Performance Pitfalls in Large-Scale Java Applications Translated from COBOL

Toshio Suganuma[*] Toshiaki Yasue Tamiya Onodera Toshio Nakatani
IBM Research, Tokyo Research Laboratory
1623-14 Shimo-tsuruma, Yamato
Kanagawa, Japan
{suganuma, yasue, tonodera, nakatani}@jp.ibm.com

Abstract

There is a growing need to translate large-scale legacy mainframe applications from COBOL to Java. This is to transform the applications into modern Web-based services, without sacrificing the original programming investments**.** Most often, COBOL-to-Java translators are used first for the base program transformations, and then corrections and fine tuning are applied by hand to the resulting code. However, there are many serious performance problems that frequently appear in those Java programs translated from COBOL, and it is particularly difficult to identify problems hidden deeply in large-scale middleware applications.

This paper describes the details of some performance pitfalls that easily slip into large-scale Java applications translated from COBOL using a translator, and that are primarily due to the impedance mismatch between the two languages. We classified those problems into four categories: eager object allocations, exceptions in normal control flows, reflections in common paths, and inappropriate use of the Java class library. Using large-scale production middleware, we present detailed evaluation results, showing how much overhead these problems can cause, both independently and collectively, in real-world scenarios. The work should be a step forward toward understanding the problems and building tools to generate Java programs that have comparable performance with corresponding COBOL programs.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – Classes and objects.

General Terms Performance, Languages.

Keywords Legacy migration, COBOL to Java translations, Object allocation.

1. Introduction

There is a growing need to translate large-scale legacy mainframe applications from COBOL to Java for modernization. This legacy transformation is happening in many industries, such as financial and manufacturing, primarily for two reasons: (1) applications need to be adapted for a modern Web-based service-oriented architecture (SOA) [4], and (2) it is becoming increasingly difficult to find skilled COBOL programmers to perform program maintenance. These applications are often decades old, monolithic, huge, and complex, produced from continuous program modifications and enhancements extending over long periods of time. Since it is both difficult and impractical to manually port such large-scale legacy COBOL programs to Java, in many cases COBOL-to-Java translators are used first and then corrections and fine adjustments are added by hand to the resulting code.

Among the post-translation refinements, performance tuning is one of the most difficult and painful tasks, especially for large scale applications. It requires collecting and analyzing execution profiles for critical application scenarios, identifying performance bottlenecks, making necessary modifications in the translated programs, and verifying the results. Fixing one major problem often reveals other problems hidden behind it, and in many cases the process needs to be repeated a number of times to finally meet the required levels of performance for an application.

There are many COBOL-to-Java translators available in the market [3][5][9][10], and there are variations in the quality of the translated code. However, even the best translators produce code that tends not to perform well in Java. Some of these performance problems are due to *impedance mismatch* between COBOL and Java programs, while others are due to inappropriate conversions into Java.

Impedance mismatch refers to significant differences of the basic program structures between the two languages. For example, COBOL declares an elementary data item (called

[*] Currently in IBM Global Business Service Division, Tokyo Japan

a local variable in many other languages) with a format specification using a PICTURE clause (e.g. PIC A(4) denotes four alphabetical characters, and PIC S9(3) denotes a signed 3-digit number). Therefore, depending on the format specified, the number of bytes to be allocated in memory is initially defined for the local variable. When translated to Java, such local variables (typically represented as String or BigDecimal) are allocated as object instances with a specified number of white space characters or zeros as initial values. These objects are eagerly allocated at declaration time and then thrown away as the execution proceeds and the variables are assigned to new values, since the objects are immutable.

This translation of local variables is natural for translators, because it would require elaborate and complex control flow analysis to avoid wasteful eager object creation by finding the first definitions and usage points of the variables in the program. However, if all the local variables, not just the variables of Strings and BigDecimals but also those of collection types (e.g. arrays), are allocated in this manner, the cumulative cost can be huge, leading to memory bloat. In one example, we encountered a class definition that included instance variables of deeply nested arrays with BigDecimal and String objects at the bottom, and a single instantiation of this class consumed more than 20 Mbytes of memory. When we examined this class, we discovered that most of these objects were not touched and the needs could be satisfied by allocating objects of a few hundreds Kbytes.

A second reason for poor performance is inappropriate program conversions, especially into uses of exceptions, reflections, and standard Java class library. For example, translators tend to use the standard class library as much as possible to implement the functions required in the original program logic. This simplifies the translation process, but many expensive operations, such as object allocations, synchronizations, and internal data conversions, are hidden in the class library. The translator is not aware of the performance penalties that must be paid at execution time when using those library calls. In some cases, the translated code has to create additional object allocations and data conversions just to satisfy the required interface to invoke the library, and in some cases the data reverts back to the original form in the library code. This is pure overhead irrelevant to the application logic. The problem of inappropriate library calls and the related overheads for crossing boundaries is not translation-specific, but may appear generally in framework-based applications [8].

This paper describes performance pitfalls that are typically found in large-scale Java applications translated from COBOL, and quantifies the performance overheads those problems can cause for real world scenarios in large-scale transaction applications. This work involved performance problems found in large-scale production middleware that was originally written in COBOL and recently converted to Java using a well-known translator. Based on these experiences, we found the problems can generally be classified into four categories: eager object allocations, exceptions used for controlling normal execution flows, reflections in common paths, and inappropriate uses of the class library. All of these problems combine for significant performance penalties.

The following are the contributions of this paper:

- **Identify performance bottlenecks and pitfalls that easily slip into large scale Java applications translated from COBOL:** We present a number of performance pitfalls typically found in large-scale Java programs translated from COBOL, and categorize them into four types of problems: eager object allocations, exceptions used in normal control flows, reflections in common paths, and inappropriate uses of the Java class library.
- **Detailed experimental evaluations showing significant performance overheads caused by those problems:** We present detailed evaluation results using production middleware that was actually translated from COBOL. We isolated each of the four types of problems, and quantified the overheads that can be reduced by addressing these problems.

The rest of this paper is organized as follows. The next section gives an overview of COBOL program specifications using some simple examples, and shows how Java programs translated from COBOL can perform poorly. Section 3 describes the four types of problems we found in most Java programs translated from COBOL. Section 4 presents experimental results using large-scale middleware that was actually translated from COBOL and shows the performance impacts of these problems in several transaction scenarios. Section 5 summarizes related work, and finally Section 6 presents our conclusions.

