operation are required.

Allocated to Live objects ratio: measures the ratio of allocated to live objects. This metric is analogous to the allocated to live bytes ratio except that now it concerns object count instead of bytes. This metric gives us an idea of the pressure on the GC in terms of object count. This data in conjunction to the object size related data can give further insights to GC designers as well as micro-architects. For example, a large object count ratio can have implication on the cache block size. If the objects are small enough a number of them could fit into the same cache block. This could work to ones

advantage and disadvantage. On the positive side it provides prefetching but on the negative side it can lead to fragmentation of the heap and thus reducing the utilization of the caches as well as causing misaligned reads, typically a costlier operation.

Average Allocated Object size: measures the average size of the objects allocated by the program. Large objects will have different VM architecture and micro-architectural requirements compared to small objects. For example, large objects are better off having larger cache block sizes and the ability to write larger amount of data in bulk (the bandwidth).

Average Live Object size: measures the average size of the live (as defined before) objects allocated by the program. If the objects that survive are smaller, more data operations are required to manage them and the efficiency of small data operations become critical (e.g. latency of data operations).

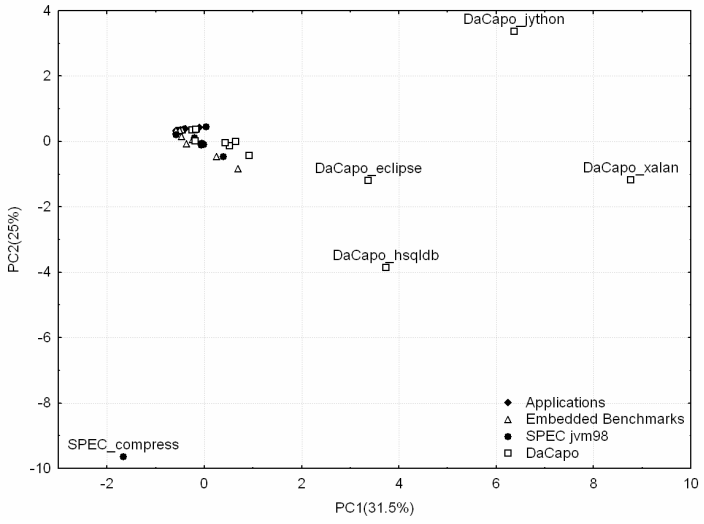
Table 8 shows the object allocation/liveness metrics for the various embedded benchmarks and embedded applications. Due to space limitation, the data for each individual benchmark and each mobile application is included in Appendix I. On average, benchmarks allocate more heap than applications, but the variation in amount of heap allocated is higher in applications than benchmarks. Thus there is a difference between average behaviors versus the range of behaviors. Similarly, on average, benchmarks allocate more objects; however, there is more diversity in object count allocation between various applications than between the benchmarks. The average allocated/live ratio is similar for both applications and benchmarks, although applications have more diversity. With respect to average object size, the benchmarks exhibit a wider range than applications, both in allocated and live. To simplify comparison and visualization we plot the two principal components PC1 and PC2 that account for more than 72% of the variance in the data (See Figure 4). The applications occupy a space distinct from that of the benchmarks (one can draw a line separating out the applications from the benchmarks). This is true of all the benchmarks except Morphmark. On PC1, the benchmarks fall within the range exhibited by the applications, whereas on PC2, applications and benchmarks exhibit significant difference in behavior.

Table 8: Dynamic Object behavior of Embedded Benchmarks and Embedded Applications

	Heap Volume(MB)			Object Count			Average Object Size	
	Allocated	**Live**	**Alloc/Live**	**Allocated**	**Live**	**Alloc/Live**	**Allocated**	**Live**
Benchmarks								
Min	0.06	0.05	1.29	1367	1305	1.05	29.67	31.77
Max	101.11	25.62	153.82	2792726	655676	174.03	142.08	883.32
Average	36.47	7.22	30.57	819556	111149	42.96	70.48	220.37
GM	14.06	1.54	9.13	224925	14178	15.87	62.49	108.58
Median	29.25	2.70	6.86	420144	6819	16.21	65.84	83.66
Applications								
Min	0.20	0.09	2.30	4162	1773	2.35	32.03	35.12
Max	211.15	0.52	407.31	6591939	14760	446.61	49.67	90.67
Average	10.12	0.21	29.36	310842	3997	40.16	38.01	54.10
GM	1.80	0.19	9.35	47799	3624	13.19	37.73	53.22
Median	1.18	0.22	6.27	33225	3752	9.50	36.51	53.78

