A Study of Code Reuse and Sharing Characteristics of Java Applications

Marie T. Conte Andrew R. Trick John C. Gyllenhaal Wen-mei W. Hwu

Department of Electrical and Computer Engineering
Coordinated Science Laboratory
Center for Reliable and High-performance Computing
University of Illinois at Urbana-Champaign
{mconte, atrick, gyllen, hwu}@crhc.uiuc.edu

Abstract

This paper presents a detailed characterization of Java application and applet workloads in terms of reuse and sharing of Java code at the program, class, and method level. In order to expose more sharing opportunities, techniques for detecting code equivalence even in the presence of minor code changes or constant pool index differences are also proposed and examined. The analyzed application workload consists of the recently released SPECjvm98 benchmarks and the applet workload is derived from three extensive searches of the Internet between May 1997 and May 1998 using an enhanced web crawler. Analysis of these workloads reveals several new code sharing and optimization opportunities.

1. Introduction and Motivation

The popularity of Java bytecode as a means of distributing software marks an important shift in the way machines manage executable content. This shift is from large, primarily monolithic binary executables to distributed modules of portable code that can be invoked by different systems and can serve as components in different application contexts. The key factors that are making this transition possible are the portability and mobility of Java bytecode. Two features that enable Java's portability are the standard bytecode instruction format and the standard Java API. Java's compact class file representation is also ideal for transferring code across networks, and the runtime resolution provided by the Java Virtual Machine allows independent modules distributed across a network to interact [1,2]. The same features that uniquely qualify Java bytecode for this new software distribution paradigm have also caused Java to exhibit unique workload characteristics. Consequently, a successful Java runtime system should take into consideration the characteristics that set Java apart from other types of workloads.

In this paper we explore one such aspect of Java workloads. We characterize Java workloads in terms of the redundancy that they exhibit in both the realms of applications and applets. In the traditional model of software distribution, each executable is considered an independent unit of code. Shared libraries and DLLs are often used to reduce total code size across multiple applications by sharing a portion of the code space. However, compilers continue to treat each binary file as an independent entity. In the Java domain, code is shared between modules at an extremely fine level of granularity. Linking and loading is done following a lazy model. In other words all class files used within a Java application or applet, are located, linked, and loaded only when the executing program accesses a portion of the code that requires them. Consequently, an efficient compiler or virtual machine must consider the relationship between class files. By detecting and exploiting redundant patterns within and across bytecode files, Java virtual machines can achieve greater efficiency.

In this paper, we report a study based on a broad range of applet and application bytecode. We provide results demonstrating the redundancy at multiple levels ranging from duplicate class files to a finer grain generic method level. We also suggest methods for exploiting this redundancy to improve the overall system performance.

2. Coarse Grain Redundancy

Our use of the general term *bytecode redundancy* simply implies that bytecode from different sources displays shared characteristics. However, this redundancy can exist at several levels. At the highest level, a single Java applet may be linked by multiple web pages. This can be viewed as a redundancy in the web pages rather than the actual bytecode files. Class file loaders that can detect this overlap save time by avoiding re-downloading the applet unnecessarily. However, rather than simply being linked from multiple pages, we observed in our study that applets are frequently copied to different physical locations. An intelligent class loader can also detect this situation to avoid redundant downloading, linking, and compiling. This coexistence of identical class files in multiple locations is also seen among Java applications. Vendors often bundle class files into a Java Archive (JAR) file [3]. JARs from different vendors will exhibit an overlap when the same class files exist in multiple JARs. This too can be exploited to reduce the system's overall code size.

2.1 Analysis of Coarse Grain Redundancy

To study the overlap of Java applets on physically different web pages, we needed to visit a large quantity of web pages. One way of accomplishing this is to start with a set of seed web pages and crawl across the web following the links on each accessed page while taking care not to re-visit sites in an endless loop. We took this approach and wrote our own modified web crawler that would look specifically for Java enhanced pages. Once we had arrived at what we felt was a large enough set of Java enhanced pages for our analysis, we downloaded the applets referenced on them. Table 1 shows the results for three separate runs of the web crawler spread throughout the course of one year. Row one of this table shows the total number of web pages we found with Java applets. Occasionally, a Java enhanced page will reference more than one applet which is reflected by the larger number of applets than web pages when comparing rows one and two in Table 1. Even though our web crawler found a

Collected	May 1997	Nov 1997	May 1998
Number of Pages	1616	5198	4316
Number of Applets	1939	7072	7194
Unique (location)	1465	1721	2959
Unique (code)	786	976	1595

Table 1: Applet redundancy from web experiments.

smaller number of Java enhanced pages in the May 1998 run than the November 1997 run, we observed an increase in the average number of applets per page from the 1.35 average in November to a 1.67 average in May. We also downloaded all the applets referenced on the web pages, determining their physical location by examining the "codebase" tag in the applet section of the web page. The third row in Table 1 lists the total number of unique applets based on the full path locations, or the number that may be identified as physically unique by name and location. Comparing the number of physically unique applets by location to the total number of applet references (row 2) indicates how frequently applets are referenced on multiple web sites. Although the results vary across time, they show that a significant number of applet references address the same physical location. We then compared applets downloaded from different locations to check for duplicates. The fourth row in Table 1 lists the reduced set of unique applet class files after the downloaded files were compared using an exact byte-for-byte match on the entire file. For all three experiments, roughly 50% of the Java bytecode class files downloaded from physically unique locations (row 3) were redundant.