2. COBOL Programming Examples

The first COBOL language specification was for COBOL-60, and the language has evolved since then, with additions and improvements for many features of the language. This section is a brief explanation of COBOL programming [1] using simple examples, focusing primarily on features that can cause performance problems when translated into Java.

A COBOL program's basic structure consists of 4 divisions (IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE). All data and variables to be used in the program need to be declared in the DATA DIVISION, and the program body is written in the PROCEDURE DIVISION. All the data and variables are first allocated in memory when the program is loaded, and they stay in the same address space until the program terminates. There are none of the dynamic allocation features that modern languages like Java, C, and Pascal offer. COBOL uses process-based execution and has only a single thread of control, so no memory corruption problem can occur with this storage management approach. Therefore, the biggest challenge for COBOL-to-Java translation is how to efficiently manage the data and variables in the Java heap. There are several language features of COBOL for data declarations and manipulations. These are quite different from Java's and thus can be sources of performance problems when translated.

Elementary data item and PICTURE clause: Variables in Java correspond to elementary data items in COBOL, and are declared with a PICTURE clause. COBOL is not a typed language, but rather it allows programmers to provide the system with an example (a picture) of how the storage of each data item should be displayed, and hence the amount of storage to be reserved. The following example shows several data items declared in COBOL.

```
01  Name          PIC A(20)        VALUE  SPACES.
01  AccountInfo   PIC X(50)        VALUE  SPACES.
01  Balance       PIC 9(10)V99     VALUE  ZEROS.
```

Each data item is declared with a level number, followed by a name for the data, followed by a picture clause, optionally with an initial value in a VALUE clause. The characters in the PIC clause indicates how many characters or digits occur in the storage (an A denotes an alphabetic character, an X denotes any character in the character set, a 9 denotes a digit, and a V denotes the position of the decimal point in a numeric value,). Recurring symbols are specified with a repeat factor in parentheses. SPACES or ZEROS are figurative constants that act like one or more spaces or zeros.

In general, numeric values with a decimal point specified (with the V in the PIC clause) are translated to BigDecimal, other numeric values to int or long, and all of the other elementary data items to String in Java. Since fixed point is the only kind of decimal numeric value in COBOL, the Java translator has to choose either to use floating point numbers (and accept some precision errors), or to use the java.math.BigDecimal class to preserve the original precision. Since many of COBOL programs are used for business applications where no precision differences are allowed, the translators generally have to be conservative and use BigDecimal in Java.

Group data item: A group represents a collection of elementary items with a nested structure. This is merely a structured way of manipulating multiple data items, and therefore no corresponding storage is reserved (the size of the group item is just the sum of the sizes of its subordinate elementary items). Tables can also be defined as a group item using the OCCURS clause. In the following example, a table of StudentRecord with a size of 40 elements is defined, each of which includes three elementary items. Group items are usually translated using classes, inner

```
01   StudentRecord    OCCURS  40  INDEXED  BY  K.
  02  Name            PIC X(10)       VALUE    SPACES.
  02  MathScore       PIC 999         VALUE    ZEROS.
  02  ScienceScore    PIC 999         VALUE    ZEROS.
```

classes, and arrays in Java.

REDEFINES clause: The storage allocated for data items can be overlaid using a REDEFINES clause. This allows users to define a data item in one format, and use the same data with another format. In the example below, MonthDef is first defined as a sequence of 36 alphabetic characters, and then the exact storage location is redefined with a group item of MonthName, where the subordinate data item, Month, treats it as an array of size 12 with 3-character

```
01 MonthDef       PIC X(36)     VALUE   "JANFEBMAR
                APR MAYJUNJULAUGSEPOCTNOVDEC".
01 MonthName      REDEFINES   MonthDef.
   02  Month        OCCURS              12     PIC X(3).
```

data strings for each element.

This corresponds to the union of C, but there is no corresponding feature in Java to handle the same data object with different definitions. Translators usually have to define two different classes and copy the corresponding data.

MOVE statement and the CORRESPONDING clause: MOVE statements are frequently used in COBOL programs for assigning values to data items. In moving data between elementary data items, there are several rules that must be followed, such as checking the compatibility rules between the sending and receiving items, a truncation rule when the sending item is longer, and how to fill with spaces or zeros when the sending item is shorter.

If specified with a CORRESPONDING clause in moving data between group items, the sending and receiving items are treated as group items, not as elementary items, based on the group semantics. In the example below, Name and Address within CustomerY receive the values of Name and Address of CustomerX, respectively. Note that types and lengths of the subordinate items of the corresponding items

```
01   CustomerX
  02   Name          PIC A(20)          VALUE  SPACES.
  02   Balance       PIC 9(10)V99       VALUE  ZEROS.
  02   Address       PIC X(50).
01   CustomerY.
  02   DateOfBirth PIC 9(8).
  02   Name          PIC X(15).
  02   Address       PIC X(60).
…..
   MOVE CORRESPONDING CustomerX to CustomerY
```

in the two groups are slightly different, so truncation and space filling occurs as appropriate.

The MOVE statement with a CORRESPONDING clause will be translated into value assignments between corresponding instance variables of two Java objects. Truncating and filling in of characters and digits for String and BigDecimal will require new object allocations, since they are both immutable types.

COPY statement and the REPLACING phrase: COPY statements are similar to "Include" statements in C and C++, and allow users to include external source code or data declarations that are used in common among multiple programs. This is a useful feature for development and maintenance of large software systems, since certain changes may only require an update to the text in the copy library and can then indirectly affect many individual programs.

A simple copy statement includes the library code or a file without any change. The text can be changed as it is copied into the program by using the REPLACING phrase. This is a powerful feature in COBOL and particularly useful for reusing data declarations. In the example below, a single data definition, PersonData, is reused for declaring two different records by renaming the original data name.

```
01   CustomerInfo.
COPY PersonData REPLACING Name BY CustomerName.
01   EmployeeInfo
COPY PersonData REPLACING Name BY EmployeeName.
……
------------- PersonData --------------------
03 PersonInfo  OCCURS  100 TIMES  INDEXED  BY  K.
   05  Name        PIC A(15)     VALUE  SPACES.
   05  Address     PIC X(50)     VALUE  SPACES.
```

COPY statements are often used to define the interfaces of subprogram calls. For each subprogram, the input and output specifications are defined as a separate file, and then this file is copied in the callers for passing arguments and receiving outputs and in the callee for receiving arguments and returning output values. This allows any program to reuse the interface declaration by simply copying the file to make the calls to the subroutine.

