

Java Method Modeling using Application Response Measurement (ARM)

Dr. Carl J. De Pasquale
College of Staten Island
Staten Island, NY

Abstract - *What should not be surprising, especially to anyone who has attempted to build a JAVA model, is that collecting method level data is difficult. The goal of this paper is to describe how to create Java method models with performance data obtained from ARM instrumented code. The paper begins by discussing how to instrument Java code using ARM and concludes by presenting an overview of analytical Java method modeling.*

Key Words

Java, ARM, User Workflow, Operational Analysis, Modeling

1 Introduction

Early research of into application modeling began during the 1970's when Buzen [1] designed a method to identify system bottlenecks using a network queuing model. Proposed by Basket [2] is a slightly different approach using a balanced queuing strategy named BCMP. Shortly after BCMP was introduced Buzen [3], provided a set of operational quantities from which reasonably accurate performance predictions could be made. During early 1980's, Buzen developed an analytical modeling and performance prediction tool based on the theories of operational analysis.

The middle 1980's saw a drastic reduction in the cost of computer operation. Smaller, less expensive and seemingly more adaptable distributed configurations began to overtake the dominant centralized computing model of the 1960's, 1970's and early 1980's. Poorly performing applications were probably no longer as important since many performance problems could now be resolved by upgrading the distributed configuration [4]. This approach remained a common strategy until the early 1990's, at which time Sun [5] introduced the highly portable, interpretive, and resource intensive Java programming environment and language. Java's write once, run anywhere interpretative architecture propelled universal access to a universal expectation where, hardware alone could no longer compensate for inefficiently designed and poorly implemented software.

The novel aspects of this work are:

- Method measurements are collected during production
- Method models are created

The remainder of this paper is organized as follows. Section 2, presents an overview of ARM. Section 3 explains details required to implement ARM in Java applications. Section 4 discusses acquiring Java CPU and memory metrics and section 5 discusses building method models. Section 6 presents future work and section 7 provides commentary and conclusions.

2 ARM Overview

The ARM standard [6] describes a software instrumentation technology that when combined with existing system management infrastructures facilitate JAVA self-discovery and self-monitoring capabilities. Self-discovery refers to any application that can generate, during execution, detailed workflow traces. Self-monitoring refers to any application that can generate application modeling and sizing metrics, such as method elapsed and CPU times, method invocation counts, and memory usage, etc.

To ARM a JAVA application the software engineer will need to download the ARM 4.0 Application Program Interface (API) library and optional ARM 4.0 SDK from www.opengroup.org/tech/management.arm/. Once the ARM class libraries are downloaded and installed the JAVA application can be instrumented with ARM 4.0 APIs.

As shown in Figure 1, when an ARM instrumented JAVA application executes it invokes ARM APIs, which dispatch messages to the ARM agent. The ARM agent performs the requested action by returning information to the application and/or inserting ARM data records into the ARM data repository [ARMJ04].