2.2 Avoiding Applet Redundancy

If a class loader can detect identical class files, it can avoid transferring the redundant files across the network and caching extra copies of the bytecode. The simplest method to improve performance is to avoid loading a class file from the same physical location multiple times. A class loader may simply check if the class has already been downloaded from the same path and, if so, use the copy it already has. Using this approach alone, however, a class loader runs the risk of using an out-of-date file instead of the latest version as required in the Java Language Specifications. Modification date requests could be used to help identify files that have changed between accesses. However, the inconsistency of file dates on PCs due to user level changes as well as drift and variance caused by the majority of PC's not running a date and time protocol such as Network Time Protocol (NTP), may cause this technique to fail. More importantly, this approach still fails to capture the redundancy of files at different physical locations.

Another technique, called *fingerprinting a file*, adopts the method used by the Internet community to identify packet uniqueness. This method relies on a quick hashing algorithm to compress the file into a short, unique number. Message Digest version 5 (MD5) is the current generation of this algorithm [4]. Our initial tests with MD5 fingerprinting showed that it is successful in detecting the full overlap between the files. The algorithm is widely available and has recently been included in the specification of HTTP version 1.1 [5]. It produces a 128-bit result that fits into a single IP packet requiring no more than the delay of a ping to obtain. When a class file that has the same name as one that has already been cached is requested from another site, a simple check against the MD5 fingerprint can verify whether or not it is indeed an identical class.

3. Fine Grain Redundancy

Although we have found that redundancy at the class level is very common, even more redundancy can be detected by considering individual methods within a class. Minor differences in class files can exist for many reasons. The most trivial type of difference we have noticed is a change in the name of the class. Because the class name is stored in the constant pool of the bytecode file, the class files will not appear identical and will produce different MD5 fingerprints. More commonly, a class will be copied but slightly varied to suit its new purpose. Frequently, programmers will reuse a class from another source, but will slightly specialize the class by changing variable names, modifying or adding just a few methods, or simply changing constant values. In all of these cases, the majority of the bytecode instructions are identical between modules.

Classes do not have to be closely related in order to exhibit fine grain redundancy. Even classes that are independently developed contain methods that are functionally the same if not exactly equivalent. For example, we have found it to be common for an instance initializer to call the initializer of the base class. Moreover, we observed that on average, more than 50% of these initializers resolve to root class, *Object*. A VM may want to keep an optimized initializer for use every time it observes this situation. Initializers are not the only methods sharing commonality across different class files. Another area of method overlap is caused by the pure data abstraction principles of object oriented programming. To access class fields, classes provide what are commonly referred to as *getter* and *setter* methods [6]. These methods simply load or store values in the fields of an object. By identifying these methods, a VM may be able to substitute a more streamline version into the executable.

3.1 Measuring Fine Grain Redundancy

To detect this low-level redundancy, we examined methods based on three levels of equivalence, which we will refer to as *byte-for-byte equivalence*, *resolved equivalence*, and *generic equivalence*. *Byte-for-byte equivalence* implies that the bytecode images of two Java methods are identical, including the constant pool and local variable indices. Using byte-for-byte equivalence, two methods from different class files may appear identical even though they perform different functions. This is because the same index into the constant pool of two different class files rarely resolves into identical references. The byte-for-byte equivalence check worked well on the full class file checks described in Section 2 of this paper because the constant pools were also included in the comparison. More commonly, two methods that perform exactly the same function may not appear byte-for-byte equivalent. Consider recompiling a class after adding a single constant value. If the compiler chooses to place the new value at the beginning of the constant pool, all of the indices referenced by methods will be shifted even though the values referenced have not changed. To obtain a more accurate measure of bytecode equivalence we resolve the constant pool and local variable indices before making the comparison. *Resolved equivalence* guarantees that two methods may be used interchangeably without affecting the behavior of the program. It is also possible, however, to exploit the redundancy between methods even if their resolved references are not identical. We, therefore, define a less restrictive measure of redundancy, *generic equivalence*, which can be determined by comparing the bytecode operations of two methods regardless of their operand