However, such uses of COPY statements can be quite expensive when translated to Java, because this is often implemented by defining service classes for the input and output of method calls. COBOL subprograms are not usually written in a modular style, but as very generic service functions that accept various patterns of parameters and that provide a variety of return values (and were often extended as the program was enhanced over the years). Therefore, the interface definitions that will be copied to any caller programs include data items covering all of the cases the subprogram can handle as input or output. For individual callers, however, many of the data items defined in the service classes may be unnecessary.

Open subroutine call with PERFORM statement: An open subroutine is a named block of code that control can fall into or through. It has access to all the data items declared in the main program, but cannot declare its own local data items. This is in contrast to a closed subroutine that can declare local variables which cannot be accessed outside of the subroutine. COBOL supports both open subroutines (using PERFORM statements) and closed subroutines (or subprograms, using CALL statements), but many other languages, including Java, support only closed subroutines.

Open subroutine calls are often used combined with GO TO statements for dealing with errors. When an error is detected during the subroutine's execution, it can stop executing the rest of a paragraph as shown in the example below.

```
PROCEDURE DIVISION.
Begin.
        PERFORM Paragraph1 THROUGH ParagraphEnd.
        STOP RUN.
Paragraph1.
        Statement1.
        IF ErrorOccurs THEN GO TO ParagraphEnd.
        END-IF
        Statement2.
ParagraphEnd.
        EXIT.
```

Although this is dangerous and not a recommended programming practice, it is actually used in many existing applications and translators have to deal with it. Because techniques such as this error handling mechanism, nested PERFORM statements, and falling through from a main program are allowed, open subroutines can produce quite complex control flows. Translators often use the Java exception handling mechanism to manage the control flows from deeply embedded subroutine calls that return to the main program, without regard to whether they are normal flows or exceptional flows. This is shown in Section 3.2.

3. Performance Problems in Translated Java Programs

As shown in the previous section, COBOL has many features that can cause poor performance when translated to Java. Here we describe several major problems and our solutions, based on intensive work with a large Java application translated from its original COBOL version. The application provides a variety of transactional services in response to user requests.

We used the JPROF profiling tool [6] to investigate the execution behaviors of the application. The tool allowed us to track the numbers of instructions executed and the sizes of the objects allocated down to the individual methods.

We categorized the performance problems into four types: eager object allocations, exceptions in normal control flows, reflections in common paths, and inappropriate use of Java class libraries. The following subsections describe each of them in detail.

3.1 Eager Object Allocations

There are many specifications in COBOL that allocate objects eagerly in the corresponding Java program. This type of problem can appear in many different forms in the translated Java program.

String initialization and reinitialization: Elementary data items which are specified with A or X in their PICTURE clauses (e.g. PIC X (10)) are converted to Java strings. By the definition of the PICTURE clause, memory for those strings needs to be reserved for the specified number of characters at declaration time. Usually, the initial values are white space characters.

Translators often define a generic function (as below) to create a space string of a specified length, and replace each string declaration with a call to that function.

```
public static String createSpaceString (int length) {
    // empty string buffer
    StringBuffer spaceString = new StringBuffer (“”);
    while ( length-- > 0 ) spaceString.append ( “  ” );
    return spaceString.toString();
}
```

This is a simple method for the translation process, but is very expensive when executed, since it creates a new instance of space string each time it is called, whether or not the same string already exists in the heap.

To insure the space strings are shared in the heap, we defined a static array of space strings of different sizes, and create new strings based on their sizes, and retain these objects until the application terminates, as shown below.

```
static final int MAXLEN = …;
private static final String [ ] strings = new String [MAXLEN];
private static final String spaceString =
                    new String ( new char [MAXLEN] );

public static String getSpaceString (int length) {
    if (length <= MAXLEN) {
        if (strings[length] == null) {
            strings[length] = spaceString.substring (0, length);
        }
        return strings[length];
    } else return createSpaceString (length);
}
```

This has a similar effect as String.intern(), but is considered even more effective since space strings of different lengths can share the character array from the initially allocated space string of the maximum length. The base structures for this implementation are hidden from users and cannot be modified (protected as private and final). Also, it does not create any garbage after the current transaction executes. Transactions of the same type can be requested many times during the application lifetime, so it is beneficial to retain each space string to minimize the production of garbage objects.

BigDecimal and its initialization: As mentioned in Section 2, all decimal numbers are represented as fixed point in COBOL, and they are usually converted to BigDecimal in Java. This can be another source of overhead for both time (computation) and space (object creation). Since BigDecimal is immutable, each time a new computation is performed, another instance of BigDecimal has to be created.

As with String, care has to be taken for initialization and reinitialization of BigDecimal with zeros (which is the most common case). Instead of creating a BigDecimal object at each declaration of the variable with a specified number of zeros of a scale factor, we created (as shown below) a static array of BigDecimal, with a new instance at the first request of each scale, and retain the objects until the program terminates.

```
static final BigDecimal ZERO = new BigDecimal (0);
static final BigDecimal [ ] bdecimals = new BigDecimal [10];
static final String [ ] zeros =
            {“0”, “0.0”, “0.00”, “0.000”, … “0.0000000000”};

public static
BigDecimal getBigDecimal (BigDecimal value, int scale) {
   if (value == ZERO) && (scale <= 10) {
      if (bdecimals[scale] == null) {
          bdecimals[scale] = new BigDecimal (zeros[scale]);
      }
      return bdecimals[scale];
   } else return
        value.setScale (scale, BigDecimal.ROUND_DOWN);
}
```

Class constructor and arrays: A Java class is defined for each subprogram in COBOL. Since all of the data is allocated and initialized at declaration time in COBOL, the constructors of the Java classes are generated (by translators) to work in the same way, creating objects for all of the instance fields. If a group data item or a table is included in the subprogram declaration, it is usually converted to an inner class or an array in Java, and all of the subfields of the inner class or elements of the array are instantiated.