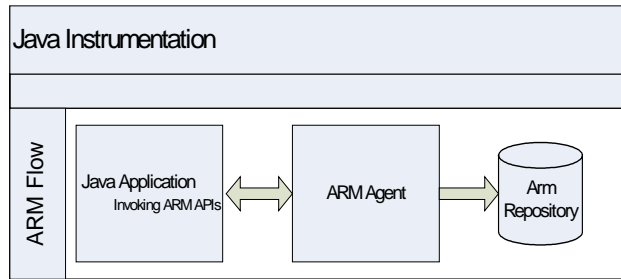


Figure 1: JAVA/ARM Interface

2.1 Arming Hello World

To instrument a Java application five ARM objects are required. The five objects are 1) Transaction Factory, 2) Application Definition, 3) Transaction Definition, 4) Application, and 5) Transaction. Once these objects are instantiated, the software engineer surrounds an application transaction with an ARM “*Transaction.start*” and “*Transaction.stop*” API. The start-stop pair requests that the ARM agent record the transaction’s elapsed time into the ARM repository. Finally, to signal the application has concluded, a call to the ARM “*Application.end*” API is made. The “HelloWorld” example shown in Figure 2 illustrates the ARM structure.

```

import
org.opengroup.arm40.transaction.ArmApplication;
import
org.opengroup.arm40.transaction.ArmApplicationDefinition;
import
org.opengroup.arm40.transaction.ArmTransactionFactory;
public class HelloWorld {
public static void main(String argv[]) throws Exception {
    ArmTransactionFactory txFactory;
    If ((txFactory=new ArmTransactionFactory())==null)
return;
    ArmApplicationDefinition appDef
        txFactory.newArmApplicationDefinition
            ("HelloTranApp", null, null);
    ArmTransactionDefinition tranDef =
        txFactory.newArmTransactionDefinition(
            appDef, "HelloWorldJava", null, null);
    ArmApplication app =
        txFactory.newArmApplication(appDef,
            "Hello", null, null);
    ArmTransaction tran = txFactory.
        newArmTransaction(app, tranDef) ;
    tran.start();
    System.out.println("Hello world!"); //The
Transaction
    tran.stop(ArmConstants.STATUS_GOOD);
    app.end();
}
}

```

Figure 2: ARMed HelloWorld

3 ARming a Java Application

In addition to the ARM objects discussed in section 2, three additional coding abstracts are needed to ARM JAVA applications. 1) A physical JAVA application can consist of several logical JAVA applications. To correctly associate an application to its transactions each application must be uniquely identified. 2) To implement a self-discovery capability (which enables user workflow traces) the ARM correlation feature is required. 3) Self-monitoring (the capability that provides dynamic application modeling, sizing measurements and root-cause identification) requires the collection of application data beyond transaction elapsed time and correlation.

3.1 Application to Transaction Mapping

One approach to associate a JAVA application to its transactions is to create a simple static class, one class for each application. Within the static class, create three ARM objects, 1) Transaction Factory, 2) Application Definition, and 3) Application. Figure 3 provides a code snippet.

```

import org.opengroup.arm40.transaction.ArmApplication;
import
org.opengroup.arm40.transaction.ArmApplicationDefinition;
import
org.opengroup.arm40.transaction.ArmTransactionFactory;
public class ArmAppTrans {
public static ArmTransactionFactory txFactory =
    (ArmTransactionFactory) new
ArmTransactionFactory();
public static ArmApplicationDefinition AppDef =
    txFactory.newArmApplicationDefinition
        ("ApplicationName", null,
null);
public static ArmApplication App =
    txFactory.newArmApplication(AppDef, null, null,
null);
public ArmAppTrans() { super(); }
}

```

Figure 3: Application Static Class

To access the static objects, simply import the package and reference its name. Figure 4 shows how a transaction definition and a transaction are created.

```

ArmTransactionDefinition armTransDef =
ArmAppTrans.txFactory.newArmTransactionDefinition
(ArmAppTrans.AppDef, "TransactionName", null,
null);
ArmTransaction armTransaction =
    ArmAppTrans.txFactory.newArmTransaction
        (ArmAppTrans.App, armTransDef);

```

Figure 4: Transaction Definition

3.2 User Workflow

A user workflow is an ordered set of application transactions where the first transaction identifies the user's login, the second through last-1 transactions detail the flow of execution, and the last transaction identifies the user's logout. The ARM correlation objects make it possible to link individual user transactions into a representative user workflow trace.

The correlation strategy can be thought of in terms of three stack operations, push, peek and pop. When a user login transaction (which is identified as the parent) is run, a correlation token is created and associated to the transaction to push the transaction onto a stack. Subsequent transactions create new correlation tokens, peek the stack to obtain the previous transaction and reference the correlation token. This procedure establishes the transaction lineage. New transactions repeat this process and the terminating transactions will pop the stack. Pictorially, this is illustrated in Figure 5.