Table 9: Dynamic Object behavior of Desktop Java benchmarks. Generated from [3]

	Heap Volume(MB)			Object Count			Average Object Size	
	Allocated	**Live**	**Alloc/Live**	**Allocated**	**Live**	**Alloc/Live**	**Allocated**	**Live**
SpecJVM98								
Min	0.7	0.6	1.1	3022	270	1.9	22	25
Max	270.7	21.1	292.7	9393097	307043	786.1	28031	24425
Average	152.53	6.31	69.99	5081259.22	142958	142.76	3164	2797
GM	86.5	3.8	23	1180850	35886	32.9	77	110
Median	140.5	6.3	19.5	6158131	153555	22.4	32	56
DaCapo								
Min	100.3	0.1	2	2402403	2788	1.4	24	23
Max	60235.6	72	8104	161069019	3223276	9304.5	392	330
Average	6564.46	16.07	1055.83	38142567.27	442848	1095.64	79.09	84.09
GM	907.5	6.2	147.6	18112439	103890	174.3	53	62
Median	779.7	9.5	186	25940819	168921	224.2	44	55



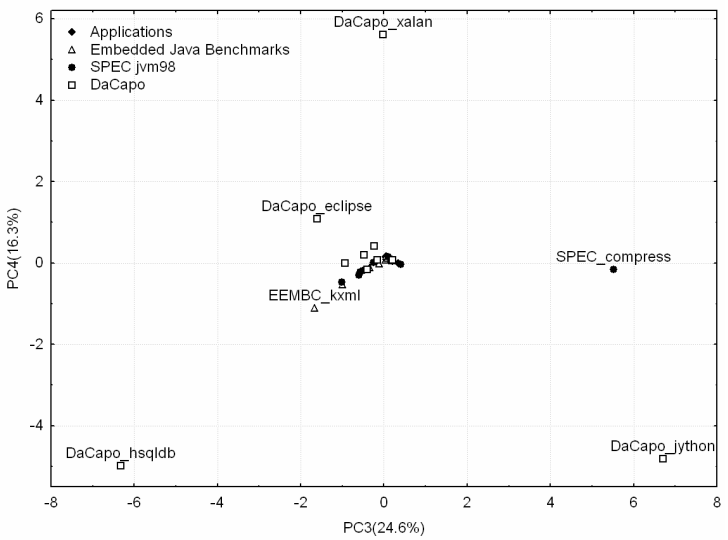


Figure 5. Principal component based analysis using dynamic object metrics for embedded benchmarks, embedded applications, Specjvm98 and DaCapo

Table 10: Code Reuse metrics

	hot fn - 80% calls	hot fn - 90% calls	lib % of hot fn calls	hot lib fns - 90% calls	% of lib calls	hot fn - 80% cycles	hot fn - 90% cycles	lib % of hot fn cycles	hot lib fns - 90% cycles	% of lib cycles	total Cycles	total Calls	Total Fnts
Benchmarks													
Min	3	4	0	0	0.5	1	2	0	0	0.34	2.30E+11	689237	78
Max	12	19	81.9	7	81.59	10	18	49.38	8	48.61	5.72E+11	65000000	172
Avg	7	11	31.25	3	30.1	6	11	14.14	2	14.7	3.39E+11	28000000	133
Median	7.5	11	28.08	4	26.62	7.5	12.5	10.1	2	10.78	3.221E+11	22000000	150.5
Application													
Min	1	1	2.33	1	8.39	2	4	6.59	1	10.94	3.52E+09	1346	43
Max	29	44	100	11	97.95	18	29	100	16	92.41	6.90E+11	24000000	318
Avg	12	17	40.19	5	39.63	6	10	65.48	6	63.39	8.44E+10	1182801	122
Median	12	17	43.16	4	42.18	5	9	67.73	5	63.71	3.332E+10	87097	109