With this eager allocation of class fields in constructors, and with deep nested hierarchies in the class relationships, it is easy to create huge data structures by merely creating a sin-

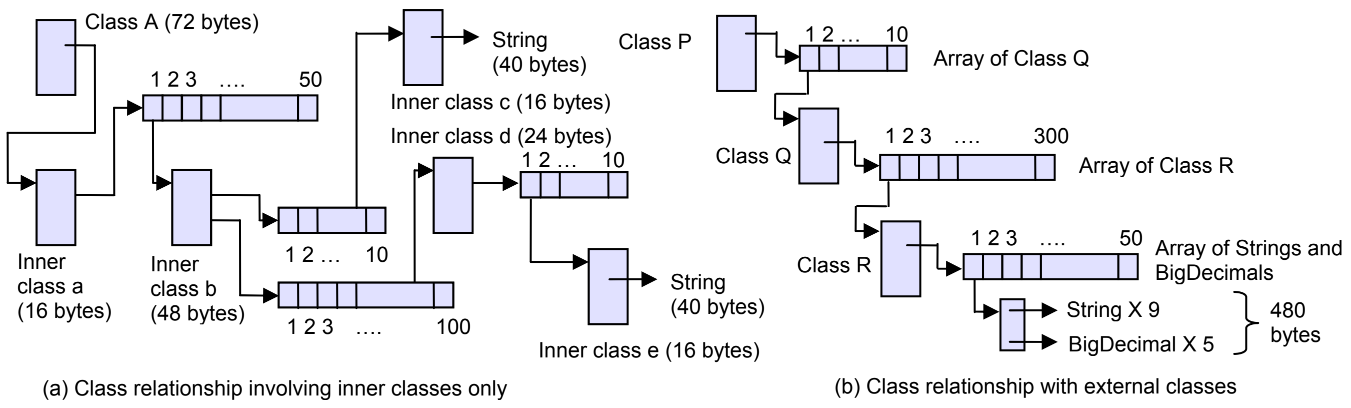

Figure 1. Two examples of class relationships, involving several layers of arrays. Example (a) uses inner classes for all subordinates, while Example (b) uses external classes. Both cases build large data structures by instantiating a single object of the top level class in the hierarchy; Class A accounts for about 3 MB of memory, but Class P accounts for more than 72 MB.

gle object. This is especially true when arrays are involved in the class relationship.

Figure 1 shows two such examples we actually encountered in the production level code. Figure 1 (a) shows a class relationship using inner classes, which indicates that it came from a single group item defined in the original COBOL program, while Figure 1 (b) shows the relationships with all of the external classes. Both cases involve several arrays in the layers of the class relationship, and instantiating the top level class in the hierarchy results in building a large data structure. Class A in Figure 1 uses about 3 MB (calculations omitted) of memory, and Class P consumes more than 72 MB (10 x 300 x 50 x 480) of memory, assuming that we use a 32-bit JVM and 8-byte object headers, so each string object costs 40 bytes, while each BigDecimal object needs 24 bytes, both including object headers.

If this is in a frequently executed path (for example, the class is instantiated each time a frequently used transaction is invoked), it can create several hundred objects that are simply thrown away at the end of each transaction, causing unacceptably frequent garbage collection. Even worse, significant parts of the initially allocated memory are not used at all during individual transactions. This is because the original COBOL program was designed to handle similar but different functions in a single subprogram, and the variables are declared to cover all of the cases.

```
public class ClassA {
    // an array instance field
    private ClassX [ ] xArray = new ClassX [ 101];
    public ClassA () {
        // constructor does not instantiate elements of the array
    }
    public ClassX getXArray ( int i ) {
        if (xArray [i] == null) {
            xArray[i] = new ClassX ();
        }
        return xArray [i];
    }
    public void setXArray ( int i, ClassX value ) {
        xArray [i] = value;
}  }
```

Note that these are not atypical cases. We encountered many similar examples. Translators usually process each subprogram independently, and produce a class with a constructor allocating those objects for the instance fields. This works well for generating functionally correct Java programs, but it causes excessive object allocations that are often detected at the very end of the testing cycle. It is extremely difficult to detect this kind of problem without having a complete picture of the class relationships.

We can solve this problem by modifying the constructor, getter, and setter methods for the private instance fields of arrays to use lazy instantiation of array elements.

```
public class ClassB {
    public void execute () {     // entry method
        try {
            initialize ();
            main ();
        } catch (RuntimeExceptionClassB e) {
    }  }
    private initialize () {
        // do some initialization task
        if (errorOccurred) exit();
    }
    private main ( ) {
        // do actual business logic, including database access
        exit ();
    }
    private final exit () {
        // some error handling logic
        throw new RuntimeExceptionClassB ();
}  }
```

3.2 Exceptions in Normal Control Flows

In Section 2, we mentioned that open subroutines using PERFORM statements are often translated to use exception mechanisms in Java. This works well to ensure the same flow of control is executed in the translated business logic as in the original version, but it tends to introduce exceptions in normal control flows, causing significant overheads.

In the above example, the intention was probably to provide an error-handling routine and call it at all of the exit points in the subroutine that raise error conditions, including after the database access in the main program in the original COBOL program. Human programmers would never do this, but due to inappropriate conversion logic, translators can generate this performance-limiting code.

3.3 Reflections in Common Paths

A COBOL table is translated into an array in Java. The elements of the array are usually initialized with instances of a specified class (unless the proposed changes described in Section 3.1 are used). While executing a transaction, some elements of the array will change. For example, if the array is used to hold objects extracted from a database, then some elements may be updated with new objects, while others are left in their initial state, depending on the number of records fetched for a particular query. Later in the transaction, the array may be traversed to find the number of updated elements by checking each element against the initial state.

For this business logic in the original COBOL, translators tend to create a single generic function that compares two arbitrary objects with the reflection mechanism, and use that function to perform comparisons for any type of array element. As shown in Figure 2, the function accepts two objects, comparing their classes first and then the defined methods, including names and parameters. It even recursively invokes the methods and checks the type of the objects returned. With reflection, this function can compare any pair of objects to each other.

The performance of reflection has been significantly improved in Java version 1.4 and later [11], but it is still better not to use reflection in frequently executed paths. We modified the code to avoid the use of the generic function, and instead call a new method added to each class to check its initialized state. If all of the arrays are guaranteed to use dynamic allocation for their elements, then a simple null check should suffice instead of the object equality check.

3.4 Inappropriate Calls to Class Libraries

Translators generally try to use class library functions as often as possible to simplify the translation process. For example, classes and methods for dates and timestamp services are called frequently since COBOL is by design a business language and many business applications use dates and timestamps quite often.