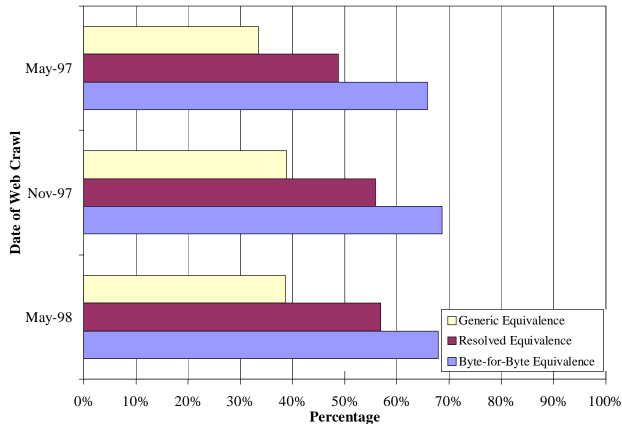

Figure 1: Percentage of unique methods utilizing different methods of comparison.

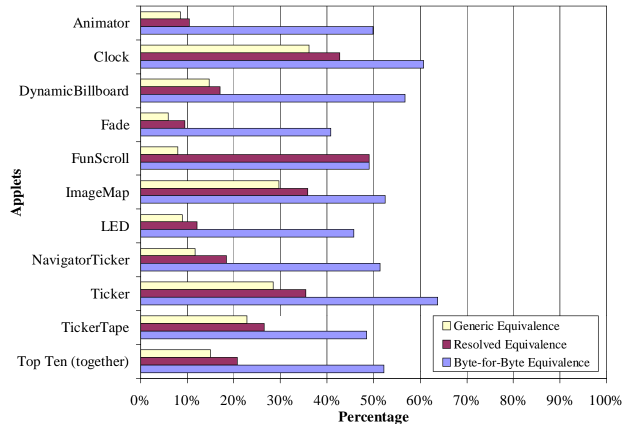

Figure 2: Method overlap in most commonly named applets from May 1997.

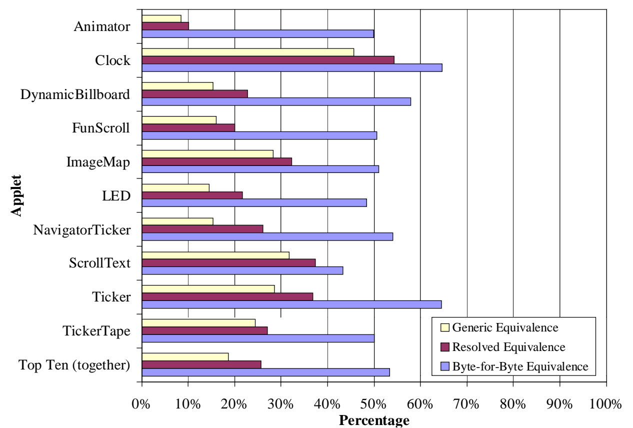

Figure 3: Method overlap in most commonly named applets from November 1997.

values. A simple technique for establishing generic equivalence involves reducing a method to a normalized form. We accomplish this by masking out the constant pool and local variable indices contained within the method's bytecode stream. Even though this technique is straightforward, we have discovered that it is extremely effective in practice. Two methods from varied sources will commonly contain the same sequence of bytecode operations but will seldom index the same constant pool value.

3.2 Method Overlap in the Web-Collected Applet Set

We measured the redundancy at the method level in the applets we had downloaded in the three runs of the enhanced web crawler. We then examined class files within the applet set we had downloaded, which was reduced from the full number of applets seen on all the pages by eliminating all byte-for-byte identical copies of the applets. We found that this reduced set of applets still contained redundant code. In fact we observed that there were a large number of redundant methods. Figure 1 shows the method-level overlap for the collected sets of applets. The bottom bar of each of the three groups shows the results of testing for byte-for-byte equivalence of the methods. Here we determined that in all three collected applet sets, less than 70% of the methods remained unique under these conditions. However, as previously stated, this is not a true measure of redundancy and can in fact produce a false overlap. We therefore also resolved the constant pool and local variable references to test for resolved equivalence. The middle bar in Figure 1 shows the results of this comparison. Here we found that an even larger set of the methods was redundant, as less than 60% of the total methods remained unique under these conditions. We next decided to look at a lower limit by comparing the methods based on generic equivalence. The top bar of each of the three applet sets in Figure 1 shows these results. Here we notice that if we could resolve these constant pool and local variable table references at runtime, we could reuse the static form of the methods and only need to retain less than 40% of the original methods in all the applets.