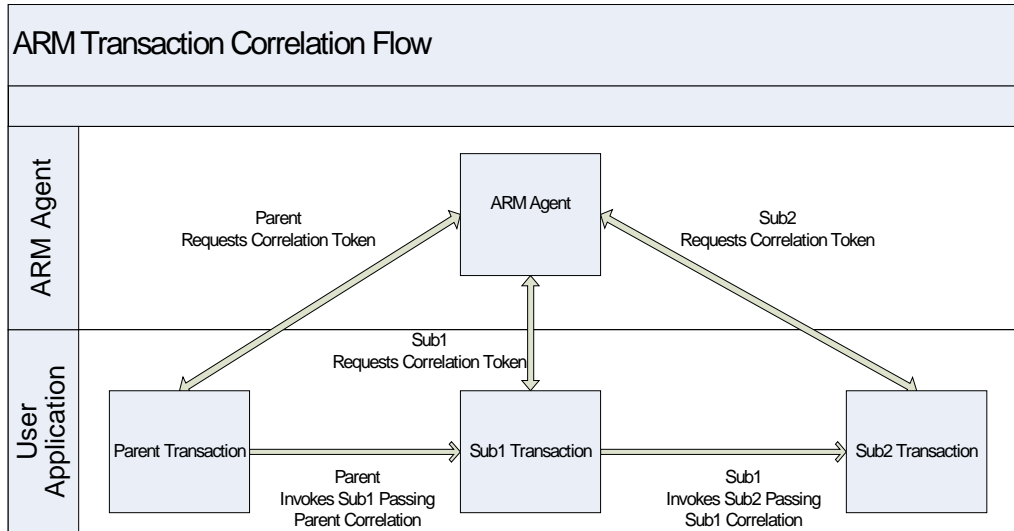


Figure 5: ARM Correlation

The implementation details that need to be discussed are: 1) How is the stack created and accessed, and 2) How do transactions associate the correlation token to the transaction? Both are relatively straightforward. Figure 6 shows that the stack is created using the WebSphere Performance Monitoring Infrastructure Request Metrics (PmiRmArmTx) class. The use of PmiRmArmTx and its corresponding methods is, of course implementation dependent, but regardless of application server the concept described here is representative. The stack is instantiated using createPmiRmArmTx() method, then all subsequent stack operations are anchored off the stack instance variable, arTranx. As shown, a previous correlation token, if available, is acquired by "peeking" the stack. If the stack is empty then this transaction is the parent. When a new transaction begins, it requests a new correlation token and associates itself with the correlation token obtained from the stack. Finally, the ARM transaction containing the new correlation token is pushed onto the stack.

```
ArmCorrelator arc = null ; //Parent
ArmTransaction armTransaction
```

```
armTransaction.setUser( UserId);
armTransaction.start(arc);
PmiRmArmTx arTranx = //Parent-Create a stack instance
PmiRmArmTxFactory.createPmiRmArmTx();
/ Create a transaction correlation token and associate
// it with an ARM transaction
ArmCorrelator arc = // null when Parent
ArmTransApp.txFactory.newArmCorrelator(
PmiRmArmStack.peekTransaction().getCorrelatorBytes());
arTranx.setArmTransaction(armTransaction);
PmiRmArmStack.pushTransaction(arTranx);
-----
---
ArmCorrelator arc = null ; //Child
ArmTransaction armTransaction
PmiRmArmTx arTranx = //Child- Create a stack instance
PmiRmArmTxFactory.createPmiRmArmTx();
//obtain correlation token from the top of the stack
try {
arc = //peek stack and get Parent Correlator
ArmTransApp.txFactory.newArmCorrelator(
PmiRmArmStack.peekTransaction().getCorrelatorBytes());
} catch (Exception e) { //handle exception }
armTransaction.start(arc);
// push the transaction that contains the correlation token
arTranx.setArmTransaction(armTransaction);
PmiRmArmStack.pushTransaction(arTranx);
-----
// pop stack when transaction completes
PmiRmArmStack.popTransaction();
```

Figure 6: PmiRmArmTx

3.3 Collecting Application Metrics

The `ARMTransactionWithMetrics` object extends the `ARMTransaction` by including seven optional metric slots. Six are numeric (counter, gauge, etc) and the seventh is a string. Using `ARMTransactionWithMetrics`, applications can provide extremely detailed runtime (method) measurements, which are needed to construction Java method models, help determine infrastructure sizing, and facilitate root-cause problem analysis. Infrastructure sizing and root-cause problem analysis are byproducts of ARM instrumentation. For example, many ARM APIs store data in a repository. This information can be used to construct an application management console.