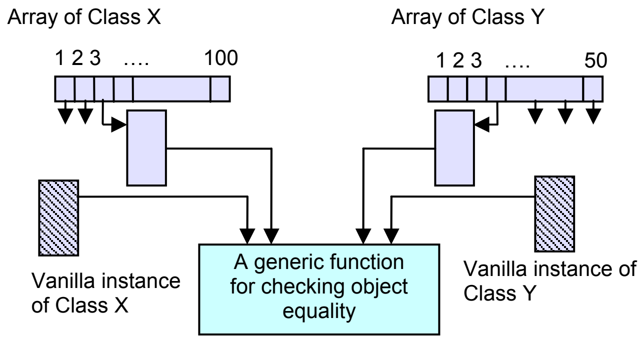
Array of Class X
1 2 3 100
Array of Class Y
1 2 3 50
Vanilla instance of Class X
A generic function for checking object equality
Vanilla instance of Class Y

Figure 2. An example of using reflections in common paths. A generic function accepts two objects of any type and checks the equality between the two objects as an array element mutation check.

The class java.text.SimpleDateFormat () provides methods for formatting Date objects into date/time strings and, in reverse, for parsing date/time or timestamp strings to create various types of objects, based on specified PATTERN strings. It is useful for translators to call library functions for the date and time manipulations. However, this simplified translation process creates additional overhead. Both format () and parse () provided in this class internally create many temporary objects. Simply to cross the library interface, the application has to convert the data by creating another object (e.g. Calendar). Worse, since these methods update the instance fields of the object as working space,

```
static final String PATTERN = "yyyy-MM-dd";
private static char filler = '-';
private static char digits[ ] = {'0','1','2','3','4','5','6','7','8','9'};

public String getDate (int day, int month, int year) {
    try {
        char tmp [ ] = new char [10];
        tmp [0] = digits [year / 1000];
        tmp [1] = digits [(year % 1000) / 100];
        tmp [2] = digits [(year % 100) / 10];
        tmp [3] = digits [year % 10];
        tmp [4] = filler;
        tmp [5] = digits [month / 10];
        tmp [6] = digits [month % 10];
        tmp [7] = filler;
        tmp [8] = digits [day / 10];
        tmp [9] = digits [day % 10];
        return new String (tmp);
    } catch (ArrayIndexOutOfBoundsException e) {
        calendar = new GregorianCalendar ();
        calendar.add (GregorianCalendar.YEAR, year); ….
        return new SimpleDateFormat (PATTERN).
            format (calendar.getTime());
}   }
```

Table 1. List of transactions used in the experiment

Name	Description	# of classes	# of methods	# of db reads	# of db writes
Transaction 1	Calculate decimal numeric values, generating future profile	1,329	5,001	515	87
Transaction 2	Calculate decimal numeric values for a specified condition	1,629	6,212	360	0
Transaction 3	Get information on records over a specified time interval	822	2,818	5	0
Transaction 4	Get information on structured records	815	2,825	11	0
Transaction 5	Create profile information from a set of specified records	980	3,580	189	0
Transaction 6	Get information from a given record number	511	4,127	21	0
Transaction 7	Validate required changes to a given record	1,468	13,641	783	7
Transaction 8	Create a set of new records with a specified customer data	1,840	6,141	139	21

creating a static instance of this class with a specified PATTERN string does not work (because it is thread unsafe). The problem was also reported in [8].

We can skip the expensive library calls for date and timestamp services in many cases. For example, using the ThreadLocal class to create thread-safe SimpleDateFormat and Calendar objects is one way to alleviate the problem. However, the above code works even better to replace format () when creating a date string in a specified format from the integer values of day, month, and year. Only in exceptional cases (as when out-of-range values are specified), need the library method be called. This code was verified to be approximately four times faster and to produce fewer than one-third as many temporary objects.

4. Experimental Results

As noted, we looked into the performance problems of a large Java middleware system that has been translated from an original COBOL version. The total number of classes in this application is nearly 60,000, and there are approximately 700 database tables. In this section, we show how the performance of this middleware can be related to these problems by using real-world transaction scenarios.

4.1 Methodology

Table 1 shows a list of the transactions we used in the evaluation. These are all real-world transaction scenarios. The numbers of classes, methods, and database reads and writes show the relative complexities of the transaction scenarios. These are all dynamic numbers. The operational characteristics of each kind of transaction processing are also described. The input data was selected to exercise all of the necessary operations for each transaction.

We implemented performance fixes for the middleware in a way that can be switched on and off for each problem type described in Section 3. We did all those fixes manually by directly modifying the translated Java code (not through the translator improvements). When comparing the performance impacts on a particular problem with several configurations, we restored the database tables (for transactions including update operations) and used the same input data to ensure that same paths and operations were executed within the middleware code.

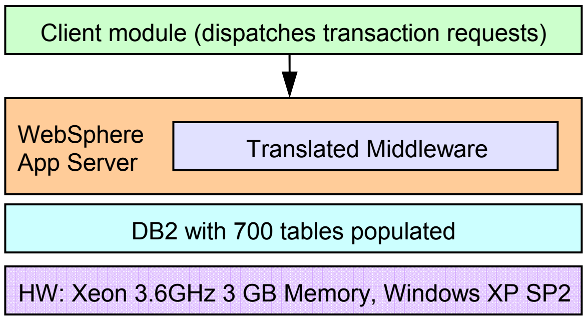
Client module (dispatches transaction requests)
WebSphere App Server
Translated Middleware
DB2 with 700 tables populated
HW: Xeon 3.6GHz 3 GB Memory, Windows XP SP2

Figure 3. Experimental environment. The translated middleware is running on WebSphere Application Server and executes transactions, accessing the database tables.

Figure 3 shows the environment for this experiment. We performed all measurements with a simple one tier configuration, running the translated middleware on the WebSphere ND (Version 6.0.2), DB2 (Version 8.2.6), and a client module that dispatches a variety of transaction requests on a single node. The client requests a single transaction at a time. The machine is an IBM IntelliStation (Xeon 3.6 GHz processor with 3 GB memory), running Windows XP SP2, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, Version 1.4.2. Both the initial and maximum heap sizes were set to 1,024 MB.

In order to exclude the effects of JIT compilation overhead from the measurement, we adjusted the threshold to invoke the JIT compilation and ran a sufficient number of warm-up runs before starting the measurements. After starting each measurement, we collected the results for 10 successive runs, and calculated the average.