We also investigated the method level overlap in the applet files we had downloaded that shared the same or similar names but were not byte-for-byte equivalent. To investigate this, we chose from each of the three runs, input sets consisting of groups of applet files that varied internally but shared a common name. For example, in all three runs, we collected a large number of applet files named "ticker.class". Once we had reduced the classes sharing the name "ticker.class" by eliminating the ones that were byte-for-byte equivalent, we still had a significant number remaining. In fact we had 69 files from the May 1997 run, 63 files from the November 1997 run, and 73 files from the May 1998 run. We chose the top ten sets of files from each run that best displayed this characteristic. We again used the static form of these files to examine the overlap using all three types of method-level comparison. These results are presented in Figures 2, 3, and 4. Again, the graphs show the percentage of unique methods found using the three types of comparisons. Some of the applets exhibited behavior consistent with only a few methods having been varied. For example, in all three runs, we observed a very low variance in the Animator applet. This is shown as the bottom sets of bars in Figures 2, 3, and 4. Note that when we looked at byte-for-byte equivalence in Animator applet set, we showed that of all the method in all the files, only 50% remained unique. However, when we resolved the reference this reduced significantly such that only around 10% of all the methods in all the files, now remained unique. Even for the applets that did not show as much method overlap, what was

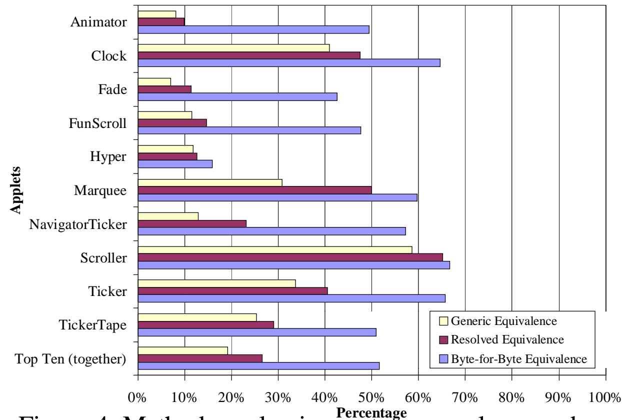

Figure 4: Method overlap in most commonly named applets from May 1998.

seen was still significant. For the most part, except for some minor exceptions, the redundancy in these sets was larger than that with all the applets combined.

3.3 Exploiting Fine Grain Redundancy

Once method redundancy has been identified, a Java Virtual Machine can benefit from the information by reusing code. Code reuse, in turn, translates into smaller space requirements and lower startup cost since the system may be able to reuse native code already in memory instead of processing another JIT request. When methods exhibit resolved equivalence they can share the same runtime memory space, and the code only needs to be loaded, linked and JIT-compiled once. Although generically equivalent methods are not as easily shared, we have developed a technique for exploiting this redundancy as well. It is only necessary to retain a single version of the generically equivalent methods on disk, as long as the locations of constant pool references in the varied bytecode forms are preserved in some structure. For example, a single generic bytecode form and tables of the specific constant pool values for each variation indexed by the method signatures or other unique tag can represent the methods. An alternative VM implementation may cache the native code rather than invoking the JIT compiler each time [7]. Because native code is typically much larger than the original bytecode, static code size becomes an even greater issue. Using our technique, however, only one copy of the native code needs to be stored for generically equivalent methods, with tables to indicate where the runtime linking needs to be performed. This can help reduce the cache size as well as loading time.

Reusing generically equivalent methods can however limit the types of optimization performed on precompiled code segments. For example, certain aggressive optimizations that occur during compilation to native code, such as loop unrolling or method inlining, may make use of the information in the constant pool. However, a simple solution exists. The generic method can be tagged so that if a bytecode method appears to match the generic form, it will be further evaluated to make certain the optimized form is used when appropriate and the generic form when not.

Another advantage of detecting method overlap is that a virtual machine can take advantage of the redundancy not only across physical locations but also across time. This shows the most benefit in systems that cache native code versions of portions of the Java programs for reuse at future invocations.

_200_check	A simple program to check the functionality of the JVM.
_201_compress	A Java port of 129.compress in the SPECint95 benchmark suite.
_202_jess	A puzzle solving expert shell system based on NASA CLIPS that progressively increases in difficulty.
_209_db	Performs multiple operations on a memory resident database.
_213_javac	Java source to bytecode compiler – scaled down from JDK 1.0.2 version.
_222_mpegaudio	An audio decompression program that operates on a 4 MB audio file.
_227_mtrt	A multithreaded raytracer.
_228_jack	A Java parser generator.

Table 2: SPECjvm98 benchmark descriptions.

When a class file on a remote location changes, the virtual machine must load, possibly recompile, and link the new version. If the virtual machine has compiled and optimized methods from a previous version of the file, it may be possible to still use these methods if it can detect that they have not changed.

3.4 Method Level Overlap for Application Benchmarks

We studied this characteristic of Java used in a wider set of programs than just the applets seen on the web pages. Because Java is not solely used for developing applets but is also used in the writing of applications, we studied this aspect as well. To characterize Java applications, we decided to use the set of programs developed, collected, tuned, and provided by the SPEC community, namely the SPECjvm98 benchmark suite [8]. Table 2 contains a brief description of each benchmark.

