A CHARACTERIZATION OF CODE REUSE WITHIN JAVA APPLETS AND APPLICATIONS

BY

MARIE THERESE CONTE

B.E.E., University of Delaware, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ACKNOWLEDGMENTS

I thank my advisor, Professor Wen-mei Hwu, for his guidance and direction in helping me focus my research efforts. I thank him also for giving me the opportunity to work with a wonderful group.

I wish to thank the entire IMPACT research group for providing an excellent infrastructure in which to do this research. In particular, John Gyllenhaal served as my mentor throughout this research, helping me overcome obstacles that arose. Cheng-Hsueh "Andrew" Hsieh, Andrew Trick, and Richard Kutter provided feedback and help in the design and implementation of this research. Mathew Merten provided help in editing and organizing the writing of this thesis.

Finally, I wish to thank my brother, who provided both counsel and encouragement through difficult periods, and my parents for their love and support throughout both my undergraduate and graduate studies. Without them, none of this would have been possible.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1 Java and the Web

In 1995 Sun Microsystems introduced a form of machine independent binaries that allow executables to travel over the Internet and execute on any host machine containing the needed interface. This new language entitled Java, swiftly found a home in the entangled Internet Web community. These small applications, referred to as *applets,* added more traffic to the already bloated network as people rushed to add the machine-independent components to their Web pages. Businesses, not wishing to have their Web pages to appear dated added some form of Java applet to them [1], [2].

One of the properties that enabled Java's swift acceptance is its use of a universal binary distribution format called *bytecode*. Among the main principles behind bytecode is that the code is designed to run on a fictitious machine referred to as a *virtual machine* (VM). This means that the bytecode can execute on any machine implementing the Java Virtual Machine (JVM) specifications regardless of the underlying hardware [3]. By the start of 1998, the JVM had been ported to everything from the processors used in the embedded devices market to IBM mainframes [4]. However, this was not the only feature that drove Java's swift adoption.

Java was another evolutionary step in the migration from the more classic monolithic program style to a more modular distributed application spanning both resources and distance. Utilizing a set of class files as parts of a larger program, these modules were designed to fit easily into multiple running applications and be swapped with more current version when they were needed. Enabling this swapping of modules is the Java feature requiring that all class files be resolved at run time versus the compile time resolution used for the more traditional monolithic programming style. Additionally the seamless integration of class file changes meant that individual components of a program could evolve with less chance of breaking a previous implementation [5].

Many of Java's features also helped with the maturation of the World Wide Web. Companies feeling their way in the new communication and commerce medium, found that, with Java applets, the graphics used on their Web pages could now come alive with motion and sound via actual executing programs. The introduction of multimedia presentation capabilities allowed businesses the marketing tools normally available only on more traditional mediums such as TV. Java also introduced a security infrastructure making it safe to allow outsiders access to executable content on a company server. The “sandbox” security model implemented in Java, offered the flexibility of hardware access restrictions to be as limited or open as company needs dictated. Also, Java introduced an encryption and signature framework that was easy to implement, making the security features needed for Internet commerce comprehensible and reliable. The incorporation of unicode as the text format, allowed for a larger character set thus encompassing the alphabets of most languages. With these and other features, Java enabled company Web pages to swiftly integrate and facilitate Web based commerce that encompassed not only the domestic market but also allowed for inclusion in an international marketplace [6]-[10].

However, such enhancements do not come without costs. Transporting the additional graphics and executables needed to provide the new level of interfaces supported by today’s computers requires additional bandwidth. This, coupled with the increased use of the Internet, introduces additional delays that could discourage the continued growth of the industry. Plans to increase the bandwidth of the Internet backbone are in the works but are still years away [11], and as history has shown, usage patterns will most likely expand the demand on it to fill any additional bandwidth. Even with the increased bandwidth, however, many users are still restricted to the 56-K baud of a modem or the 64- to 128-K bit rate of an ISDN line for Internet access. Even in business and research environments where many of the intercomputer transactions take place over a local area network (LAN), the new demand to transport the graphical interface along with the information has taxed even these higher speed connections.

More importantly, when accessing information from a remote computer, ignoring several lessons of the past can have detrimental effect. Although some caching is used in Web servers and in most client-based Web browsers in use today, these appear only to maintain recent Web page information and restrict reuse to physically identical access localities [11], [12]. Currently, no attempt is made to check for overlapping components between two physically different pages. Therefore most Web servers and browsers are not truly capturing the full reuse of components and efficiently utilizing the information they have stored in their caches. One area where the potential reuse of components across Web pages is most noticeable is with these Java applets. As the use of Java applets to enhance Web pages has matured, the number of unique applets have begun to converge onto into a smaller overlapping, heavily used subset. Although some of this overlap can be attributed to accessing applets via a centralized location within a company or institution, there is still a large percentage of applets that are copied to physically different locations, making their overlap undetectable by current caching algorithms.

1.2 Java and Software Vendors

The applicability of Java as the medium to solve problems with dynamic software as well as heterogeneous hardware also sparked interest in the computer software industry. Switching to Java supplied a new level of uniformity across platforms. For example, there was now only one thread model to learn and program for versus the headache of dealing with the multiple flavors of Posix threads on UNIX systems and the Microsoft thread and fiber model on Windows-based systems. In addition, the network socket model had also been simplified. Now there is one application programming interface (API), and it would work on UNIX and Windows-based systems. The simplified component model adopted in the Java specification also enables greater opportunities for the reuse of code segments across multiple applications [13]. In addition, the difference between interfaces to shared libraries on different systems, for example the .sl and a dynamically linked library (DLL) interface, were now also removed. All of the run-time loading and linking would be handled by the Java run-time environment (JRE). Because the JRE is provided by the hardware/operating-system (OS) manufacturer, the need for performance on a given system could also be pushed back to them. The software developer no longer needed to worry about imbedding an assembly level section into critical sections of the program in order to gain performance. Getting performance out of the underlying hardware was now the responsibility of the JRE interface. Because Java's dynamic loading and linking features allowed for the run-time customization of an application, Java caught the interest of database manufacturers. These developers saw a way to scale their product, as well as making it more responsive the user's needs. Because Java linked and loaded classes as needed, the search engine could be sent with the request customizing each search to fit the data instead of incorporating all possible searches into one large monolithic program [14]-[16].

Although all of these features helped Java's acceptance, Java still suffered from one fundamental problem, performance. Because Java is compiled into bytecode, which is targeted to run on a virtual machine, something was needed to translate the bytecode to the proper binary needed for the underlying hardware. Initially implemented with interpreters, the JREs soon encompassed a compiler that would take the bytecode to a native binary fast and only when needed. These compilers produce the native code swiftly when the code segment is executed the first time [17]. Called just-in-time (JIT) compilers, the native code produced (JITed) by them disappeared when the Java application finished execution. This production of native binaries at run time inhibited the level of optimizations that could be applied to the code, meaning that the quality of the code produced was inferior to what could be produced by a static code compiler that was not under these severe time constraints.

Combining the benefits enjoyed by static compilers and support for the dynamic specifications and features of the Java language [18] should provide an ideal solution. Native code as well as bytecode reuse across invocations appeared to be a natural evolutionary step for the JRE. However, the JREs ignored code reuse across applications both at a class file level and at a method level. Class files were located and reloaded when needed even if they had not changed since the last time they were used. Also, methods were re-JITed even if they were identical to a method JITed earlier. If opportunities for code reuse could be exposed, then opportunities for producing better code quality could be explored.

In this thesis, I present several tools designed and developed to expose the overlap within Java programs. I further show the results of tests conducted on a collected set of actual Java applets collected from the Internet over the space of a year to discover and expose different layers of redundancy. In addition, I present a description of a working interface capable of inserting static native versions of Java code when available in place of using a JIT to generate the native code. I also give examples of tested static native code that was successfully inserted in place of the normal JREs code generation facilities. This research sets the stage for off-line code optimization that would enable performance improvements in the JRE.

2. CLASS FILE LEVEL REDUNDANCY

The time span from when Java was first introduced until its widespread acceptance and appearance on Web pages, was relatively short. Within a year of its introduction, a noticeable percentage of publicly accessible Web pages had been enhanced with everything from text scrollers to interactive games. On the surface, these applets exhibited a similar look and feel to each other, which suggested they also shared, similar if not the same code base. Investigation of this observation showed an overlap in the actual applet files, that could be exploited to improve the performance of an existing JRE. By recognizing applet class files already on the local system and eliminating the download delay of reobtaining these files, the JRE could improve start-up and response time.