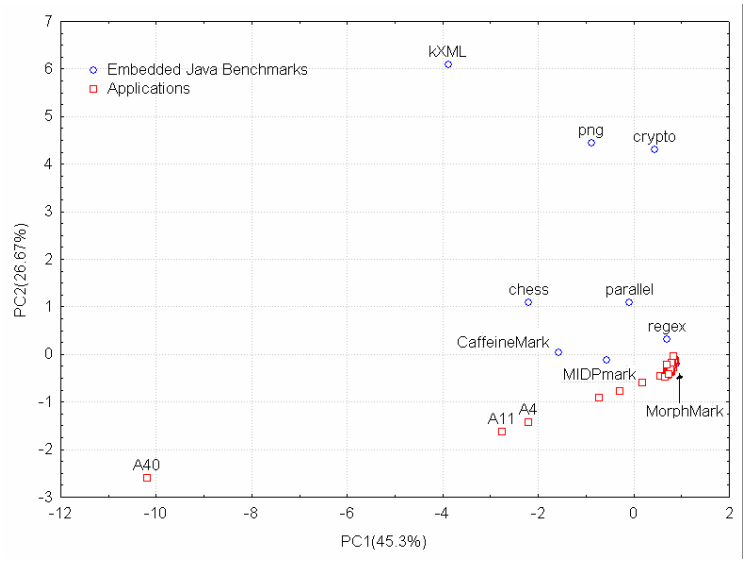


Figure 4. Principal component based analysis using dynamic object metrics for comparison of Embedded benchmarks with mobile phone application suite. To avoid cluttering not all points are labeled

Now, let us consider SPECjvm98 and DaCapo. Table 9 presents a summary of the comparison between embedded Java benchmarks and the SPECjvm98/DaCapo information with the same IBM J9 tool suite. Due to space limitation, a summary of the results from Blackburn et al [3] is included as Appendix II. On average, the object sizes are higher for SPECjvm98 for both allocated and live. However, live size is higher for DaCapo compared to SPEC. The PCA plot for the workload space with embedded benchmarks, embedded applications, SPECjvm98 and DaCapo is shown in Figure 5. Four principal components are presented, accounting for about 97% of the variance in the data. As expected, DaCapo's richness in code complexity translates to more complex behavior even for object allocation and management. Once all the Java workloads are presented in the same figure (as in Fig 5), some outliers from DaCapo and one from SPECjvm98 programs (compress) define a broad envelope, however, all embedded benchmarks and applications and many SPECjvm98 programs are clustered in a small region of the workload space. The embedded Java benchmarks have more similarity to SPECjvm98 than DaCapo.

Since many garbage collection algorithms are most concerned with live object behaviors these demographics are more indicative for designers of new garbage collection mechanisms. The object demographics also dictate many other choices such as the design of per-object metadata, locking mechanisms, etc.

All such decisions in the end affect the performance of the micro-architecture.

2. Code reuse/hotness:

Modern Virtual Machines implement various techniques to take advantage of the reuse of code segments; either by caching bytecode translation or compiling frequently used code segments into more efficient native code during runtime or ahead of time. The impact of these techniques depends on the code hotness behavior of different applications. Hence, to compare the applications and benchmarks from the perspective of code hotness, we use some metrics pertaining to code reuse. We collect reuse information at a function granularity using the Sprint Wireless Toolkit 3.1[28]. The metrics and the reasoning behind using them are discussed below.

Hot 80% function call: The number of functions required to make up 80% of the total function calls. The smaller the number, the higher the code reuse.
Hot 90% function call: The number of functions required to make up 90% of the total function calls.
Library % of hot function calls: The percentage of hot functions calls that are libraries. This metric and the next five metrics were designed to measure the contribution of library functions to the top hot functions. A high reuse of library functions can be exploited by the VM by say pre-compiling those functions ahead of time or caching or preloading translations for those functions. This can also affect the hardware architecture. For example, storing precompiled code can affect space available in the system memory. Another good example would be for Java hardware accelerators like Jazelle [1] in the ARM Cortex architecture. Jazelle, require the bytecode to be reorganized to make optimum use the hardware accelerator. If the applications are deployed dynamically over the network from a third party, such processor specific optimization might not be possible. That would be less of an issue if a significant part of the execution time is spent on library function which can be transformed ahead of time to take advantage of the hardware and stored in the device permanently.
Hot lib function calls: The number of library function among the hot functions (based on call count).
% of lib calls: The percent of the total function calls that are to library functions.
Library % of hot function cycles: The percentage of hot functions cycles that are libraries.
Hot lib function cycles: The number of library function among the hot functions (based on cycle count).
% of lib cycles: The percent of the total function cycles that are contributed by library functions.
Hot 80% function cycles: The number of functions required to make up 80% of the total cycles.
Hot 90% function cycles: The number of functions required to make up 90% of the total cycles.
Total Cycles: The total number of cycles it takes to execute the program. This gives on an idea of the typical execution length of programs.
Total Calls: The total number of function calls invoked.
Total Function Count: The total number of unique functions present.