We performed the same static analysis on these benchmark programs to determine whether or not the method redundancy in applications is similar to that found among applets. As shown in Figure 5, the SPECjvm98 applications displayed different behavior when it came to method redundancy. Although this result contrasts with the behavior of applets, it should be expected given the nature of the benchmark programs. Each benchmark is intentionally distinct from the others and executes a very specific functionality, whereas applets often perform many similar operations. Additionally, many of the benchmarks, such as _209_db, invoke such a small number of methods, that an overlap between them is unlikely. Although individually we observed significantly less redundancy in the SPEC benchmarks than in the downloaded applets, when we examined the full suite as a set, we observed a slightly higher redundancy with generic equivalence. Still not as high as what we observed earlier, 45% of all the methods were redundant. Also interesting in the SPECjvm98 benchmark suite, we found benchmarks that clearly exemplified the problems with using byte-for-byte equivalence techniques for detecting redundancy. Looking at Figure 5 and the individual graphs for _202_jess**,** _227_mtrt, _213_javac, and even the All Spec graph, it is clear that doing an byte-for-byte equivalence, incorrectly identified methods as being the same when in fact they were not. The middle bar in Figure 5 shows that when the references were resolved in the resolved equivalence, less of a method redundancy overlap actually existed.

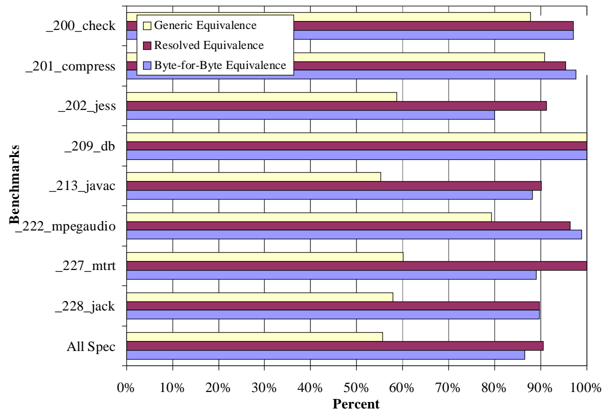

Figure 5: Method level redundancy in SPECjvm98.

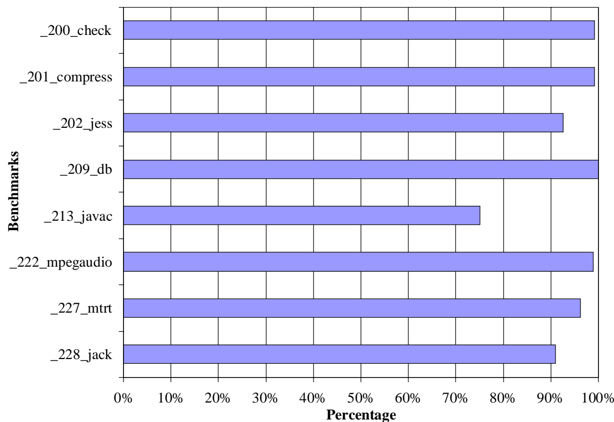

Figure 6: Percentage of generically unique code to full bytecode in SPECjvm98.

Another interesting feature of the SPECjvm98 suite was it appeared to show a correlation between the amount of code reduction that can be achieved and the size of the methods. We found that smaller methods tend to exhibit greater redundancy characteristics therefore affecting the amount of actual code reduction that can be achieved. For each of the SPECjvm98 benchmarks, Figure 6 shows the percentage of total bytecode size that is represented by retaining only the generically unique methods. It is interesting to note that larger benchmarks, such as _213_javac, tend to contain more redundant code while at the other extreme, small benchmarks such as _209_db show no redundancy.

3.5 Dynamic Analysis of Fine Grain Redundancy

Analyzing the static code allows us to efficiently characterize a very large set of bytecode. Static analysis, however, may capture extraneous data. For example, an application may only use a small percentage of the methods actually contained in the file, and this approach would include even those methods never invoked. Also, since the core Java libraries are needed for a working Java runtime environment, we restricted the input set of our static analysis to only the code that is distributed with the applet or application. However, in doing so, we did not reflect methods from the core Java libraries that may be called by the program and may also exhibit redundant characteristics. Therefore, we performed a dynamic analysis to isolate the methods that are important during execution. Because our focus was on the possibility of reusing previously loaded or compiled methods, we decided to narrow our scope by only looking at those methods for which the VM requests a Just-In-Time (JIT) compiled version. To monitor JIT requests we used Microsoft's Java virtual machine 2.02 [9] and replaced the DLL that services JIT requests with our own DLL. Our DLL was built as a slim layer between the Microsoft VM and the Microsoft JIT. It simply records statistics on the requested methods and passes control to the real JIT.