2.1 Description of Investigative Tools

To study the overlap of Java applets on physically different Web pages, an input set of reasonable size needed to be acquired. Visiting a large quantity of Web pages and culling out the subset that was Java enhanced proved the most efficient means of collecting this set. One way of accomplishing this is to start with a set of seed Web pages and crawl across the Web, expanding links found on accessed pages and taking care not to revisit sites in an endless loop. Using this solution, a modified Web crawler was developed. This crawler would look specifically for Java-enhanced pages, taking care not to revisit an already visited page. Figure 2.1 shows the basic algorithm used for the Web crawler in a Java style of pseudocode. Basically, the Web crawler visited a page, checking it for any Java applet tags used to denote the presence of an applet. If such a tag was found, it was recorded in a master list of Java-enhanced pages. Also, any links to other pages that existed on the page currently being visited were checked to see if they had been visited. If not, they were added to the list of pages to visit in the future. The Web crawler used a queue-type structure to implement the list. This enforced a breadth first type of search pattern.

```
while pageList != empty do
      if pagesOnTier ! = empty then
            crntPage = nextPageOnTier
      else
            crntTier = nextTier
            crntPage = firstPageOnTier
      page = download(crntPage)
      if checkForAppletTags(page) == True then
            appletList += crntPage
      if checkForWebLinks(page) == True then
            for(crntLink = firstLink; moreLinks; crntLink = nextLink)
       if notOnVisitedList(crntLink) == True  then
                        addToTierList(crntLink)
      addToVisitedList(crntPage)
end while
```

Figure 2.1 Algorithm for Web crawler

Once the accumulated set of Java enhanced pages was a large enough for analysis, the applets were downloaded. Figure 2.2 shows the fields in an example HTML applet specification taken from the *http://www.jars.com/index.html* (204.74.88.17) in May 1998. The applet specification example given in Figure 2.2 was chosen because it illustrates several aspects concerning the location of an applet referenced on a Web page. The `applet` tag denotes the existence of an applet on a Web page. This tag is immediately followed by information on how to locate the applet, as well as where and how it should appear on the page. The fields are the parameter-initialization fields and are denoted by the `param` tag. These `param` fields allow the author of the Web page to initialize parameters available within the applet thus customizing the look, feel, and behavior of the applet to best fit their usage of it. The location of the actual applet code, as specified by the “`codebase`” field. In Figure 2.2, this field shows that the actual applet code resides in a physically different location than the actual page using it. In fact, in this example, the code resides at the Web site on another network, namely Guestplanet. Additionally, looking at the `code` field in Figure 2.2, additional path information exists. This path information is denoted by the dot separator, used to denote paths in the Java specifications, allowing the path specifier to be architecturally independent. By concatenating these two fields, the applet in this example actually resides at *http://www.guestplanet.com/classes/com/earthweb/guestbook* (209.242.68.54) and is named GB5. When the actual applets were collected from the pages, this information was maintained in tables similar to those used by the Web crawler. This information was further analyzed to determine the amount of uniqueness in the applets based on their physical locations versus the pages accessing them.

```
<applet code="com.earthweb.guestbook.GB5"
      codebase="http://www.guestplanet.com/classes/"
      align="baseline"
      width="165"
      height="30"
      archive="gb5.zip">
<param name="cabbase" value="gb5.cab">
<param name="BUTTON" value="Sign JARS Guestbook!">
<param name="TITLE" value="JARS Guestbook">
<param name="FILENAME" value="gb000002.html">
<param name="MANDATORY_ID" value="YES">
</applet>
```

Figure 2.2 Sample HTML Applet Specification

2.2 Exposed Class File Level Redundancy

The first generation of the Web crawler was not perfect and missed some types of link specifiers. However, the Web crawler did allow us to collect a large enough sample set to study. An improved Web crawler was later used to collect a larger set of pages.

This new and improved crawler was run 6 months after the initial one and allowed us to test the validity of some early conclusions presented here. One year after the initial launch of the Web crawler, the improved version was launched again to collect another sample set. The first run of this Web crawler was for several days in May 1997, accumulating a set of 1616 Java-enhanced Web pages. The improved version of the crawler was run in November of 1997 collecting a set of 5198 Java-enhanced pages, and again in May of 1998 collecting a set of 4316 Java-enhanced pages.

Table 2.1 Web crawling results

Collected	May 1997	November 1997	May 1998
Number of Pages	1616	5198	4316
Number of Applets	1939	7027	7194
Unique (physical location)	1465	1721	2959
Unique (physical code)	786	976	1595

The results of the Web crawling experiments were analyzed further and are recapped in Table 2.1. This table contains some rather interesting observations used to motivate the next step in this tool set development. In Table 2.1, row 1 lists the number of Java-enhanced pages visited or the number of Web pages among the total visited that were found to contain Java applets. Although differences in the number of pages collected between May 1997 and November 1997 are partially attributable to improvements made to the Web Crawling, part of the increase can also be attributed to improvements in the Web browsers. These improvements make running of Java applets smoother thus encouraging wider usage of applets on Web pages. The number of applets seen in row 2 of Table 2.1, is a total count of all applets seen on the pages listed in row 1. Note that on all runs, the average number of applets per page exceeded one. Also, this value has actually increased over time. The May 1997 collection showed the average number of applets per page at 1.20, whereas by November of that year, it had risen to 1.35. Within a year, however, this average had risen even further to the 1.67 seen in May 1998.

The third row in Table 2.1 lists the number of unique applets if the `codebase` tag and `code` tag are used to locate the file. Examples of these fields were shown in Figure 2.2, and an example of how these fields are expanded and the applet located was given earlier. Note that the Internet protocol (IP) addresses of the Web addresses specified in this field were used to determine uniqueness. Observing that although there were a large number of pages that contained Java applets, on average, almost 50% of them referred to the same physically located code. Although it is true that most current caching algorithms used in most browsers are capable of detecting this redundancy [11], [12] and reducing this overhead, they still do not detect the full redundancy in the applet sets. Next downloading the applets from the physically different locations, they were compared on a byte-for-byte basis. Noticing that several of these applets were identical, further analysis was performed. The last row in Table 2.1 shows this overlap. This level of redundancy is probably attributable to the copying of applets to physically different locations. Almost 30% of the applets identified as unique by physical location overlapped when actually downloaded.

If a class loader can detect identical class files, it can avoid transferring these 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* [3]. 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 PCs 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 [19]. Initial tests with MD5 fingerprinting showed that it successfully detects the full overlap between the files. The algorithm is widely available and has recently been included in the specification of HTTP version 1.1 [20]. MD5 produces a 128-bit result that fits into a single IP packet requiring no more than the delay simple, single packet location protocol such as *ping*, to obtain. The measured computing time of the MD5 fingerprint on the class files collected in the applet sets was an average of 6 µs. This time was measured on a Pentium 200-MHz machine with 64 Mbyte of memory. The measured average ping delay a to the servers in the applet sets as 140 ms using a 10-Mbyte Ethernet line connecting through a T3 line from the network router. For a 33.6 kbaud modem, this delay was measured at 330 ms. 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 it is indeed an identical class.

3. METHOD LEVEL REDUNDANCY

One thing that stood out when looking at the files collected from the Web crawler runs was the number of class files found that shared similar or the same name but had differing downloaded bytecode. Noting that in Java minor variations (such as the compiler used, inclusion of a new field, rearranging of existing fields, changes in either the filename or the name of files used by it) could cause the files to appear different when they actually behaved and ran the same. Theorizing that the majority of the difference seen in the collected class file sets may be minor, a closer examination the files was conducted at a method level.

3.1 Description of Investigative Techniques and Tools

Minor differences in class files can exist for many reasons. The most trivial type of difference 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 show similarities within certain methods. For example, observations of the collected data found it to be common for a class initializer to call the initializer of the base class. Moreover, the analysis of static code, this observation showed that on average, more than 50% of these initializers resolve to root class, *Object*. A VM may want to keep an optimized class object 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 [5]. 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 streamlined version into the executable.

3.1.1 Criteria used for comparison

To compare the files, three different criteria were used. The first comparison technique used for comparing two methods determines if they are byte-for-byte matches. This comparison required the methods being compared to be identical at a byte level meaning every byte in the method matches with every byte in the other method. Chosen initially because this was the way the full class files had been compared at download, its applicability to this new problem set was questionable. Although this method sounds viable, it actually fails in some simple cases. Note that the majority of Java bytecode instructions that take arguments, take reference values that are represented as indices into a global class file level table called the constant pool or a local variable table which is a run-time constructed from information stored by the compiler [21]. Therefore, variations in either of these structures will cause a discrepancy when comparing for byte-for-byte equivalence. For example, consider an instance of the class Bar that has the field foo1. There are two bytecode instructions used to access nonstatic fields in class instances, `getfield` and `putfield`. These instructions take as their argument an index into the constant pool of the class file from which they are called. For example, if the `getfield` instruction in the example were used in a method from class *FooyBar*, it would contain an index into the constant pool of class *FooyBar*. Figure 3.1 gives a diagram of how a `getfield` call from the method in *FooyBar* would be resolved using *FooyBar's* constant pool. First, the argument to the `getfield` instruction, `0x0024`, shown in box 1 of Figure 3.1, is used to locate a constant pool entry in *FooyBar's* constant pool. The entry at that index should have a tag field denoting it as type `CONSTANT_Fieldref`, as shown in box 2 in Figure 3.1. The fields of this constant pool entry are used to track down the class that the field belongs to, boxes 3 and 4 in Figure 3.1, and the name and descriptor of the field, boxes 5, 6, and 7 in Figure 3.1. Once the class has been located, the name and descriptor of the field are used to search the class's field table for the field. Therefore it is possible for two `getfield` calls in two different files to have different constant pool indices that resolve out to the same field in the same class. The converse is also true. This can also happen with other instructions in the bytecode stream. Therefore, to more fully evaluate whether two methods were the same, modifications were made to the equivalence testing to take into account what the constant pool indices point to.

With the shortcomings of the byte-for-byte equivalence technique in mind, the next form of comparison used resolves the constant pool entries before the comparison is made. This comparison technique is called *constant pool resolution equivalence*. To accomplish this, a string was constructed consisting of the resolved values for the fields, replacing the index value in the field. Using the example given in Figure 3.1, the `getfield 0x0024` instruction, which in the bytecode stream would appear as `0xb40024`, would be replaced with the string `b4Barfoo1I`. Similar replacements were made in the byetcode stream for the methods wherever a constant pool value occurred. Once the replacements had been made, the methods were compared. This approach does help identify class files that appeared byte-for-byte equivalent in the first comparison method but were in actuality different. It also helps identify files that appeared different due to constant pool index differences but the resolved reference is the same.