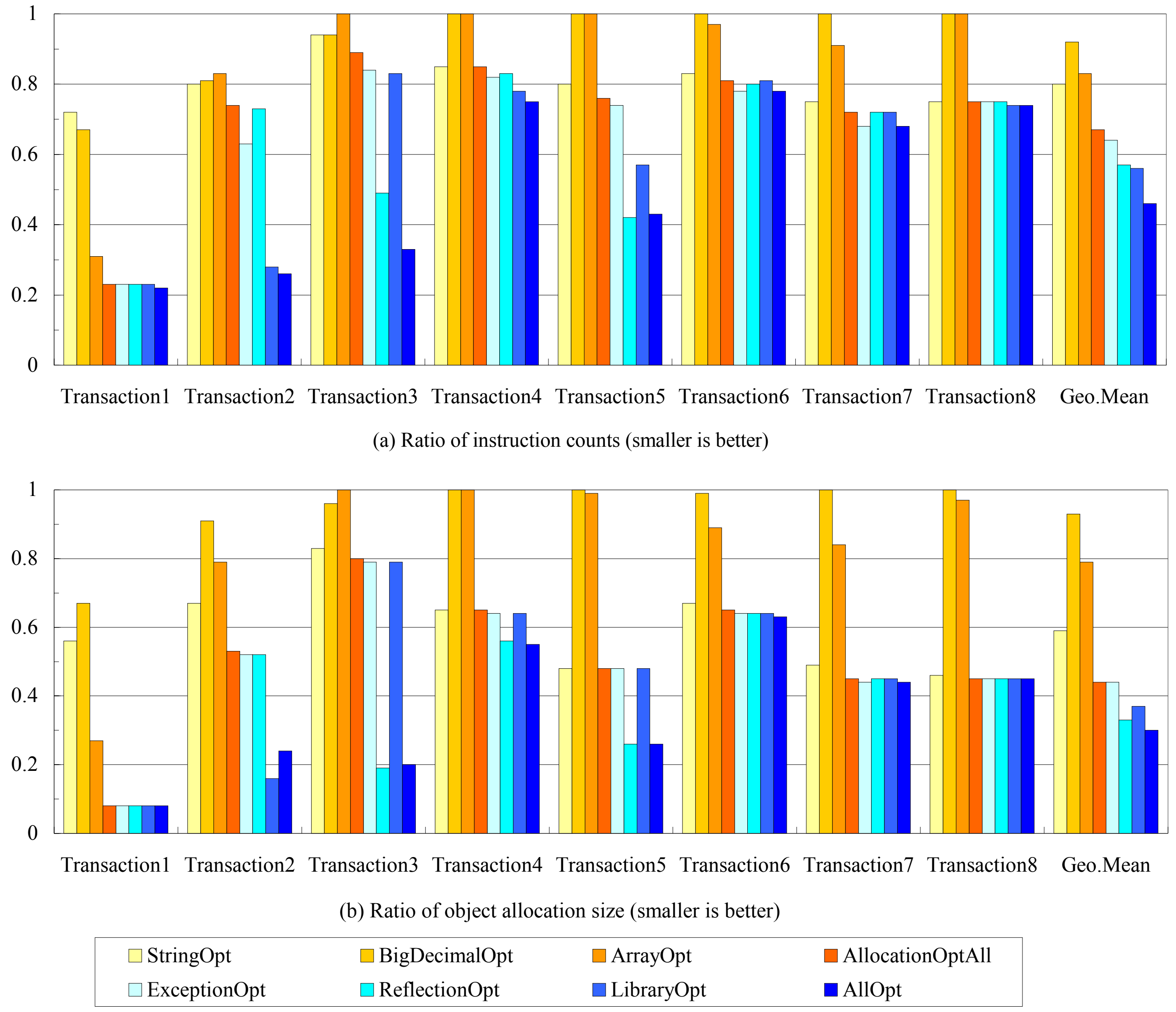

Figure 4. Results from fixing eager object allocations and other performance problems in terms of (a) ratios of instruction counts and (b) ratios of object allocation size. The baseline of the comparison in this graph is NoOpt (value of 1.0)

We first show the impacts of addressing the eager object allocations (Section 3.1) and other performance problems (Section 3.2 through 3.4) on the reduction of instruction counts and the memory used for object allocation in Section 4.2. We then present the end-to-end performance improvements for each transaction when addressing all of the problems in Section 4.3.

4.2 Impacts on Java Performances

Figure 4 shows the reduction ratios of (a) instruction counts and (b) size of total object allocations for each transaction, collected by profiling with JPROF. The baseline of the performance comparison is NoOpt (all optimizations disabled). The four bars in the left (colored various shades of orange) show the following optimizations.

1. **StringOpt** shares space characters of String objects at initialization and reinitialization time.
2. **BigDecimalOpt** shares initial zero values of BigDecimal objects at initialization and reinitialization time.
3. **ArrayOpt** creates objects for array elements in an on-demand basis, avoiding eager object instantiations in class constructors.
4. **AllocationOptAll** enables all three optimizations 1 to 3, showing the cumulative effects for fixing eager object allocations.

Since the eager object allocations have significantly larger impacts on performance than the other problems, we first present the results when addressing the eager object allocation problems. On top of these fixes, we then show how much gain we can achieve by addressing the other problems later in this section.

Fixing eager object allocations is particularly effective for large transactions (in terms of execution time; e.g. Transactions 1, 2, 7, and 8). The size of object allocations was drastically reduced to 8% for Transaction 1 and to less than 50% for Transactions 7 and 8, with all optimizations enabled. These transactions consumed over 300 MB of Java heap memory per transaction in the original version, causing frequent invocations of garbage collection. Most of the memory used was simply wasted with these eager allocations for initializations.

Which of the three optimizations is the most effective depends on the nature of the transactions. For example, Transaction 1 uses a large deeply nested array of data structures for its working space (as illustrated in Figure 1), and ArrayOpt alone has a large impact on reducing the total memory consumption. Both Transactions 1 and 2 perform lots of decimal numeric calculations, and they both heavily use BigDecimal variables. Therefore, the effect of BigDecimalOpt on these two transactions is much higher than for the other transactions. The optimization on String is consistently effective for all transactions.