Benchmark	Total Number of JIT Requests	Byte-for-Byte Unique	Resolved Unique	Generic-ally Unique
_200_check	391	362	353	309
_201_compress	326	313	293	269
_202_jess	759	619	674	466
_209_db	339	326	310	280
_213_javac	1113	1062	1039	912
_222_mpegaudio	496	478	453	396
_227_mtrt	464	434	432	342
_228_jack	572	532	518	393
Sum of Above	4460	4126	4072	3367
All Spec	2542	2100	2173	1632
Applets	5040	4445	4378	3482

Table 3: Dynamic method redundancy.

To handle the realm of the applet benchmarks, we chose a smaller set of applets that we felt characterized the type of applets we had seen. Since we had already explored the overlap in applets of the same or similar name, we judiciously selected a set of applets that were both diverse in type and author. This set also included the source level code for the applets allowing us to confirm hypothesis all the way back to the source. The accumulated set ranges in scope from trivial banners to business and educational software applets. We altered some of them to make them self-triggering using provided sample inputs. This set is being made available to fellow researchers, and the applets can be download from the IMPACT web page (http://www.crhc.uiuc.edu/IMPACT/).

The results of our study are presented at a very high level in Table 3. The first eight rows summarize the JIT requests that occurred while running each of the SPECjvm98 benchmark applications in test mode with the largest input (control-set 100). These were run using the command line option for SPECjvm98 to reduce the overhead of the GUI SPECjvm98 harness. They do still contain some harness functions, but eliminating the GUI minimized this overhead. The row labeled "All Spec" lists JIT requests from a single run of the entire SPECjvm98 benchmark suite again using the command line option. Consequently, any methods that are redundant across benchmarks are detected. This is seen most clearly if we sum the top eight rows and compare the sums to the results of the full run. For the overlap, we found that about 280 methods were seen in all the benchmarks. Of these, approximately 80 in each benchmark run were SPEC harness specific, and another 200 were system specific. Although all of these were called individually in each specific run, the full run called many of them only once. This is the primary reason for the discrepancy in the summed total of the eight benchmarks and the recorded total of a full SPECjvm98 run. The last row shows the JIT requests from executing each applet in the benchmark set exactly once during a single virtual

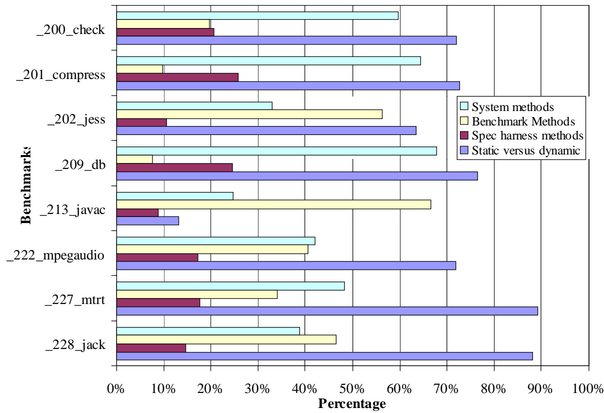

Figure 7: Relative percentages of dynamic methods.

machine invocation. Column two of Table 3 shows the number of methods requested by the JIT that were unique using byte-for-byte equivalence. As pointed out previously, a byte-for-byte equivalence does not guarantee that the methods are identical due to differences in constant pool resolution. Column three shows the number of unique methods after checking for resolved equivalence. These methods can be used interchangeably at runtime, even after they have been compiled to native code and optimized. An intelligent virtual machine can therefore reduce its memory footprint by sharing a single version of the resolved equivalent methods. In column four, we list the number of unique methods found using the test for generic equivalence as previously described. A comparison between columns three and four reveals that our technique of masking out constant pool indices exposes a much larger set of redundant methods. One item we found interesting is that the amount of redundancy is similar for both the applet and application code. For applets, the number of generically unique methods is 69% of the number of JIT requests. In other words, 31% of the methods requested by applets were redundant at this level. For the combined SPECjvm98 application code, 36% percent of the methods were redundant. As mentioned earlier, aggressive performance optimizations may interfere with code reuse between generically equivalent methods.

Figure 7 gives a breakdown of the methods called in each of the SPECjvm98 benchmarks. The bottom bar for each of the benchmarks in Figure 7 gives the percentage of static methods measured and reported in Section 3.4 that were actually JIT-compiled in the dynamic run. As explained at the start of this section, not all the methods we analyzed statically are actually used during a full run. As seen in Figure 7, this ranges from only 13% of the static methods in _213_javac being used to 89% of the static methods in _227_mtrt. Therefore when making a decision on which methods to keep in a reusable optimized form that could be used across benchmarks (like the object initializer suggested in introduction to Section 3), the JIT-compiled set of methods from dynamic runs of the program should be used. However, when looking at a reduction in static code size (perhaps by storing a generic form and using a mapping like that suggested in Section 3.3), the static version of the files may still be used.