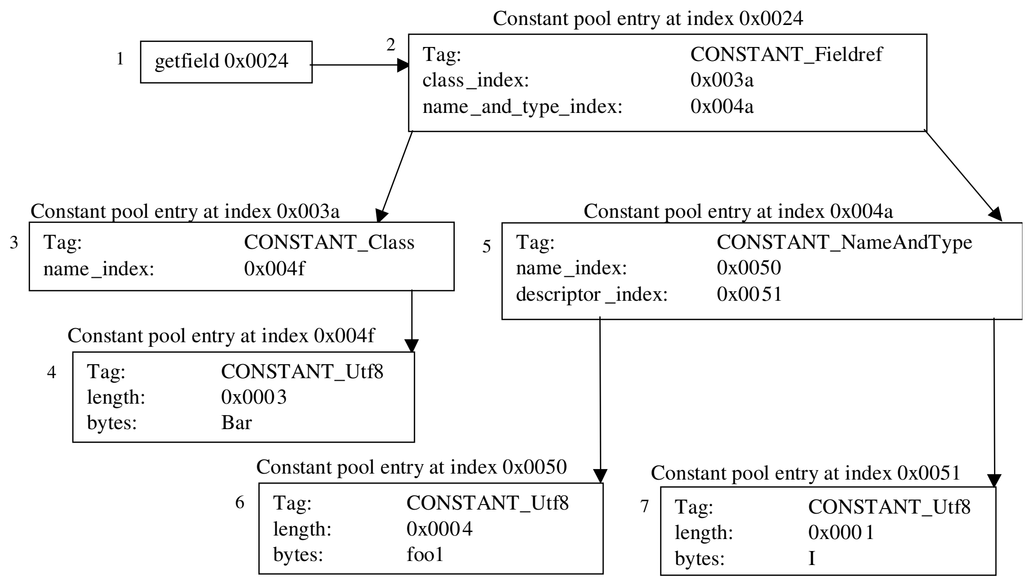

Figure 3.1 Sample resolution of the constant pool information accessed via a `getfield`

The final method used masked out all indices, not just the constant pool indices but also the local variable indices to form what can be viewed as a generic form of the bytecode stream and is referred to as *generic redundancy*. The reason for masking out local variable indices was to mask out any differences in the location of the variables in the local variable table. For example, although Java does not require a method to statically maintain a local variable table, the Java source to bytecode compiler does decide which local variables to place where in the local variable table. The Java specifications also do require a unique local variable table location for each local variable used in a method. This allows a Java source to bytecode compiler to overlap variables of the same type in the table. The run time only needs to recover the size the compiler used for the local variable table to first allocate enough space for it and second make sure the method does not expand beyond it. This information is provided in the structure used to represent a method within a bytecode file. The verification process for a method tracks the local variable table usage to ensure that the type restriction is not violated [21].

Therefore it is possible for a method to be source level identical and yet compile into different bytecode images simply because a different compiler was used. For example, let us consider method *foo()* which has among its local variables *i* and *j* of type `int` and *f* of type `float`. Now if bytecode compiler *A* choose to place local variable *i* at the local variable table location 4, *j* at local variable table location 5, and *f* at local variable location 6, the instructions for loading and storing these variables would use indices into these values. However, compiler *B* notices that local variable *i* is dead before local variable *j* comes live. Therefore compiler *B* decides to overlap these two in the local variable table at location 4, placing local variable *f* at location 5. Now when comparing the two bytecode versions of *foo()*, even though the two methods are indeed identical, the use of two different compilers will make them appear different.

Java bytecode has several instructions for loading and storing local variables. These instructions, recapped in Table 3.1, denote the type of local variable by the instruction name. Also, there are four variations that do not take arguments but rather imply the offset by their name. These four variations, given in columns 2 and 3 of Table 3.1, denote the location of the local variable by the value associated with <n>. For example, `aload_1` pushes the reference variable at local variable table location 1 onto the operand stack. In creating a generic form, only the type of the variable, the operation being performed (load or store), and its relative location within the bytecode stream was of interest. Therefore, instances of the bytecode instructions in columns 2 and 3 were replaced with their general forms in columns 4 and 5. The argument field was also removed from the previous substitution as well as instances of the instructions represented by the sets in columns 4 and 5 of Table 3.1. Constant pool values are removed in a similar manner, leaving only the bytecode instructions. By extracting out the points of possible variation between two methods, a lower bound on the number of unique methods, or an upper bound on the amount of redundancy that can be exploited was obtained. Therefore, of the three methods used, the generic equivalence method will always show the smallest set of unique methods on any given input set.

Table 3.1 Overview of Local Variable Accessing Instructions

Type	Load from implied location $n = 0, 1, 2, 3$	Store to implied location $n = 0, 1, 2, 3$	Load from specified location	Store to specified location
int	`iload_<n>`	`istore_<n>`	`iload xx`	`istore xx`
long	`lload_<n>`	`lstore_<n>`	`lload xx`	`lstore xx`
float	`fload_<n>`	`fstore_<n>`	`fload xx`	`fstore xx`
double	`dload_<n>`	`dstore_<n>`	`dload xx`	`dstore xx`
reference	`aload_<n>`	`astore_<n>`	`aload xx`	`astore xx`

To illustrate the effects of using the different equivalence tests on the methods, a simple method from the ticker.class files collected in the November 1997 run is used. The code for this simple method is shown in Figure 3.2. The bytecode stream for this method would look like `0x2a2bb6xxxxb1` where *X*s have been used in place of the constant pool reference. Note that there is only one constant pool index in the method. In Table 3.2, the variations of this method that appeared identical in a generic comparison are shown. Note that the method identified under the generic comparison guidelines has several names denoted in column 1 by appending the classfile name to the front of the method name and separating the two names with a colon. The next column lists the actual constant pool indices and what they actually resolved to. The resolved value contains the method name, argument types, and the return type for the method. These arguments are actually stored in links that follow a series of links through the constant pool similar to the `getfield` resolution shown in Figure 3.1, page 17. The argument and return type of the method are located in a string value store in a constant pool field similar to that in box 7 of Figure 3.1 on page 17. Symbols are used in the constant pool entry to represent the actual types. Table 3.3 contains a description the symbols used. For example, looking at row 2, the resolved value is given as *tickerupdate(Ljava/awt/Graphics;)V*. Using Table 3.3, this can be decoded as a method called “tickerupdate” that takes an instance of a graphic object as an argument, and returns void.

```
2a    aload_0                  //load local variable at table index 0
                               //which must be of type reference
2b    aload_1                  //load local variable at table index 1
                               //which must be of type reference
b6    invokevirtual xxxx       //call the method located by resolving
                               //constant pool index "xxxx"
b1                             //return
```

Figure 3.2 Code for the *paint*, *update*, and *paintAll* methods from the *ticker.class* files

Table 3.2 Example method indices and resolution values

Method Name	Indices/Resolved values
ticker:paint	0072, 003f, 005f, 0054, 0058, 003b, 0034, 004d, 0033, 006b, 003d, 006f
	Tickerupdate (Ljava/awt/Graphics;)V
Ticker:paint	0129, 0037, 0084, 0080
	Tickerupdate(Ljava/awt/Graphics;)V
ticker:update ticker:paintAll	00fb, 00fd, 00f2, 00c1, 009e, 0098, 004e, 0067, 00b1
	Tickerpaint(Ljava/awt/Graphics;)V
Ticker:update Ticker:paintAll	005f, 0175, 002a, 00ab, 01bc, 0043, 0198, 00bf
	Tickerpaint(Ljava/awt/Graphics;)V
Ticker2:paint	0072
	Ticker2update(Ljava/awt/Graphics;)V

Table 3.3 Java symbols used for identifying types

Symbol	Type	Description
B	byte	signed byte
C	char	character
D	double	double-precision ieee 754 float
F	float	single-precision ieee 754 float
I	int	integer
J	long	long integer
L<classname>;	...	class instance
S	short	signed short
Z	boolean	true or false
V	void	--
[	...	one array dimension

Looking at a particular instance of the method shown in Figure 3.2, and choosing the constant pool index of `0x005f`, the bytecode stream for this method would look like `0x2a2bb6005fb1`. Next, applying the constant pool resolution equivalence method to the method in Figure 3.2, and using the resolved constant pool values from Table 3.2, the constant pool index in the bytecode stream is replaced with the string created by concatenating the method name and method descriptor. These values are shown in the lower half of the second column in Table 3.2. When this substitution is made, two different resolved methods are created, `2a2bb6tickerupdate(Ljava/awt/Graphics;)Vb1`, given by the resolution in the first row of Table 3.2, and `2a2bb6Tickerpaint(Ljava/awt/Graphics;)Vb1`, given by the resolution in the fourth row of Table 3.2. This illustrates how the byte-for-byte equivalence technique for this method would have identified these two constant-pool-resolved equivalent methods as the same method, when they were in fact different. However, the generic form of the method in Figure 3.2, is the same for all the variations in Table 3.2. For this form of the method the `aload_0 (0x2a)` and `aload_1 (0x2b)` instructions are replaced with the `aload` (`0x19`) instruction. This gives a generic form of the method in Figure 3.2 of `0x1919b6b1`.