Summary of the data is presented in Table 10(detailed data is included as Appendix III) From the summarized statistics we observe that the applications exhibit wider range of reuse metrics than the benchmarks. For example, it takes 1-29 functions to amount for 80% of the function calls in the case of applications while it takes only 3-12 functions to do the same for embedded java benchmarks. The locality of applications is as not as high that suggested by the benchmarks. The benchmarks need improvement in this respect. This finding also suggests that the effort in implementing even light weight JIT compilation is higher than what is projected by the current benchmarks. It is also interesting to note that the composition of libraries in the execution behavior is higher for applications when compared to the benchmarks. For example, in the applications, 65% of the cycles on average are library functions as opposed to 14% for the benchmarks. A higher contribution of library function has special significance for embedded Java because it allows for ahead of time compilation and optimization of key libraries to be worthwhile. This allows one to side-step JIT to some degree and to attain the same benefits by employing ahead-of-Time compilation and ROMizing of the library functions.

V. Related Work

Prior work [17, 18, 2, 3, 10, 19] in understanding the nature of Java workloads has focused on characterizing the performance of standard client and server applications using industry benchmarks such as SPECjvm98 and SpecJbb [24]. In the embedded domain, prior work includes Chen et al. [6] which characterized the performance of embedded Java applications through the development of their PennBench suite. Even though this study concerns Java in the embedded space, the analysis is limited to method and object distribution across the proposed suite. Furthermore, our study focuses on industry standard benchmarks and performs the analysis of the method and object level properties elaborately. In another study by Chen et al [5], the impact of various garbage collection properties and allocation strategies, that can take advantage of banked memory, are studied. Griffin et al [13] also proposed improvements to the garbage collection strategies with the objective of being energy efficient for embedded systems. They use a small suite of application to model embedded Java applications and only focus on relative benefit of their scheme, but no characterization of the applications is presented. We perform an elaborate characterization of standard embedded benchmarks and embedded applications. We also present a comparison between desktop and embedded Java benchmarks.

VI. Conclusion

With the goal of understanding embedded Java benchmarks and their representativeness of real-world embedded applications, we gathered and characterized industry benchmarks such as MIDPmark, Morphmark, Caffeine, EEMBC Java benchmarks, and an actual cell phone application suite. Using a versatile embedded virtual machine, a Sprint wireless toolkit, and a JVMPI based module, we obtained the characteristics of embedded benchmarks and approximately 50 applications from the cell phone Java application suite. On code complexity metrics, as well as object allocation and live properties, there are differences between the properties of embedded benchmarks and embedded applications. On most characteristics, the range of variability represented by the applications is much higher than the range of variability exhibited by the benchmarks. On the object allocation and live properties, the benchmarks and applications occupy an almost disjoint space. Furthermore, we find that the applications exhibit less code reuse/hotness compared to the embedded benchmarks. Future embedded Java benchmarks

can be designed to include benchmarks with a wider range of complexity characteristics as well as object allocation properties.

We also compared the embedded programs to desktop/client benchmarks such as SPECjvm98 and DaCapo. Utilizing Principal Component Analysis and clustering techniques we present a similarity/dissimilarity analysis of the different Java benchmarks. As expected, desktop Java benchmarks, especially DaCapo, is significantly richer in code complexity metrics and object allocation/liveness characteristics. Embedded benchmarks and embedded applications are seen to be different from SPECjvm98 and DaCapo programs.

We also find that, embedded applications spend a significant part of the time executing bytecodes from the standard library; on average 65% of the time. This suggests that software and hardware techniques that aid ahead of time compilation of library functions can provide a significant performance boost without having to incur the resource overhead of just-in-time compilation of code. Compared to the applications, embedded java benchmarks spend far less time, an average of 14%, in libraries.