Defining the components of an `ARMTransactionWithMetrics` object, while not difficult is somewhat more involved than simple `ARMTransaction`. However, it should be noted that `ARMTransactionWithMetrics` definitions can be referenced by all transactions and would need to be defined only once during application initialization.

The ARM technical specification [ARMJ04] defines 4 data types (counter, gauge, slider, string). For each metric that will be associated with an `ARMTransactionWithMetrics` object, a variable object/variable definition object pair, and a grouping object must be defined. The variable definitions are grouped using the `newArmMetricGroupDefinition` and finally the metric itself is associated with its definition. To assign a value to an object variable the objects' "set" method is used. The example in Figure 7 shows the setup required for one "counter" metric.

```
import org.opengroup.arm40.metric.ArmMetric;
import rg.opengroup.arm40.metric.ArmMetricCounter32;
import
org.opengroup.arm40.metric.ArmMetricCounter32Definition;
import org.opengroup.arm40.metric.ArmMetricDefinition;
import org.opengroup.arm40.metric.ArmMetricFactory;
import org.opengroup.arm40.metric.ArmMetricGroup;
import
org.opengroup.arm40.metric.ArmMetricGroupDefinition;
import
org.opengroup.arm40.metric.ArmTransactionWithMetrics;
import org.opengroup.arm40.metric

ArmTransactionWithMetricsDefinition;

ArmTransactionFactory txFactory = null;
ArmMetricFactory txMetricFactory = null
ArmMetricCpuDefinition cpuDef = null;
ArmMetricCpu cpu = null;
ArmMetricGroupDefinition metricGroupDef = null;
ArmMetricGroup metricGroup = null;

txFactory = ArmTransApp.txFactory;
txMetricFactory =
```

```
(ArmMetricFactory) new
ArmMetricFactory();

txMetricFactory.newArmMetricCpuDefinition(appDef, "Cpu",
"Cpu", ArmMetricDefinition.METRIC_USE_GENERAL, null)

metricGroupDef =
txMetricFactory.newArmMetricGroupDefinition
new ArmMetricDefinition[]{cpuDef, null, null, null,
null, null, null
});
ArmTransactionWithMetricsDefinition atd =
txMetricFactory.
newArmTransactionWithMetricsDefinition(appDef,
"TransWithMetricName", null, metricGroupDef,
null);
cpu = txMetricFactory.newArmMetricCpu(cpuDef);
metricGroup =
txMetricFactory.newArmMetricGroup(metricGroupDef,
new ArmMetric[]{cpu, null, null, null, null, null});

ArmTransactionWithMetrics at =
txMetricFactory.newArmTransactionWithMetrics(app,
atd, metricGroup);
cpu.set(CPUTIME);
```

Figure 7: `ArmTransactionWithMetrics`

4 Java CPU and Memory Metrics

Historically CPU and memory measurements could only be obtaining by using Java Native Interface (JNI). The most common is the Java Virtual Machine Profiler Interface (JVMPPI). While extremely robust, supporting the collection of a wide range of data metrics, the JVMPPI is notoriously slow and its usage was typically reserved for low volume testing [5]. With the introduction of the JDK1.5.0 management library, modeling and performance measurements previously unobtainable from a production environment can now be collected programmatically or by byte code instrumentation [7]. Once CPU or memory or other method specific measurement is obtained it is inserted into the `ARMTransactionWithMetrics` object.