3.1.2 Discussion of techniques for exploiting method level 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. When methods exhibit resolved equivalence, they can share the same run-time memory space, and the code only needs to be loaded, linked and JIT-compiled once. Although generically equivalent methods are not as easily shared, techniques for exploiting this redundancy can be developed. For example, 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. 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 be used to represent the methods. An alternative VM implementation may cache the native code rather than invoke the JIT compiler each time. Because native code is typically much larger than the original bytecode, static code size becomes an even greater issue. Using techniques discussed in Chapter 4, however, only one copy of the native code needs to be stored for generically equivalent methods, with tables to indicate where the run-time linking needs to be performed. This can help reduce the cache size as well as load time.

Reusing generically equivalent methods, however, can limit the types of optimization performed on precompiled code segments. For example, certain aggressive optimizations that occur during compilation to native code, such as method inlining, may make use of the information in the constant pool. This problem, too, may be solved. For example the generic method can be tagged so that if a bytecode method appears to match the generic form, it will be further evaluated, and if an optimized form is available, it will be used instead of the more general generic form.

Another advantage to 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 parts of the Java programs for reuse at future invocations. 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.

3.2 Measuring Redundancy in the Collected Applet Sets

In this section, results of investigating redundancy on a method level are presented. This redundancy is explored from many levels, including class files that shared the same name but were physically different, the full sets of applets, the applet set variance over time, and finally individual applications.

3.2.1 Exploring redundancy in homogeneous class files

Starting with the already reduced set of applets in which all duplicate class files had been removed, further investigation was made of class files that had shared the same or very similar names but that were in fact physically different. Out of each of the three runs of the Web crawler, 10 such sets of files were chosen because the collected set contain a large number of variations for that particular applet name. Theorizing that within these file sets, referred to as the top 10 variable applets, a large amount of redundancy would exist under all three tests. Figures 3.3, 3.4, and 3.5 show the redundancy in each of these cases for the three runs of the Web crawler. As can be seen from all three graphs, less than 50% of the methods proved to be unique when the constant-pool-resolved equivalence method was used.

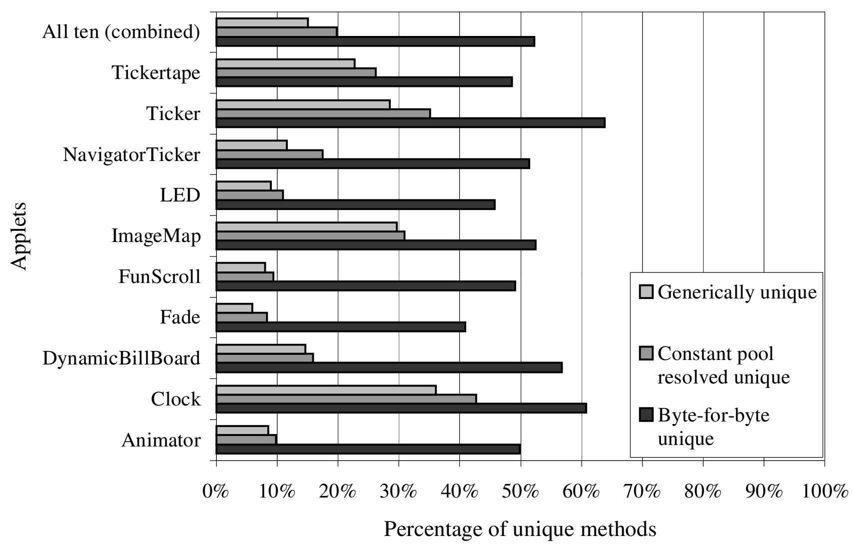

Figure 3.3 Top 10 variable applets from May 1997

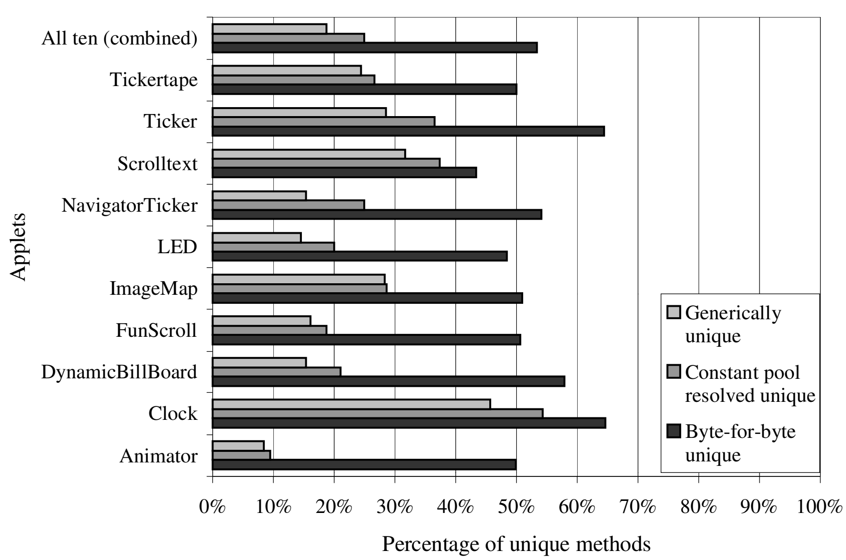

Figure 3.4 Top 10 variable applets from November 1997

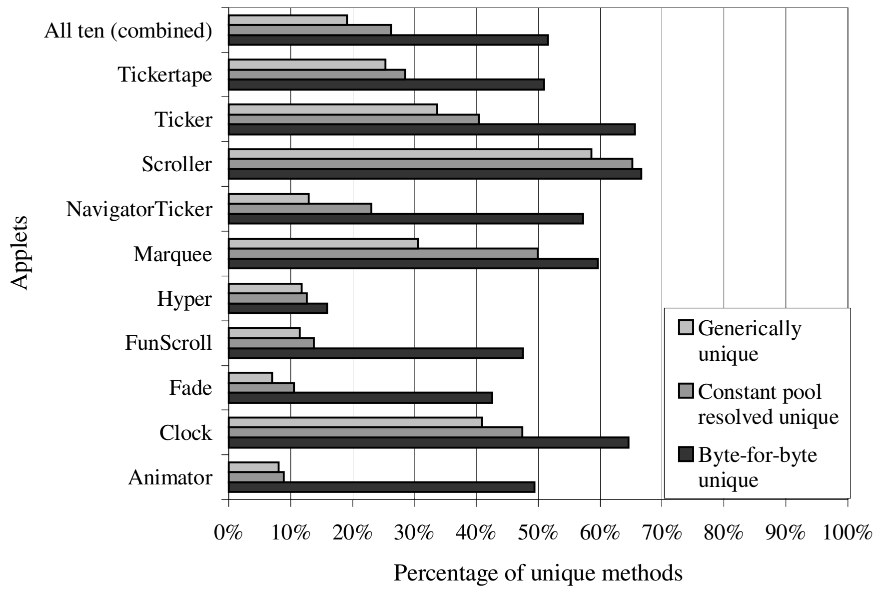

Figure 3.5 Top 10 variable applets from May 1998

One of the advantages to exploiting redundancy is in a JRE that reuses static native code versions of the bytecode methods. For example, looking at the Animator applet (the bottom sets of bars in the three figures), exploiting and identifying this overlap in a running JRE can reduce the amount of retained static native code. In this particular case, Animator is reduced to 10% of the methods identified necessary with only class file level redundancy exploited. Even in JREs that JIT the methods at run time, identifying and exploiting method level redundancy can significantly reduce the JIT time, reducing start-up costs and response time of a Java applet.

Although Figures 3.3, 3.4, and 3.5 show the potential of utilizing redundancy information, they focus on a particular moment in time. In a JRE system that retains static native code versions of the Java bytecode files, the native code is optimized to gain the best possible performance, a process that can be time consuming. Therefore any measure that reduces the need to recompile not only retains the improved code quality of the native code but also saves recompilation time. In the next model, the key issue investigated is changes across time. In the presence of changes to the bytecode files, a JRE utilizing static native code would invalidate the retained native code, falling back to a less optimized bytecode execution model. The ideal system would only invalidate code if none of the native code could still be used. This maximizes the retention of as much of the benefits from the static native code versions as possible.

Table 3.4 Unique applets of combined sets

	May 1997 and November 1997	November 1997 and May 1998
Total applets	1762	2571
Unique applets	1413	2245

To explore this issue further, the unique applets collected in the May 1997 Web-crawler run and the November 1997 Web-crawler run were combined. Also combined were the applets collected in the November 1997 Web-crawler run with those collected in the May 1998 run. Applying the class file level redundancy techniques discussed in Chapter 2, these two new sets were reduced to unique applet subsets. Table 3.4 shows the results of this reduction. Row 1 in this table shows the total count of the applets that were physically unique for the two sets added. In other words, this row lists the value obtained by adding the corresponding values from row 3, the physically unique row, in Table 2.1. The next row in Table 3.4, shows how many of these remained physically unique when the two set were combined. Note that if the applets that were unique in the May 1997 run had remained unchanged over the 6 months, the total of unique applets left after combining those applets with the applets unique in the November 1997 run should be close to the larger of the two values. This would be expected because all of the Web-crawler runs started with the same seed pages. However, this is not the case. The larger of the two values (the November 1997 values from Table 2.1, row 4) is only 976 unique applets. The number of applets in the combined set is significantly larger, showing a 30% increase in the class file variations. For the combined set of the November 1997 and May 1998 Web-crawler runs, this increase in class file variations is just under 29%.