VII. Acknowledgement

We would like to acknowledge Kathryn S McKinley (Department of Computer Sciences, The University of Texas at Austin) for her guidance and help in the selection of metrics and the evaluation methodology for Java. The authors are supported in part by NSF grants 0429806 & 0702694, the IBM Systems and Technology Division, IBM CAS Program, and AMD.

References

[1] ARM Inc, "Jazelle.", *http://www.arm.com/products/esd/jazelle_home.html*

[2] Barisone, A., Bellotti, F., Berta, R., and Gloria, A. D., "Instruction Level Characterization of Java Virtual Machine Workload," presented at *Workload Characterization for Computer System Design*, 1999.

[3] Blackburn, S.M., et. al, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," *The ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2006.

[4] Chidamber, S. R. and Kemerer, C. F. "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476-493, 1994.

[5] Chen, G., Shetty, R., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Wolczko, M.,"Tuning garbage collection in an embedded Java environment", *Proceedings. Eighth International Symposium on High-Performance Computer Architecture,* 2002.

[6] Chen, G., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J., "PennBench: a benchmark suite for embedded Java," presented at *IEEE International Workshop on Workload Characterization*, 2002.

[7] Club Java, "MIDP Mark.", *http://www.club-java.com/TastePhone/MIDP.jsp/*

[8] Dieckmann, S., Hölzle, U., "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark", *Proceedings of the 13th European Conference on Object-Oriented Programming*, p.92-115, June 14-18, 1999

[9] G. Dunteman, *Principal Components Analysis*, Sage Publications, 1989.

[10] Eeckhout, L., Georges, A., and Bosschere, K. D., "How Java programs interact with virtual machines at the microarchitectural level," presented at *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, Anaheim, California, USA, 2003.

[11] Eeckhout, L., Vandierendonck, H., and Bosschere, K. D., "Workload Design: Selecting Representative Program-Input Pairs," presented at *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2002.

[12] EEMBC. Ltd., "Java GrinderBench.", *http://www.grinderbench.com/.*

[13] Griffin, P., Srisa-an,W. and Chang, J. M., "An energy efficient garbage collector for java embedded devices", *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*,2005.

[14] Hugues J. De La Vergne, "Dataquest Insight: Java and BREW in Mobile Devices", *Gartner*, April, 2007.

[15] IBM Inc, "WebSphere Everyplace Micro Environment", *http://www-306.ibm.com/software/wireless/weme/*

[16] Infoworld, *http://weblog.infoworld.com/techwatch/archives/006425.html.*

[17] Kim, J.-S., and Hsu, Y., "Analyzing Memory Reference Traces of Java Programs," presented at *Workload Characterization for Computer System Design*, 2000.

[18] Kim, J.-S., and Hsu, Y., "Memory system behavior of Java programs: methodology and analysis," presented at *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Santa Clara, California, United States, 2000.

[19] Li, T., John, L. K., Narayanan, V., Sivasubramaniam, A., Sabarinathan, J. and Murthy, A., "Using complete system simulation to characterize SPECjvm98 benchmarks", *Proceedings of the 14th international conference on Supercomputing.*, pp 22-33. 2000.

[20] Lieberman, H. and Hewitt, C., "A real-time garbage collector based on the lifetimes of objects", *Communications of the ACM*, Vol 26, Issue 6, pp 419--429, 1983.

[21] McKinley, K. S. and Blackburn, S. M., "O Java, Java! Wherefore Art Thou Java?", Invited paper, *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Phoenix, AZ, January 2007.

[22] Morpheme Ltd., "Morphmark.", *http://www.morphmark.com/index.jsp*.

[23] Pendragon Software, "Caffeine Mark 3.0.", *http://www.pendragon-software.com/.*

[24] Standard Performance Evaluation Corporation, "SPEC Java Business Benchmark (SPECjbb2000)." http://www.spec.org/jbb2000/

[25] Sun Microsystems, "Java.", *http://java.sun.com/.*

[26] Standard Performance Evaluation Corporation, "SPEC JVM98 Benchmarks.", *http://www.spec.org/jvm98/.*

[27] Spinellis, D. D., "ckjm Chidamber and Kemerer metrics Software", *http://www.spinellis.gr/sw/ckjm/.*

[28] Sprint Nextel Inc, "Sprint Wireless Toolkit.", *http://developer.sprint.com*

[29] Sun Microsystems, "J2ME Building Blocks for Mobile Devices."