To collect CPU time using the JDK1.5.0, each method of interest will invoke "getCurrentThreadCpuTime" at its start and end. To determine the overall CPU usage simply subtract the ending CPU from the beginning CPU and store the result in a `TransactionWithMetric` "Counter" variable. Figure 8 shows a code snippet that can be used to obtain CPU and memory information.

```
import java.lang.management.*;

ThreadMXBean tmxBean =
ManagementFactory.getThreadMXBean();

MemoryMXBean mmxBean =
ManagementFactory.getMemoryMXBean();
```

```

long tmxCpu = tmxBean.getCurrentThreadCpuTime();
-----
MemoryUsage = mmxBean.getHeapMemoryUsage();
MemoryUsage nhmu =
    mmxBean.getNonHeapMemoryUsage();

```

Figure 8: JVM CPU and Memory

Having described how to ARM enable a Java application, section 5 discusses how the method measurements can be used to construct a Java method model.

5 Building Java Method Models

Building Java application models is not difficult, provided the data is available. With the exception of disk I/O, the ARM based Java instrumentation solves the data acquisition problem by providing a means to gather method measurements such as 1) class.method name, 2) method invocation count, 3) method CPU time, 4) method elapsed time, 5) method visit count, 6) total stack visit count, 7) total

method invocation count, 8) total CPU time, 9) total elapsed time, 10) heap memory used, 11) non heap memory used. Once the application method data are available, analytical models can be constructed using third party or homegrown tools. The homegrown operational analysis and regression tools in Figure 9 and Figure 10 show how the response time and CPU time is affected when running multiple concurrent instances of a single class or method. The operational analysis tool is based on equations 1 through 6, which follow directly from the work of Denning [3], and the non-parametric regression tool shown in Figure 10 implements three kernels. It should be noted that Montgomery in [8] suggests that the bandwidth and not the kernel most affects the regression. Therefore, the ability to adjust the bandwidth provides improved predictability when compared to least square regression. The kernel regression is implemented using equations 7 through 11, which follow from Nadaraya in [9].