For a JRE that reuses static native code, the change in the class files across time would invalidate at least 30% of the static native code. However, looking at the finer grain method level redundancy under the three main tests, this invalidation was unnecessary. To investigate the amount of redundancy left unexposed by just the class file level checks, the methods that shared the same name but were not physically identical were examined closer. Figures 3.6 and 3.7 show the results of this comparison. As implied from the results of the class file level redundancy reduction shown in Table 3.4, there is a large variance between the files when comparing them using a byte-for-byte equivalence tests. The increase in the percentage of unique methods under this test, when compared to the same test conducted on a single run, implies that the files have changed over time. This change could unnecessarily trigger an invalidation of the static native code. As can be seen from looking at the applets in the combined sets, less than 50% of the methods remain unique when under the constant pool resolved equivalence mapping, whereas over 90% of the methods were unique if compared using byte-for-byte equivalence.

The result is even more dramatic for the Animator applet discussed in the individual runs. The results for this applet are shown as the bottom sets of bars in Figures 3.6 and 3.7. Note that for this applet, only 20% of the methods actually changed.

However, the byte-for-byte equivalence tests indicates that over 95% of the methods appeared different.

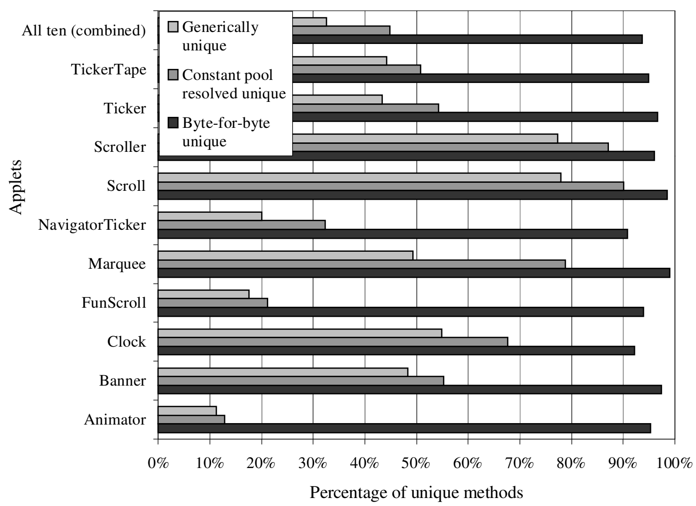

Figure 3.6 Changes over time May 1997 to November 1997

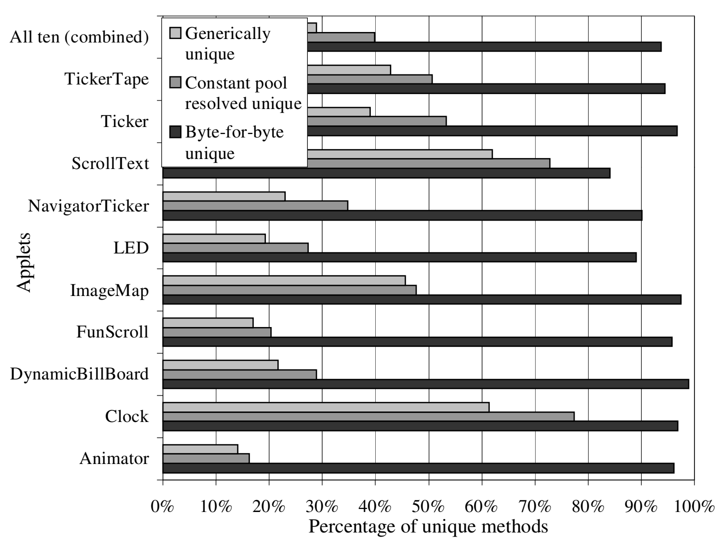

Figure 3.7 Changes over time November 1997 to May 1998

3.2.2 Exploring method redundancy in heterogeneous class files

Although the benefits to a JRE system resulting from exploiting method-level redundancy in a pseudo-homogeneous set of class files have been shown, the question still remains as to whether or not there are benefits to exploiting redundancy across a more heterogeneous set. The investigation of this question focused on the method-level redundancy under the three defined redundancy mappings for each of the collected applet set. Figure 3.8 shows the results of this investigation. Although not as dramatic as what had been seen in the homogeneous collections, the results show a definite overlap. In all three sets, less than 70% of the total methods remained unique, even under the byte-for-byte equivalence testing. This percentage of unique methods decreases even further under the other two mappings.

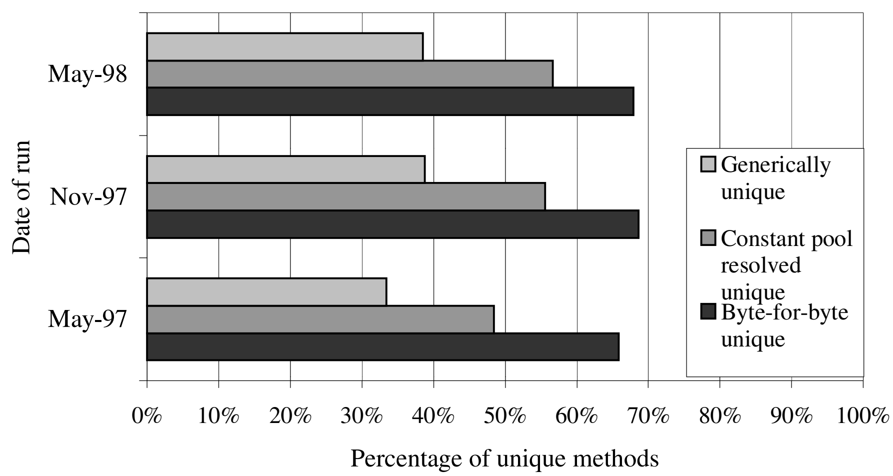

Figure 3.8 Method redundancy in all applets combined

Even though Figure 3.8 gives a more heterogeneous view of the redundancy effects, it is still only a pseudoview of a heterogeneous result, because this set of applets still contains the homogenous file variation described previously. To remove this factor, a collection was made of applets that characterized the set collected in the Web-crawler runs but that did not include the pseudo-homogeneous files seen in these runs. The investigation was also expanded to examine whether there was redundancy within a given program that could also be exploited. To facilitate this investigation, the SPECjvm98 suite of benchmarks was used [22]. The SPECjvm98 benchmark suite consists of a wide range of Java applications chosen because they most closely characterize the current Java workload set. A brief description of each of these benchmarks is given in Table 3.5.

Table 3.5 Description of the applications in the SPECjvm98 benchmark suite

`_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 level.
`_209_db`	A database application performing multiple operations on a memory resident database.
`_213_javac`	Java source to bytecode compiler -- scaled down from the 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.

Figure 3.9, shows the results of the defined tests for method level redundancy on these two new benchmark sets. As anticipated, with these heterogeneous sets, the results were not as dramatic as those seen in the more homogeneous collections already analyzed. Although this result contrasts with the behavior of applet sets collected from the Web-crawler runs investigated earlier, it should be expected given the nature of the benchmark programs. Additionally, many of the benchmarks, such as `_209_db`, invoke such a small number of methods that an overlap between them is unlikely. Each benchmark is intentionally distinct from the others and executes a very specific function, whereas the previous applets often performed many similar operations. Although individually the SPEC benchmarks exhibited significantly less redundancy than the downloaded applets, the full suite, as a set exhibited a slightly higher redundancy, with

the generically equivalent mapping. Still not as high as what was observed earlier, 45% of all the methods in this benchmark suite were redundant. Also interesting in the SPECjvm98 benchmark suite were benchmarks that clearly exemplified the potential problems with using only a byte-for-byte equivalence mapping. Looking at Figure 3.9 and the individual graphs for `_202_jess`, `_227_mtrt`, `_213_javac`, and even the All Spec graph, it is clear that doing a byte-for-byte equivalence incorrectly identified methods as being the same, when in fact they were not. This present a danger in a system that uses optimized native code. If a byte-for-byte equivalence test were used to detect changes in the class files, then the system would not detect the variance in the target and would incorrectly use the byte-for-byte equivalent code. As seen bar in Figure 3.9 from the middle of each benchmark, when the references were resolved with the constant-pool-resolved equivalence mapping, the percentage of unique methods actually increases. However, on systems that use a generic native version, this problem disappears because none of the indices are included, and all resolution is done at run time.

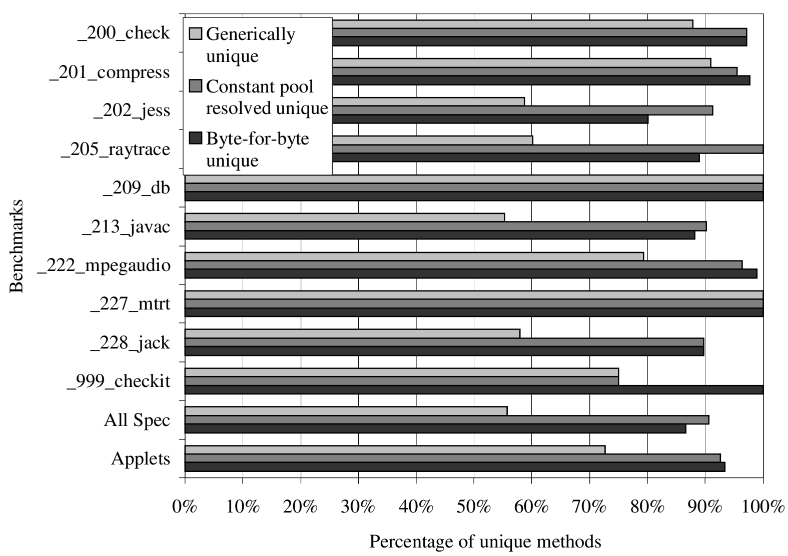

Figure 3.9 Method level redundancy in SPECjvm98 and benchmark applets

Various opportunities to exploit redundancy have been shown, ranging from the more significant redundancy exposed with the Web-crawler applet set to the smaller redundancy available to exploit in a single application or heterogeneous set of applications.