The third bar for each benchmark in Figure 7 is the percentage of total methods JIT-compiled that result from the overhead of the SPECjvm98 harness. The second bar is the percentage of JIT-compiled methods that are benchmark specific.

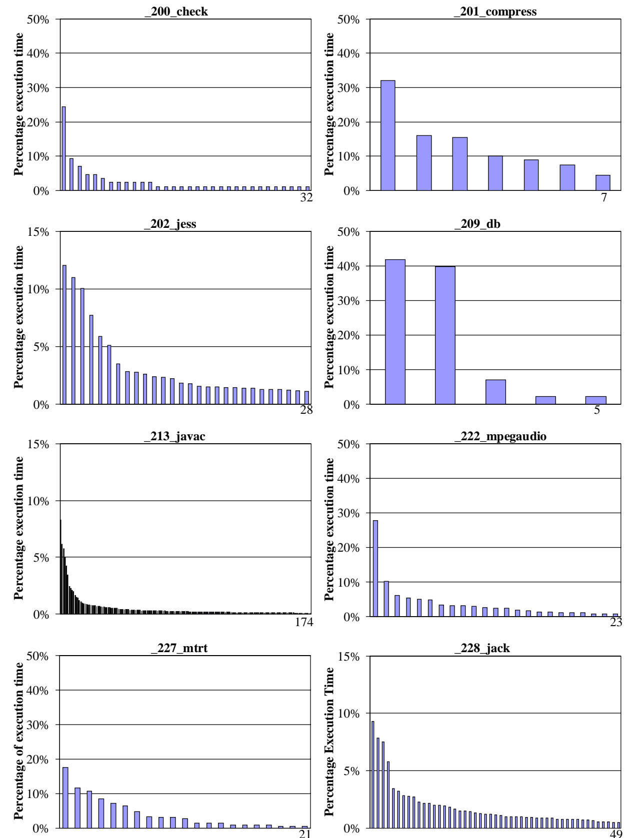

Figure 8: Methods that comprise 90% of the execution time in each SPECjvm98 benchmark.

This ranges from 8% in _209_db to 67% in _213_javac. Finally, the top bar in Figure 7 lists the percentage of system methods. The actual number of these was relatively consistent across all the benchmarks, ranging in size from 209 for _209_mjpegaudio to 275 for _213_javac. The smaller benchmarks show the largest impact from system level methods.

To determine whether or not other optimizations would impact code reuse, we expanded our study to measure the percentage of execution time actually spent in each method. To obtain an accurate profile of the benchmarks and applets, we used Intel's VTune v3.0 [10]. For each of the eight SPECjvm98 benchmarks, Figure 8 shows the percentage of execution time spent in the methods that together comprise 90% of the total execution time. The graphs for _202_jess, _213_javac, and _228_jack are on a 0-15% scale because they both access a large and varied number of methods, each of which accounts for less than 15% of the total execution time. The other graphs are shown on a 0-50% scale.

The methods comprising 90% of the execution time are usually those chosen for performance optimization. However, utilizing redundancy properties, we may be able to identify other opportunities. For example, as pointed out in Section 3.3, additional information on redundancy, especially information on resolved equivalence can expose opportunities for more efficient runtime memory usage. If we can isolate the resolved equivalent methods from those we would like to aggressively optimize (e.g., inline), we can share the same piece of native code across different bytecode files or even methods within the same file.

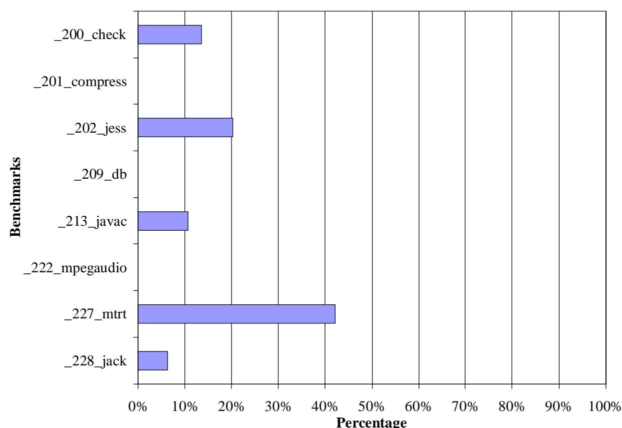

Figure 9: Percentage execution time in methods that are redundant based on generic equivalence.