Appendix II. Average allocated and live object size and count data – Desktop Java Benchmarks(Data from [3])

	Heap Volume(MB)			Heap objects			Mean Object Size	
Benchmark	**Alloc**	**Live**	**Alloc/Live**	**Alloc**	**Live**	**Alloc/Live**	**Alloc**	**Live**
SPEC								
201.compress	105.4	6.3	16.8	3942	270	14.6	28031	24425
202.jess	262	1.2	221.3	7955141	22150	359.1	35	56
205.raytrace	133.5	3.8	35.1	6397943	153555	41.7	22	26
209.db	74.6	8.5	8.8	3218642	291681	11	24	31
213.javac	178.3	7.2	24.8	5911991	263383	22.4	32	29
222.mpegaudio	0.7	0.6	1.1	3022	1623	1.9	245	406
227.mtrt	140.5	7.2	19.5	6689424	307043	21.8	22	25
228.jack	270.7	0.9	292.7	9393097	11949	786.1	30	81
pseudojbb	207.1	21.1	9.8	6158131	234968	26.2	35	94
DaCapo								
antlr	237.9	1	248.8	4208403	15566	270.4	59	64
bloat	1222.5	6.2	195.6	33487434	149395	224.2	38	44
chart	742.8	9.5	77.9	26661848	190184	140.2	29	53
eclipse	5582	30	186	104162353	470333	221.5	56	67
fop	100.3	6.9	14.5	2402403	177718	13.5	44	41
hsqldb	142.7	72	2	4514965	3223276	1.4	33	23
jython	1183.4	0.1	8104	25940819	2788	9304.5	48	55
luindex	201.4	1	201.7	7202623	18566	387.9	29	56
lusearch	1780.8	10.9	162.8	15780651	34792	453.6	118	330
pmd	779.7	13.7	56.8	34137722	419789	81.3	24	34
xalan	60235.6	25.5	2364	161069019	168921	953.5	392	158

Appendix I. Average allocated and live object size and count data

	Heap Volume(MB)			Heap objects			Mean Object Size	
Benchmarks	**Allocated**	**Live**	**Alloc/Live**	**Allocated**	**Live**	**Alloc/Live**	**Allocated**	**Live**
CaffeineMark	31.04	0.20	153.82	300202	1725	174.03	103.38	116.96
EEMBC_chess	82.33	5.48	15.02	2775187	172512	16.09	29.67	31.77
EEMBC_crypto	19.57	7.15	2.73	137712	8100	17.00	142.08	883.32
EEMBC_kXML	101.11	25.62	3.95	2792726	655676	4.26	36.21	39.08
EEMBC_parallel	18.51	2.70	6.86	420144	4687	89.64	44.07	576.13
EEMBC_png	43.50	23.21	1.87	465707	147129	3.17	93.41	157.75
EEMBC_regex	2.83	0.20	14.19	38700	2387	16.21	73.21	83.66
MIDPMarkMain	29.25	0.39	75.41	444260	6819	65.15	65.84	56.88
MorphMark	0.06	0.05	1.29	1367	1305	1.05	46.44	37.80
AVG	**36.47**	**7.22**	**30.57**	**819556.11**	**111148.89**	**42.95**	**70.48**	**220.37**
Applications								
A1	0.20	0.09	2.30	4162	1773	2.35	48.33	49.43
A2	2.35	0.28	5.88	28368	3902	7.27	36.21	44.79
A3	1.11	0.13	2.63	11811	3721	3.17	46.91	56.64
A4	36.21	0.26	7.48	31395	3018	10.40	36.76	51.13
A5	18.25	0.26	8.25	42702	3577	11.94	36.26	52.49
A6	1.03	0.17	9.59	36627	2481	14.76	35.84	55.16
A7	1.09	0.29	9.34	36487	2445	14.92	35.69	57.01
A9	7.85	0.26	5.13	32146	2864	11.22	35.30	77.27
A10	0.46	0.12	5.01	40652	4195	9.69	37.67	72.88
A11	41.15	0.26	8.83	34303	3242	10.58	35.19	42.17
A12	0.73	0.19	9.63	50333	3669	13.72	43.41	61.84
A13	0.39	0.11	8.44	67827	5137	13.20	34.58	54.12
A14	1.10	0.25	8.47	28039	3015	9.30	39.62	43.48
A16	13.38	0.25	140.06	1127123	5324	211.71	32.13	48.56
A17	0.55	0.21	70.33	565806	4857	116.49	32.26	53.43
A21	3.95	0.26	3.71	28626	4777	5.99	38.17	61.69
A22	1.15	0.15	30.10	240104	4533	52.97	32.68	57.50
A23	1.55	0.19	3.68	10632	2137	4.98	43.14	58.30
A24	0.83	0.19	160.03	1281856	4855	264.03	32.10	52.96
A27	1.31	0.14	3.84	18117	4086	4.43	40.32	46.54
A28	1.60	0.26	3.59	9642	2060	4.68	40.34	52.53
A29	0.98	0.10	4.40	28919	4477	6.46	37.98	55.71
A31	0.85	0.26	54.25	413785	5108	81.01	32.34	48.29
A32	0.52	0.10	15.18	115933	6085	19.05	34.08	42.77
A33	1.30	0.14	4.27	18889	3489	5.41	43.94	55.71
A35	1.13	0.22	6.07	45637	5092	8.96	34.98	51.69
A36	1.53	0.31	9.88	28227	1833	15.40	34.87	54.35
A38	0.88	0.32	3.28	21764	2853	7.63	39.01	90.67
A40	211.15	0.52	5.28	11953	1780	6.72	43.79	55.65
A41	0.70	0.12	2.75	17643	5394	3.27	49.67	59.04
A43	1.34	0.24	407.31	6591939	14760	446.61	32.03	35.12
A44	1.44	0.24	5.94	18347	2711	6.77	37.99	43.32
A49	2.04	0.15	5.55	36941	4278	8.64	36.14	56.20
A50	1.21	0.14	6.10	37694	4545	8.29	38.21	51.93
A51	0.77	0.12	13.97	59250	3782	15.67	34.38	38.56
A52	2.18	0.23	6.44	16644	2025	8.22	46.06	58.80
AVG	**10.12**	**0.21**	**30.02**	**319247.97**	**4053.00**	**41.08**	**37.78**	**53.97**