4. INTERFACE DESCRIPTION

Although the techniques discussed have proved successful in detecting and exposing redundancy within a class file, the feasibility of using a form of equivalence mapping to locate and use static native code in a JRE still needs to be shown. An equivalence form of the bytecode method will be used to serve several purposes in the native code system. First, as shown in Chapter 3, the amount and accuracy of the redundancy that is exposed is dependent on the equivalence technique used; therefore, the technique chosen will have a direct impact on the size of the native code tables. Second, the equivalence form of the method will be used as an index tag to locate the native code within the run-time tables; therefore, speed both in creating the equivalence form and speed in locating the proper table entry using the created form becomes an issue. The generic form of a bytecode method did not contain enough information to uniquely identify the correct native template. Although the constant pool resolved equivalence method correctly identified methods that were indeed the same, it was the most time intensive of the three mappings. Also, since the constant pool values need to be resolved at run time in order to use this form of mapping; a more generic version will gain a larger reuse and not add significant overhead to the run time. The use of a more generic form and resolving values at run time also allows for a wider reuse of static native code. For example, looking back at Table 3.2, in Section 3.1.1, the differences in these variations of the generic form of the method are simply in the method name and the name of the method being invoked. The equivalence mapping should expose this overleap while maintaining the integrity of the mapping.

Once the characteristics wanted in the equivalence mapping technique had been identified, acquisition of the proper Java tools to investigate the feasibility of using this equivalence mapping in the design and implementation of a JRE reusing static native code, was next. To study the use of static native code in a JRE and the proper equivalence mapping to obtain the correct native code, a state-of-the-art JRE, Microsoft's JVM 2.02 [23], was chosen. Using documentation provided by Microsoft, a JIT interface layer that impersonated the Microsoft JIT was constructed. The new layer was used as a simple slim bypass layer that would check for native code templates, and if available, substitute the native code version for the JIT-compiled version. In the absence of a native code template, the interface simply called through to the Microsoft JIT.

To arrive at a one-to-one mapping between an equivalence form of the bytecode method and a template form of the native code, the native code generated by the Microsoft JIT was captured. For the input set of benchmarks, the SPECjvm98 benchmark suite and the characteristic set of applets were used. The important thing to note in these two sets, is that any pseudo homogeneous redundancy on a class file level has already been removed. Therefore, a redundancy overlap similar to that exposed in Figure 3.9 was anticipated.

4.1 Checking Redundancy at JIT Interface

Before testing the use of static native code in place of JIT calls, measurements of the actual method level redundancy at this interface were needed. Analyzing the static code allowed 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 the static approach would include even those methods never used. With the static analysis, the Java input set was restricted to the code that is distributed with the applet or application because this is the additional code that would occupy the additional disk/storage space. However, in doing so, redundancy in methods from the core Java libraries that may be called by the program were not investigated. Because this investigation focuses on the possibility of reusing previously loaded or compiled methods, only those methods for which the VM requests a JIT-compiled version were investigated. To monitor JIT requests made during execution of the benchmarks, the JIT interface layer described earlier was used. This interface actually replaces the DLL that services JIT requests in Microsoft's VM 2.02. The statistics on the requested methods were recorded and control passed to the real JIT.

Table 4.1 Unique method JIT requests

Benchmark	Total Number of JIT Requests	Byte-for-Byte Unique	Constant Pool Resolved Unique	Generically 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 4.1 shows the redundancy seen at this interface for each of the three mapping techniques. 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 by comparing the value in the row labeled "Sum of Above" with that of the full SPECjvm98 run. Approximately 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 with each specific run, the full run called many of them only once. This is part of the 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 machine invocation.

Column 2 of Table 4.1 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 3 shows the number of unique methods after checking for constant pool resolved equivalence. These methods can be used interchangeably at run time, 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. Column 4 lists the number of unique methods found by using the test for generic equivalence. A comparison between columns 3 and 4 reveals that the technique of masking out constant pool indices exposes a much larger set of redundant methods. One interesting item is that the amount of redundancy is similar for both the applet and the 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.

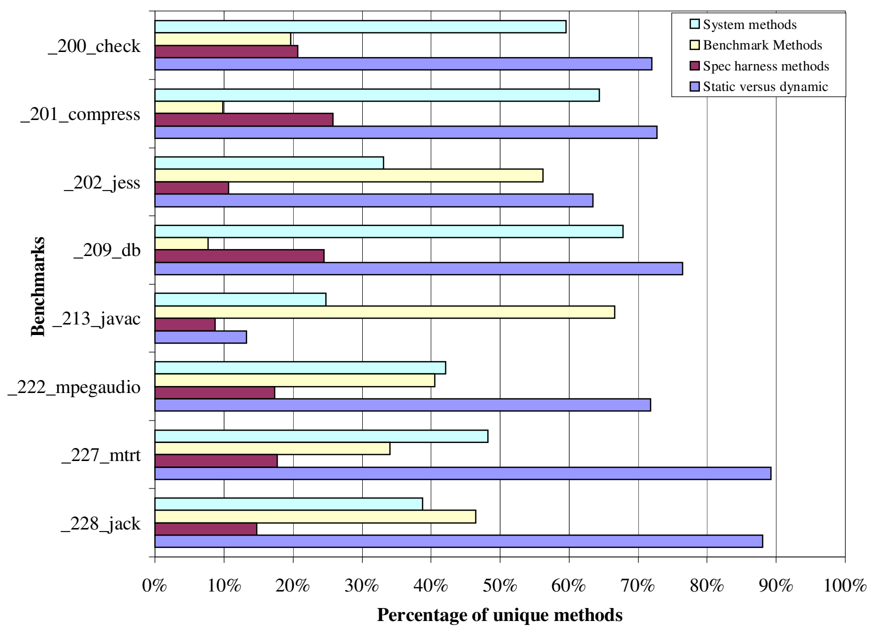

Figure 4.1 Relative percentages of dynamic methods

Figure 4.1 gives a breakdown of the methods called in each SPECjvm98 benchmark. The bottom bar for each of the benchmarks in Figure 4.1 gives the percentage of static methods measured and reported in Section 0 that were actually JIT-compiled in the dynamic run. As explained at the start of this section, not all methods investigated during static analysis are actually used during a full run. As seen in Figure

4.1, 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 for use across benchmarks and to save on memory usage (as with the *object* initializer earlier) the JITed set of methods from dynamic runs of the program should be used. However, when looking at a reduction in static code size by storing a generic form and using a mapping (like that suggested in Section 1.2.2), the static version of the files may still be used.

The next bar in Figure 4.1 (third from the top of each set) is the percentage of total JIT-compiled methods that result from the overhead of the SPECjvm98 harness. The next bar up is the percentage of JIT-compiled methods that are benchmark specific. This ranges from 8% in `_209_db` to 67% in `_213_javac`. Finally, the top bar in Figure 4.1 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.

4.2 Template Description and Template Insertion

To reuse static native code and gain the most benefit from a run-time system, the chosen form of native code representation must allow for run-time linking and resolution information to be incorporated in the structure. This native code version of a bytecode method is referred to as a *native template*. The mapping from the original bytecode stream to the generic version used to locate the native code template should fall somewhere between the constant-pool-resolved mapping and the generic mapping. Also, the main purpose of the system is to explore the feasibility of reusing statically maintained native code at run time. Therefore, care is taken to maintain the boundary between (a) run time linking and customizing by mapping *arguments* into the native template at run time, and (b) JIT-compilation by changing and adding native *instructions* at run time.

To decide the correct mapping, which is also used as an accessing technique for locating the native templates at run time, the generic equivalence form was used as a starting point. Starting with this mapping of the bytecode as a tag to place the collected native code versions of the methods into a table, the generic equivalence form was refined slowly until arriving at a mapping that allowed the location of a unique template version of the native code.

The first item noticed during the refining process was that, in `x86` code, the return statement is dependent on what is being returned. This led to the first modification of including the return type. Next, the type of arguments used by a function also directly affected the instruction mix of the native code. For example, if the arguments are passed by reference versus primitives that are passed by value, the instructions for loading and manipulating these arguments change. Therefore the second modification was the inclusion of the argument types passed to the method. Finally, noticing that the limited number of registers available within the `x86` architecture set affects the instruction mix of the native template due to spill code and register usage patterns, the live range for the method's local variables was included. To maintain the variable position abstraction within the variable table, a normalized mapping of indices is used. Figure 4.2 shows the algorithm used to produce the new generic form of the bytecode stream. This algorithm only includes the mapping for the actual bytecode instructions. The appending of the additional information is not shown.