For smaller transactions (such as Transactions 3 and 6), the effect is less dramatic, but still the eager object allocation contributes around 10% to 20% reductions of instruction counts, and 20% to 30% reductions of the total heap consumption. From these results, it is clear that eager object allocations are pervasive over a variety of transactions in the translated Java programs.

From the above results, we think it is mandatory to fix the problems with eager object allocations before addressing the other problems, since the impact is significantly larger and pervasive. Thus, we simply incorporate all of those fixes (AllocationOptAll), and *on top of that*, the performance impacts caused by the other problems were studied. The four bars on the right (colored various shades of blue) in Figure 4 show the following optimizations.

5. **ExceptionOpt** eliminates exceptions thrown in normal control flows in transaction executions (described in Section 3.2).
6. **ReflectionOpt** eliminates reflection used in commonly executed paths in transactions (described in Section 3.3).
7. **LibraryOpt** eliminates expensive class library calls (described in Section 3.4).
8. **AllOpt** applies all the above optimizations

The effect of the three optimizations (5 to 7) is not as widespread across the transactions as the optimizations for eager object allocations. ExceptionOpt is useful for Transaction 2 with roughly a 15% to 20% reduction of instruction counts. We think this transaction executes code that was implemented using many open subroutines in the COBOL.

ReflectionOpt looks particularly effective for Transactions 3 and 5 (and to a lesser extent, for Transaction 4), but this optimization actually produces almost equal contributions for all of the transactions. Transactions 3 and 5 are relatively small, and this means the optimization has a larger impact on those transactions.

Transaction 2 does some computations related to the time intervals over specified periods in the past, so there are lots of date and timestamp operations during the transaction execution. LibraryOpt has a large impact on this transaction, reducing both instruction counts and object allocation to less than 40% of the level after the optimizations for eager object allocations.

4.3 End-to-end Performance

Figure 5 shows the end-to-end performance (response time) improvements with all of the optimizations enabled. The (blue) triangles in the graph show the response times with the translated code (NoOpt) and the (purple) squares show the response times with all of the optimizations enabled (AllOpt). After the warm up runs (to the steady state), we measured 20 consecutive runs for the response time of each transaction. Since Transactions 3 to 6 are small, we iterated several times for each point in the graph to improve the timer resolution. The plots in the graph are normalized relative to the average response times with NoOpt over 20 runs.

The database accesses in the application were generated by the translator from the original embedded SQL in the COBOL to SQLJ in the Java programs. In this experiment, we did not perform any database access optimizations at all, such as caching. Thus a fair amount of the fixed costs for database accesses, e.g. more than 50% of the total response time for some transactions, were included in the end-to-end performance. The data presented here is based solely on the optimizations on the Java applications.

The performance was consistently improved for all of the transactions. The largest improvement was observed in Transaction 3, about 40%, followed by Transaction 2, between 25% to 30%. Many other transactions achieve around 10% to 20% improvements. This is significant, especially for large transactions (such as 1, 2, 7, and 8), where there are large numbers of database accesses, as shown in Table 1.

The spikes in the graph show where garbage collection occurred. As we can clearly see from the graphs for Trans-

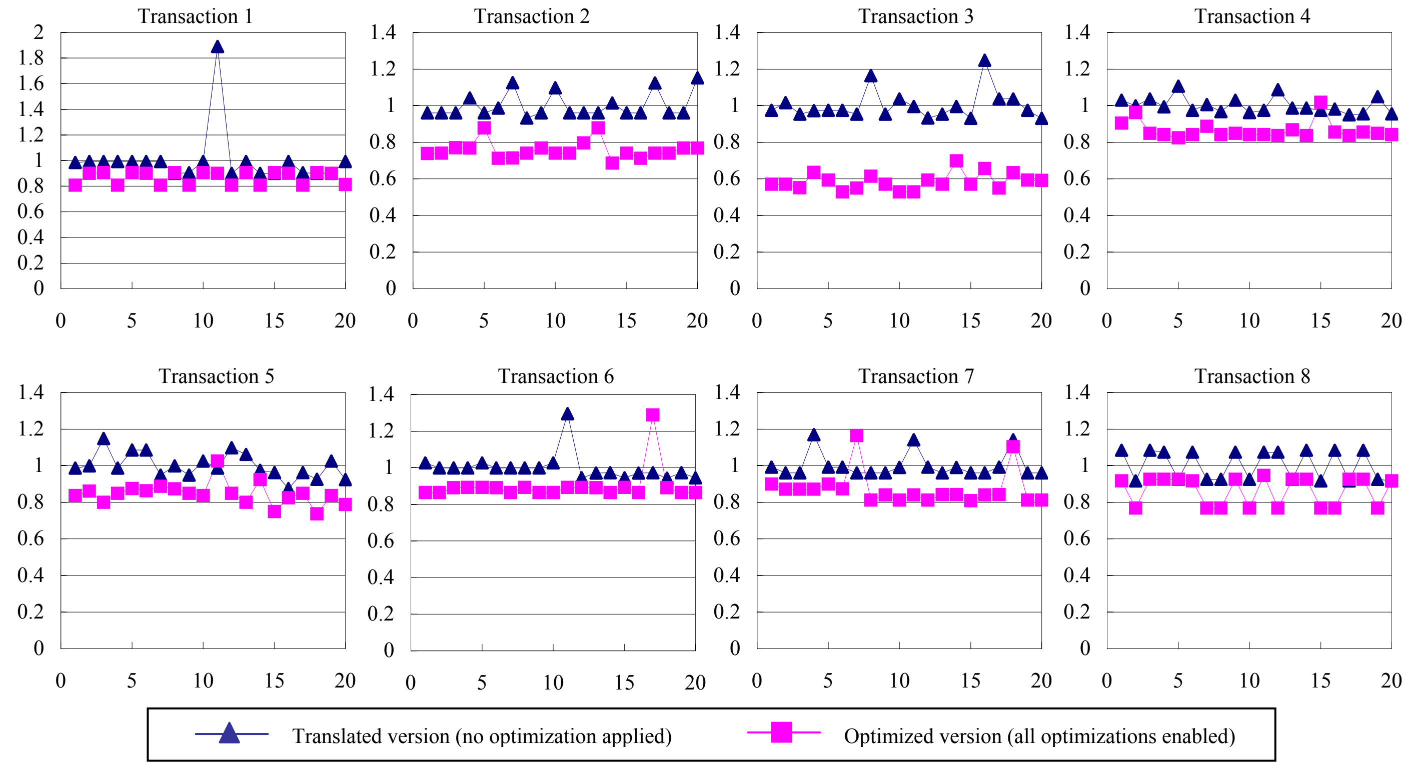

Figure 5. End-to-end performance improvements on Transactions 1 to 8. The (blue) triangles are with all optimizations disabled (NoOpt) and the (purple) squares are with all enabled (AllOpt). After the warm up runs, we measured the response times of 20 consecutive runs for each transaction. The X-axis is the time for the run and the Y-axis is the normalized response time relative to the average response time with NoOpt over 20 runs.

actions 1, 2, 4, and 7, the frequency of the garbage collection was reduced by our optimizations to eliminate eager object allocations.