Appendix III: Code/function reuse characteristics.

Benchmarks	total Cycles	total Calls	Total Fnts	hot fn - 80% calls	hot fn - 80% cycles	hot fn - 90% calls	hot fn - 90% cycles	lib % of hot fn calls	lib % of hot fn cycles	hot lib fns - 90% calls	hot lib fns - 90% cycles	% of lib calls	% of lib cycles
CaffeineMark	5.718E+11	1737511	78	4	2	6	4	56.67	49.38	4	3	54.28	48.61
eembc_chess	2.301E+11	2E+07	172	12	9	15	16	43.75	10.10	4	3	40.56	10.78
eembc_crypto	3.216E+11	3.5E+07	162	8	9	12	16	2.65	0.00	1	0	2.40	1.14
eembc_kxml	2.678E+11	2.4E+07	153	7	10	19	18	12.40	33.17	7	8	12.69	31.74
eembc_parallel	3.226E+11	689237	83	4	1	5	2	81.90	10.35	4	1	81.59	13.54
eembc_png	3.836E+11	6.5E+07	97	3	3	4	4	0.00	0.00	0	0	0.50	0.71
eembc_regex	3.827E+11	6.1E+07	148	8	6	10	9	8.87	0.00	1	0	8.19	0.34
midpmark	2.301E+11	2E+07	172	12	9	15	16	43.75	10.10	4	3	40.56	10.78
AVG	***3.388E+11***	***2.8E+07***	***133***	***7***	***6***	***11***	***11***	***31.25***	***14.14***	***3***	***2***	***30.10***	***14.70***
Apps													
A1	5.242E+09	1346	66	16	5	24	8	34.21	66.51	8	4	34.10	62.79
A2	4.813E+10	409796	150	15	5	22	12	20.61	52.10	2	4	19.18	50.10
A3	9.828E+09	25006	80	17	8	21	13	46.02	79.19	6	11	46.27	75.14
A4	6.903E+11	2.4E+07	233	13	5	18	12	24.58	54.52	4	4	23.58	52.39
A5	1.875E+11	91531	112	14	3	18	4	44.56	6.59	4	1	42.48	10.94
A6	1.553E+10	72717	120	10	6	20	12	38.77	93.68	8	8	38.27	90.06
A7	9.156E+09	917141	207	1	10	1	16	100.00	78.60	1	12	97.95	75.00
A10	1.74E+10	8833	61	8	6	11	8	38.82	37.54	4	4	39.67	38.42
A11	1.632E+11	4328264	224	19	7	30	16	13.33	70.57	4	7	13.00	65.35
A12	2.301E+10	22310	83	8	4	10	6	19.13	19.30	2	3	19.91	22.13
A13	2.536E+10	32687	73	8	6	11	10	52.69	42.04	5	6	49.97	41.84
A14	3.604E+10	187220	121	26	4	38	7	49.56	100.00	8	7	45.28	92.41
A15	6.364E+10	448268	120	10	5	17	9	46.14	94.59	6	7	46.43	90.29
A16	8.518E+10	789351	109	6	8	13	13	19.58	67.73	4	9	21.76	63.71