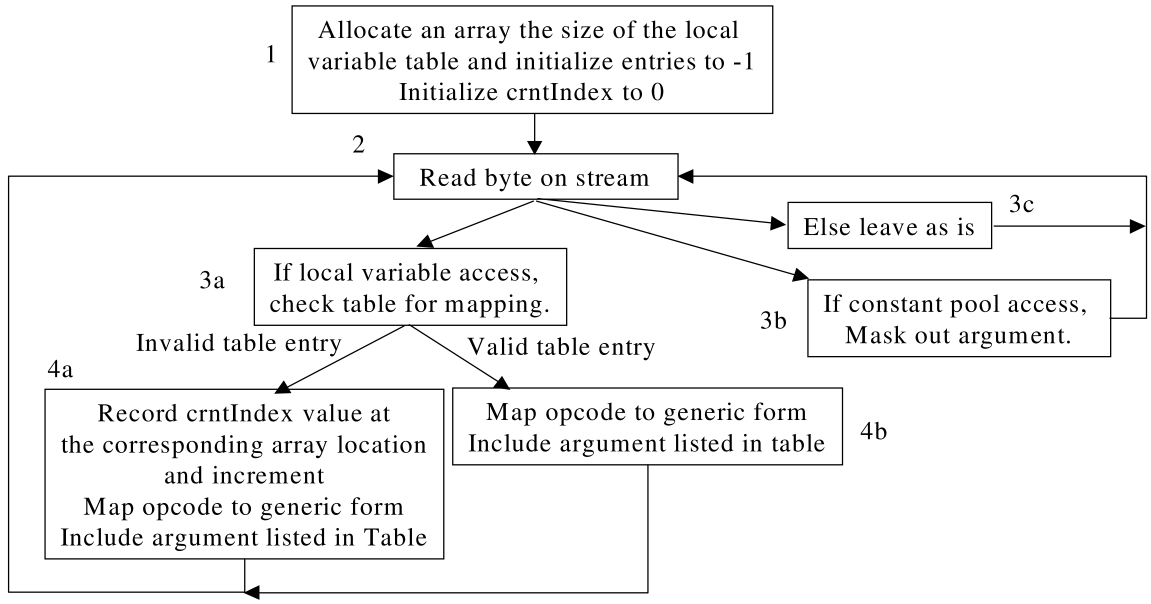

Figure 4.2 Algorithm used to form generic tag field

Figure 4.3 gives the output of walking through the algorithm in Figure 4.2. The third column in Figure 4.3 shows the table that holds the new index values. It is initialized to all negative ones to denote that no index value has been assigned yet. This corresponds to step 1 in Figure 4.2. The entries in the table in column 3 of Figure 4.3 are replaced as needed following the rules listed in step 4a of Figure 4.2. Following the steps listed in the rows of Figure 4.3, a refined generic form of the bytecode stream is produced.

After the bytecode stream given in Figure 4.3 has been formed, the local variable table size, followed by the types of the local variables and the return type, is appended to the stream. This final generic form of the example method is shown as the indexing tag for the final entry in the sample template entries in Figure 4.4. Table 3.3 (page 21) shows

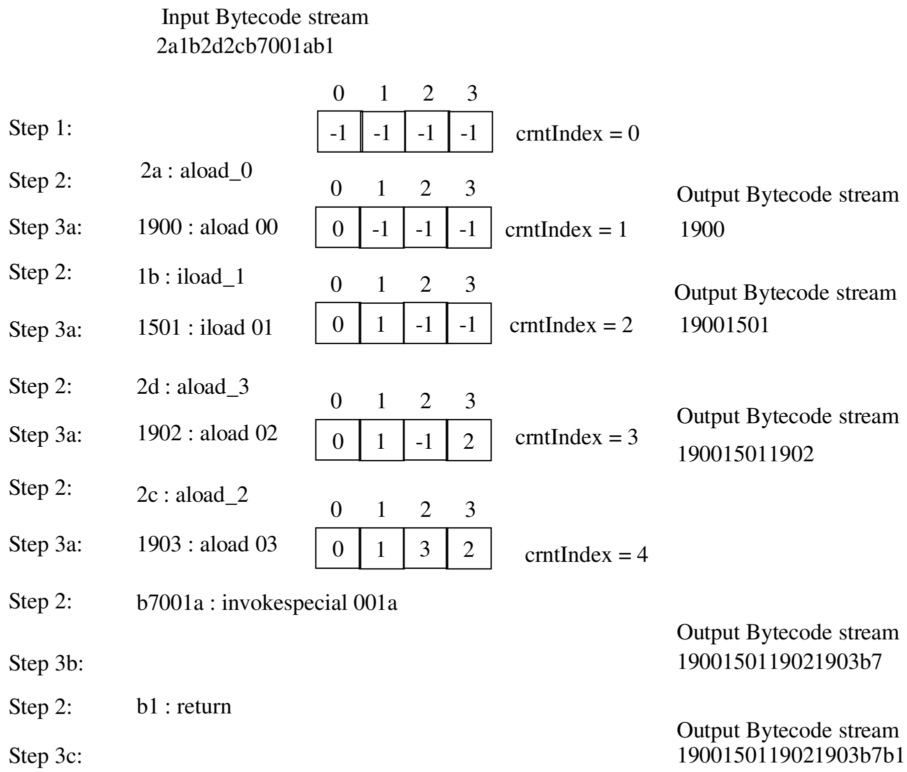

Figure 4.3 A walk through of the algorithm shown in Figure 4.2

```
'1900b719001901b7b1 2)V'
   (Index(0 1)
    Native('568b74240c56ff15'
           '56ff74240cff15'
           '5ec20800')
    count(0));
 '19001901b8b7b1 2)V'
   (Index(0 1)
    Native('ff742408ff742408ff15'
           '50ff15'
           'c20800')
    count(0));
'1900150119021903b7b1 4LL)V'
   (Index(0)
    Native('ff742410ff742410ff742410ff742410ff15'
           'c21000')
    count(0));
```

Figure 4.4 Samples of table entries in the native template code tables the majority of the mapping for the symbols used to represent the local variable and return types. The symbol ")" (not listed in Table 3.3), is used to represent the "this" pointer, a reference pointer to the class instance where the method resided and implicitly passed to all nonvirtual methods.

```
Refined generic form: '03ac 4II)Z'
Bytecode:
03          iconst_0
ac          ireturn
Native code:
33 C0               xor    EAX, EAX, EAX
C2 10 00            ret    0010h
========================================
Refined generic form: '03ac 1Z'
Bytecode:
03          iconst_0
ac          ireturn
Native  code:
33 C0               xor    EAX, EAX, EAX
C2 04 00            ret    0004h
========================================
Refined generic form: '03ac 3L)Z'
Bytecode:
03          iconst_0
ac          ireturn
Native  code:
33 C0               xor    EAX, EAX, EAX
C2 0C 00            ret    000Ch
========================================
```

Figure 4.5 Samples of native code variations

Referring back to Table 3.2 in Section 3.1.1, this refined generic form maps all the rows to the same native template. This is because all 61 occurrences of this generic form in the 69 *ticker.class* files have the same return type, argument type, and local variable table size. If any one of these values had been different, then the refined generic form would have mapped the method containing the variance to the correct native template. For example, Figure 4.5 shows three native code versions of a simple bytecode method. All three versions return a Boolean type. The variations in the three methods come from the argument types and the local variable table size. In fact, in this example, it is the local variable table size that directly affects the value used by the return statement in these three versions. Note that the native code for the first example in Figure 4.5 returns the item offset by four word locations, whereas the second example offsets only one, and the final example offsets three. However, with a more advanced mapping technique, this variation in the native code could be handled. Samples of the native code template used in the interface are given in the native field of the sample table entries shown in Figure 4.4. Note that this template actually breaks the native form into disjoint pieces. The pieces are fit together at run time by resolving the values that belong to the calls at each piece boundary.

Figure 4.6 illustrates a basic flow diagram of the algorithm used for the JIT interface that maps these native versions at run time. Box 1 in Figure 4.6 represents a request made by the Java run time for a JITed version of a bytecode method. In box 2, a call is made to the function that forms the refined generic form of the bytecode method. This function performs the algorithm shown in Figure 4.2 with the additional function of resolving constant pool references into their corresponding run-time values. The resolution of constant pool values is facilitated by the use of helper function provided with the Microsoft Java virtual machine. The run-time values are placed in a table at the entry index that corresponds to when the instruction appeared in the bytecode stream.