To investigate this effect further, we compared the intersection of the set of methods comprising the 90% execution time set with the generically equivalent methods seen across the JIT interface. Noting that the generically equivalent set also covers generic forms of the resolved equivalence set, we looked at generically equivalent methods first. We further restricted the generically equivalent method input set to methods that had been used more than once in each of the benchmarks. This allowed us to investigate the effects on individual programs versus the diverse benchmark suite. The results of our investigation are shown in Figure 9. Note that in _201_compress, _209_db, and _222_mpegaudio, the generically equivalent methods do not intersect with the 90% execution time methods. However, we may still benefit from the resolved equivalence information concerning these benchmarks. For example, in Table 4 we show a sampling of the resolved equivalence overlap for each of the benchmarks as well as for the whole benchmark suite. In all eight of the benchmarks, we found overlapped calls to a resolved equivalent initializer that simply called the initializer for the root class, object. In fact, in these three benchmarks where the generically unique method set and the 90% execution time set are disjoint, there is a total of 25 calls to this resolved equivalent initializer. Therefore only a single native version of the method is needed to handle all the calls. This becomes an even more dramatic savings when multiple applications are running at the same time. As seen in Table 4, the All Spec run showed that this resolved equivalent initializer was used 120 times. By identifying other such areas of redundancy, a set of native code versions of resolved equivalent methods could be used as a dynamically linked library set for all Java programs.

At the other end of the spectrum, however, is _227_mtrt. As seen in Figure 9, we found that over 42% of the total execution time overlapped with the generically equivalent set of methods. Upon further investigation, we discovered that of the nine methods that comprised this set, the top seven of them were getter methods described in Section 3. These methods are simply an artifact of the data abstraction principles of object oriented programming and used to retrieve the value of a private data field in a class instance. These seven methods account for 40.6% of the total execution time of this benchmark, and therefore recognizing these and inlining the field accesses would eliminate the calling overhead and improve the performance. Although getter methods show a high generic equivalence, they show a poor resolved equivalence because they rarely resolve to the same field. This means that, by considering both the type of method and its resolved equivalence behavior, we can recognize

Benchmark	Resolved equivalent methods used more than once	Average number of times used	Resolved equivalent object initializer
_200_check	23	3.29	10
_201_compress	11	2.75	6
_202_jess	69	17.25	56
_209_db	13	3.25	6
_213_javac	67	3.94	11
_222_mpegaudio	21	4.2	13
_227_mtrt	15	3.75	8
_228_jack	53	3.31	12
All Spec	1652	6.14	120
Applets	476	3.09	30

Table 4: Dynamic resolved equivalence behavior.

methods that do not benefit from the dynamically linked and shared library strategy and therefore are more promising candidates for inlining. As seen from this brief discussion, a careful investigation must be made of the actual methods and their usage before making optimization decisions.

4. Conclusions

This research has found that Java programs share a large segment of their application space. Designing and utilizing systems that can exploit these traits may enable performance improvements in future Java systems. Our work has indicated the potential opportunities for reducing loading, linking, and compilation delays, system code cache sizes, and unnecessary invalidation of previous compiled code. We have also suggested how this information can be used to guide decisions on which portions of a Java program will gain the most benefit from maintaining statically compiled and which will benefit from optimized native versions. This study was directed at locating general areas of performance improvement within the Java programming paradigm. We have discussed some of the tradeoffs and costs involved in exploiting these properties and we intend to continue our investigation into other opportunities for improvement of Java performance in the future.

5. Acknowledgements

The authors would like to thank Andrew Hsieh, Matthew Merten, and all the members of the IMPACT compiler team for their support, comments, and suggestions. This research has been supported by Hewlett-Packard and the National Science Foundation (NSF) under grant CCR-9629948. The authors would also like to thank Microsoft and Intel for their donation of some of software tools used in this study.

References

[1] Gosling, J., Joy, B., and Steele, G., *The Java Language Specification,* Addison Wesley, Reading, MA. 1996.

[2] Venners, B., *Inside the Java Virtual Machine,* McGraw Hill, New York, NY, 1998.

[3] *JAR - Java Archive*, Sun Microsystems, Inc., Mtn. View, CA, http://java.sun.com/products/jdk/1.1/docs/guide/jar/index.html

[4] Rivest, R., *The MD5 Message-Digest Algorithm,* MIT Laboratory for Computer Science, and RSA Data Security, Inc. Request for Comments: 1321, April 1992.

[5] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T., *Hypertext Transfer Protocol -- HTTP/1.1,* Request for Comments: 2068, UC Irvine, DEC., MIT/LCS, January 1997.

[6] Eckel, B. *Thinking in Java,* Prentice Hall PTR, Upper Saddle River, NJ, 1998.

[7] Hsieh, C.-H., Conte, M., Johnson, T., Gyllenhaal, J., and Hwu, W., *Optimizing NET Compilers for Improved Java Performance*, IEEE Computer, June 1997, pp. 67-75.
[8] *SPEC JVM Client98 Suite*, Standard Performance Evaluation Corporation http://www.spec.org/osg/jvm98/ , Release 1.0 8/98.
[9] *Microsoft Java Virtual Machine version 2.02,* Microsoft Corp. Redmond, WA. http://www.microsoft.com/java/vm/dl_vmsp2.htm
[10] *VTune™ Performance Enhancement Environment,* Intel Corp. Santa Clara, CA http://www.intel.com/design/perftool/vtune/