A17	8.423E+09	3084	99	16	4	21	7	38.37	90.43	7	5	38.78	88.80
A21	8.715E+09	98117	137	6	10	11	15	45.64	77.21	6	10	45.16	76.26
A22	1.966E+10	131164	121	18	7	26	12	16.49	93.29	5	8	16.34	86.56
A23	4.89E+10	51242	119	14	6	18	9	46.56	53.00	7	6	45.21	52.20
A24	4.845E+10	40477	72	7	3	9	6	76.61	76.24	5	4	72.06	76.01
A25	9.623E+09	23753	122	21	5	24	9	26.19	94.79	5	7	24.83	91.52
A27	7.929E+10	79413	120	13	6	17	9	47.46	49.96	7	6	45.56	48.94
A28	1.271E+10	37932	87	2	5	5	7	48.80	78.25	2	4	48.33	76.19
A29	5.922E+10	94008	73	12	6	17	9	40.47	75.38	4	6	41.75	69.98
A30	8.906E+10	1847010	225	7	18	14	29	2.33	43.13	2	16	8.39	44.39
A31	5.296E+10	118068	62	8	3	11	5	70.19	41.04	4	3	66.33	41.50
A32	6.856E+09	15317	130	12	8	23	11	54.64	94.98	7	10	51.27	88.31
A33	9.97E+10	94699	120	13	6	17	8	47.40	47.59	7	5	45.33	47.64
A34	3.449E+11	252335	142	5	3	11	5	59.65	15.12	5	3	54.79	15.63
A35	2.299E+10	38546	61	8	3	9	4	53.59	46.44	3	2	56.15	47.29
A36	3.332E+10	59541	104	17	7	23	10	31.68	52.45	6	8	31.88	52.83
A38	1.179E+10	40091	104	12	6	18	9	51.31	95.26	9	7	49.69	90.89
A39	8.227E+09	9244	51	9	5	11	8	43.16	62.52	3	5	41.26	61.66
A40	6.002E+11	1.3E+07	307	17	5	24	10	27.44	87.23	4	5	27.85	81.30
A41	1.911E+10	8564	47	6	3	7	6	17.44	63.47	2	3	18.97	61.81
A42	4.078E+11	2505370	125	5	4	10	5	53.21	74.02	3	3	50.22	71.73
A43	1.669E+10	38771	71	12	4	15	6	5.60	34.35	2	2	10.74	35.51
A44	6.391E+10	45491	96	7	4	10	8	26.21	52.68	3	4	27.52	52.78
A45	1.498E+10	172246	79	7	3	11	8	42.96	76.01	3	5	42.18	72.49
A47	3.521E+09	1350	43	13	6	16	11	46.03	59.12	7	7	50.59	60.81
A48	3.6E+10	87097	94	4	4	7	7	45.20	88.22	2	5	45.24	84.08
A49	2.37E+10	129003	304	29	7	44	13	26.44	97.03	11	12	25.76	90.77
A50	3.772E+10	228200	318	27	9	42	17	19.10	92.13	10	12	19.55	86.35
A52	5.954E+10	118594	67	4	2	7	4	70.21	41.27	2	2	64.66	41.06
AVG	***8.436E+10***	***1182801***	***122***	***12***	***6***	***17***	***10***	***40.19***	***65.48***	***5***	***6***	***39.63***	***63.39***