From these results, we can see that our optimizations on the translated middleware produced significant improvements in the response times for all of the transactions.

5. Related Work

There are a number of tools and services currently available in the market for fully automatic or semi-automatic COBOL to Java translation. SoftwareMining provides a toolkit (called CORECT) that translates COBOL to Java or C# [10]. The tool's primary objective is to generate "legible and maintainable" code. Approximately 95% of the legacy code is transformed automatically and the remainder is dealt with in a semiautomatic manner. During the transformation, a program analyzer uses heuristics for some optimizations, such as identifying data usages, identifying and removing dead code, and simplifying REDEFINE clauses.

Relativity [9], EvolveWare [5], and Datatek [3] all support legacy application migration tools, including for COBOL to Java. Some of these employ sophisticated tools for smooth transitions, such as the Modernization Workbench from Relativity or the automated code analysis from Datatek. However, the goal of these tools is to make the final translated program as readable and maintainable as possible while generating sufficient documentations, and performance seems to be a secondary concern. Our work in this paper exposes the serious performance problems that can pervade Java programs translated from COBOL, based on extensive study of large production middleware translated from COBOL.

Mitchell and Sevitsky [7] studied the details of the Java memory bloat problems for many applications and benchmarks. They classified the memory usage of Java objects into actual data and the rest (infrastructure overhead such as object headers of individual instances and entry instances for collection structures), and showed the actual data occupies only a small fraction of the total heap consumption. Based on extensive study, they proposed a health metric, the ratio of actual data to the total, and how its application-neutral asymptotic behavior can be linked to the underlying program design. The eager object allocations in our study of the COBOL-translated Java programs pose more serious problems than what they addressed in ordinary Java programs, since these data structures have actual data, but are not used in a meaningful way during program execution.

An interesting question involves the different health signatures between the two languages, COBOL and Java. For example, a COBOL group item does not impose any memory overhead when gluing its elements together into a collection structure, as described in Section 2. Therefore using collections is a good practice in COBOL and improves the health metric, but lowers the metric for the translated Java. In contrast, the static nature of the COBOL storage system results in lots of white space allocated in the data fields, without actual data. In general, as an application-neutral metric, the health signatures of snapshots of COBOL programs can be considered much higher than those of the translated Java programs. If the health metric can be estimated from the translated Java source program, that may be an indicator of the program's performance.

The inappropriate use of Java class libraries described in Section 3.4 is just one form of larger problems in the framework-based application development. Mitchell et al. [8] studied the runtime inefficiency of the framework-based applications, and showed inefficient processing of date and timestamp data when parsing XML texts in the SOAP framework. Since it is extremely difficult for both translators and human programmers to be aware of the runtime costs that must be paid by calling library routines or framework-provided services at development (translation) times, new optimization techniques are needed for runtime or deployment time, to span several layers of components.

6. Conclusions

We described the details of some performance pitfalls of large scale Java applications that were translated from COBOL. As mentioned, this study is based on our experiences, investigating performance problems in mission-critical commercial middleware code that was actually translated from COBOL using automatic translation. There are many differences, particularly in storage management, between the two languages, and a direct and naive translation may lead to high pressure on the Java heap and can cause unacceptable amounts of garbage collection.

We classified the performance problems into four categories: eager object allocations, exceptions in normal control flows, reflections in common paths, and inappropriate uses of the Java library. The eager object allocation was found to be the biggest and most serious problem. Because of the differences in the language specifications, translated Java programs often encounter performance pitfalls that would never occur in programs written by human programmers.

There are strong demands to port COBOL to Java for business applications in many industries, and the need is expected to increase in the near future for modernizing large assets instantiated in legacy programs. However this is not practical unless the performance of the translated code is close to that of the original COBOL versions. We showed where we need to focus in the translated Java code to improve the performance, and in the future, we hope we can apply this work to improve a translator or to create a refactoring tool that supports translated Java programs.

Acknowledgments

We are grateful to Hiroshi Arai of IBM Japan for technical materials and suggestion on the COBOL-to-Java translation.

References

[1] COBOL programming – tutorials, lectures, exercises, examples. http://www.csis.ul.ie/cobol/default.htm.

[2] S. Chandra, J. de Vries, J. Field, H. Hess, M. Kalidasan, K. V. Raghavan, F. Nieuwerth, G. Ramalingam, J, Xue. Technical Forum: Using Logical Data Models for Understanding and Transforming Legacy Business Applications. *IBM Systems Journal* 45(3), 2006.

[3] DATATEK Inc, http://www.datatek-net.com/dtk_cobol.htm?

[4] D. F. Ferguson, and M. L. Stockton. Service-oriented architecture: programming model and product architecture. *IBM System Journal*, 44(4), 2005.

[5] Evolveware Inc, Legacy Rejuvenator, available at http://www.evolveware.com/lr.html.

[6] IBM Corporation. JPROF, open source version available, http://sourceforge.net/projects/perfinsp.

[7] N. Mitchell and G. Sevitsky. The Causes of Bloat, The Limits of Health, In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 245–260, 2007.

[8] N. Mitchell, G. Sevitsky, and H. Srinivasan. The Diary of a Datum: An Approach to Modeling Runtime Complexity in Framework-Based Applications, Workshop on Library-Centric Software Design (LCSD), at *the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[9] Relativity Technologies, http://www.relativity.com.

[10] SoftwareMining Inc, The COBOL Transformation Toolkit: CORECT, available at http://www.software-mining.com/services/translation.jsp

[11] D. Sosnoski. Java programming dynamics: Introducing reflections. http://www.ibm.com/developerworks/library/j-dyn06