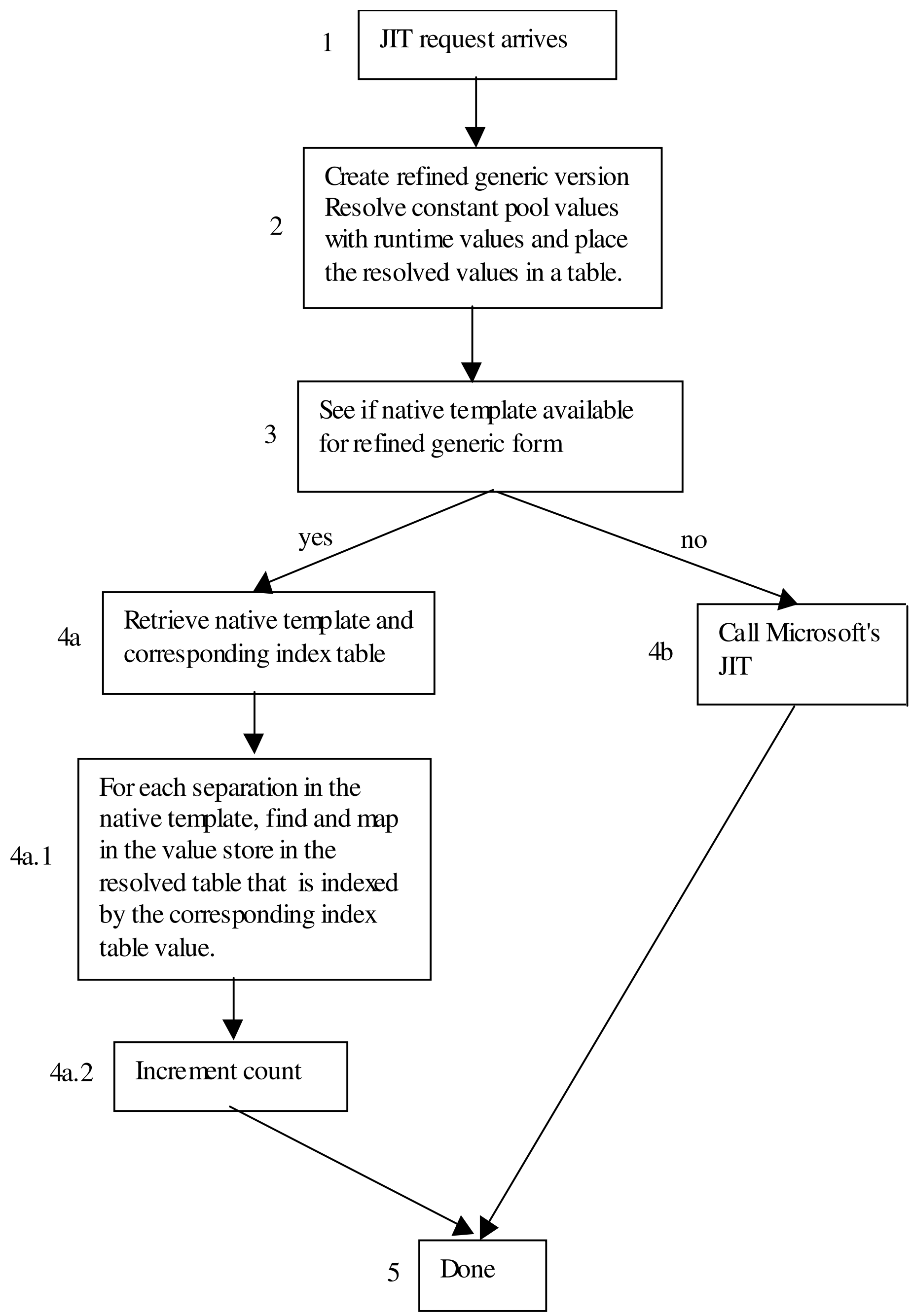

Figure 4.6 Basic flow chart of JIT interface

After the refined generic form has been created and the table of the resolved constant pool run-time values has been made, the JIT interface procedure checks to see if a native template exists for this generic form. This corresponds to box 3 in Figure 4.6. If none exists, then the Microsoft JIT is called and the interface is done, box 4.b in Figure 4.6. However, if a native template exists, then the template is mapped into the corresponding memory location provided by the virtual machine to hold the native code version of the bytecode method. The first step in this mapping is to retrieve the native template pieces and the index table that are part of the JIT interface's run-time table. These fields are shown in Figure 4.5. The mapping of the values in the resolved table created by the function represented by box 2 in Figure 4.6, and the positions between the segments of the native code template, is done by using the index fields. These index fields are shown next to the index tag in Figure 4.4. They correspond to actual indices into the resolved table. For example, if the bytecode stream contained an invoke followed by a `getfield` and then a `putfield`, the resolved values would be placed at 0, 1, and 2 in the resolved table formed by the function represented by box 2 in Figure 4.6. The index values represented by the index field in Figure 4.5 would be 0, 1, and 2, respectively.

By segmenting the native template and an index table to map between the corresponding native template locations of the instructions, the native code version can be restructured and optimized. If the native template for this example bytecode method were to perform the `getfield` first and then the invocation followed by the `putfield`, the index values in the JIT interface table entry are all that need to be adjusted in order for this revised version to work. The new index field would be 1, 0, and 2, respectively. This technique also allows the native code template to optimize out any values and to disregard the corresponding resolved values.

Using hand techniques on collected data to determine the proper form and segmentation of the native code templates, 64 native code templates were created. The results for using these native code templates in place of JIT request for the SPECjvm98 benchmark suite and the characteristic applet set, are shown in Table 4.2. The middle column of Table 4.2 gives the percentage of total JIT calls that were replaced by a native code template. The final column of Table 4.2 shows the percentage of the 64 native code templates that were actually used. Note that for the SPECjvm98 runs, most of the benchmarks individually use less than 40% of the available native code templates, whereas a full run of the entire suite uses over 50%. These numbers increase if SPECjvm98 is run through the GUI harness versus the command option used for these measurements. In fact, running the SPECjvm98 benchmark suite through the Java GUI harness that accompanies it increases the percentage used of the available native code templates to 85.5%.

Table 4.2 Native code template results

Benchmarks	Percent of Templates versus JIT calls	Unique Templates
`_200_check`	18.4%	34.4%
`_201_compress`	16.9%	34.4%
`_202_jess`	15.9%	39.1%
`_209_db`	15.6%	34.4%
`_213_javac`	6.5%	45.3%
`_222_mpegaudio`	13.3%	35.9%
`_227_mtrt`	11.6%	34.4%
`_228_jack`	14.9%	34.4%
All SPEC	8.4%	51.6%
Applets	7.5%	93.8%

Although the handcrafted set of native code templates covered between 6.5% and 18.4% of the total JIT requests seen in the benchmark sets, the goal of these experiments was to test feasibility of the system, not its range. With expanded tools, the percentage of JIT requests handled in this fashion is expected to increase.

5. SUMMARY

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. This study was directed at locating general areas of performance improvement within the Java programming paradigm. We have demonstrated techniques for exposing redundancy and given an experimental system that uses the techniques to reuse static native code.

REFERENCES

[1] S. Singhal, and B. Nguyen, "The Java factor," *Communications of the ACM*, vol. 41, no. 6, pp. 34-37, June 1998.

[2] D. Bank, "The Java saga," *Wired,* vol. 3, no. 12, pp. 166-169, December 1995.

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

[4] D. Gardner, "Sun plans to make Java ubiquitous," *InfoWorld*, vol. 20, no. 44, November 2, 1998*, http://www.infoworld.com/cgi-bin/displayArchive.pl?/98/44/t08-44.8.htm.*

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

[6] X. Zhang, "Secure code distribution," *IEEE Computer*, vol. 30, no. 6, pp. 76-79, June 1997.

[7] "Secure computing with Java: now and the future," White Paper, Sun Microsystems, Mountain View, CA, April 1997.

[8] B. Morrey and J. Broderick, "Java in the enterprise," *InfoWorld,* vol. 19, no. 37, pp. 76-80, September 15, 1997.

[9] T. S. Bowen, "IBM pushes on-demand server management scheme," *InfoWorld,* vol. 20, no. 41, p. 12, October 12, 1998.

[10] D. Gardner, "Server-side Java gains more momentum," *InfoWorld*, vol. 20, no. 41, p. 24, October 12, 1998.

[11] J. G. Angus, "Browser acceleration utility Peak Net.Jet speeds up Web browsing for some users," *InfoWorld*, vol. 19, no. 5, p. 2/JW, February 3, 1997.

[12] V. R. Garza, "CacheFlow stems Web-traffic tide at hefty price," *InfoWorld,* vol. 20, no. 18, p. 62D, May 4, 1998.

[13] P. Korzeniowski, "Boosting bandwidth," *InfoWorld,* vol. 19, no. 18, May 5, 1997, *http://www.infoworld.com/cgi-bin/displayArchive.pl?/97/18/e01-18.71.htm.*

[14] E. R. Harold, *Java Developer's Resource.* Upper Saddle River, NJ: Prentice Hall, 1997.

[15] D. Barry and T. Stanienda, "Solving the Java object storage problem," *IEEE Computer,* vol. 31, no. 11, pp. 33-40, November 1998.

[16] S. Bailey and R. L. Grossman, "Jtoll: Accessing warehoused collections of objects with Java," in *Proceedings of the Second International Workshop on Persistence and Java,* December 1997, pp. 151-167.

[17] F. Yellin, "The JIT Compiler API," White paper, Sun Microsystems, Mountain View, CA, October 4, 1996.

[18] C.-H. Hsieh, M. Conte, T. Johnson, J. Gyllenhaal, and W. Hwu, "Optimizing NET compilers for improved Java performance," *IEEE Computer*, vol. 30, no. 6, pp. 67-75, June 1997.

[19] R. Rivest, "The MD5 message-digest algorithm," MIT Laboratory for Computer Science, and RSA Data Security, Inc. Request for Comments: 1321, April 1992.

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

[21] T. Lindholm and F. Yellin, *The Java Virtual Machine Specifications.* Reading, MA: Addison Wesley, 1997.

[22] *SPEC JVM Client98 Suite*, Standard Performance Evaluation Corporation, Stanford, CA, Release 1.0 August 1998, *http://www.spec.org/osg/jvm98/*.

[23] *Microsoft Java Virtual Machine version 2.02,* Microsoft Corp., Redmond, WA. June 23, 1998, *http://www.microsoft.com/java/vm/dll_vmsp2.htm*.