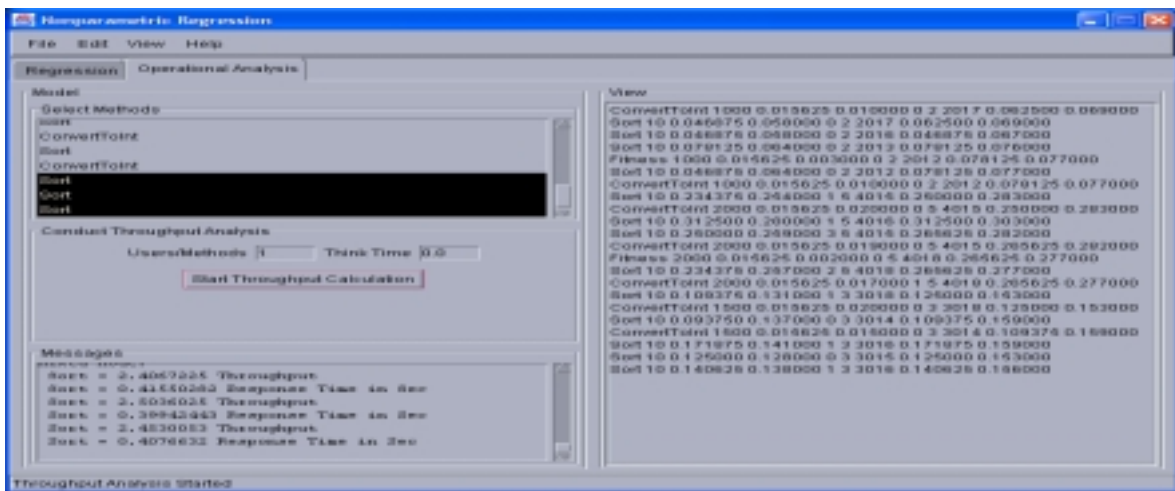


Figure 9: Java Method Operational Analysis Modeling

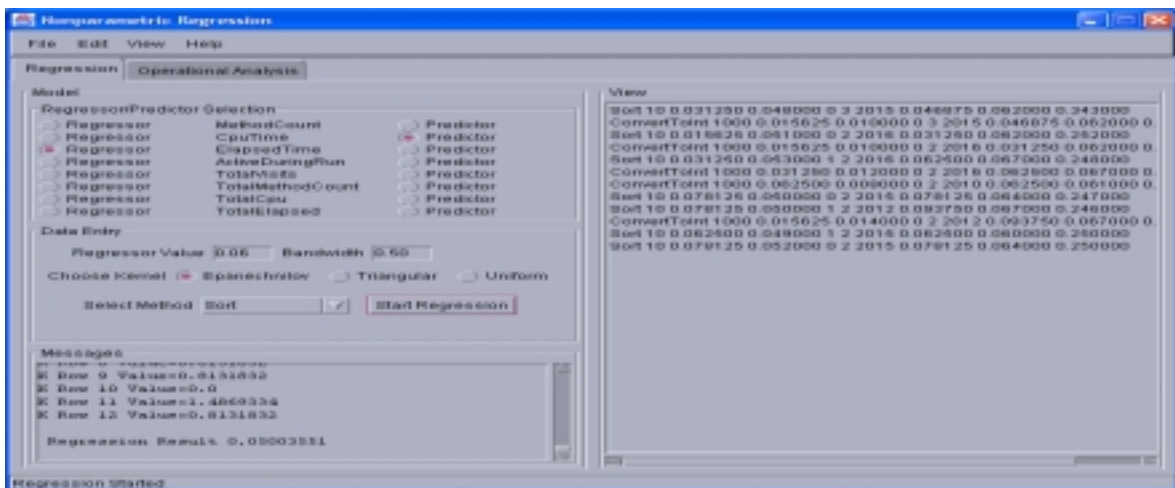


Figure 10: Non-Parametric Analysis of Sort Method Data

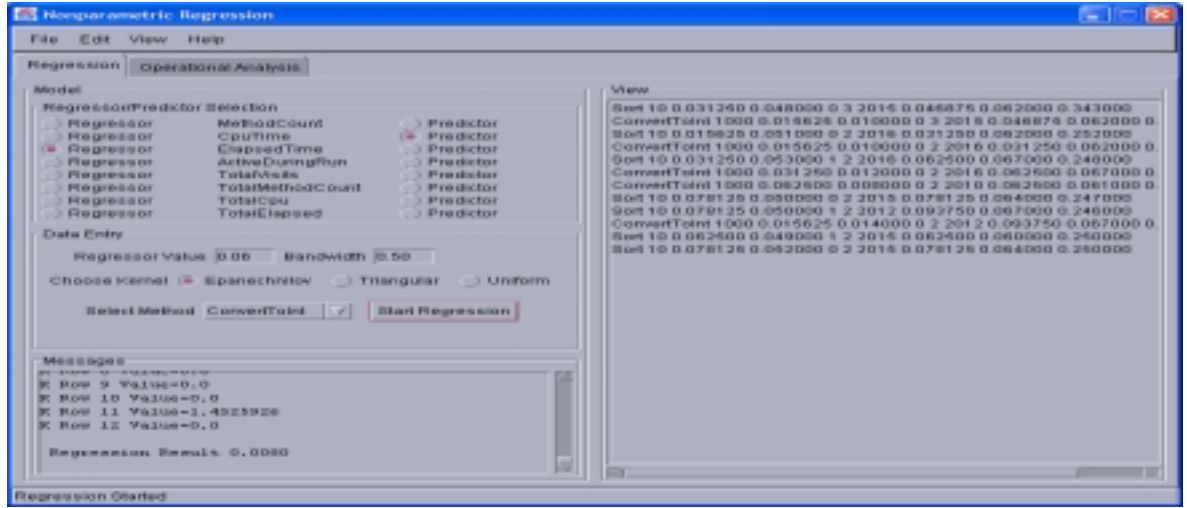


Figure 11: Non-Parametric Analysis of ConvertToInt Method Data

$$Completions = Method Count \quad (1)$$

$$T = Observation Period \quad (2)$$

$$\frac{Utilization = Method Elapsed Time}{T} \quad (3)$$

$$Service = \frac{Method Elapsed Time}{Total Visits} \quad (4)$$

$$X = \frac{Completions (Method Busy Time)}{t} \quad (5)$$

$$ET = \frac{Users / Method Busy Time}{x} - TT \quad (6)$$

$$M_h(X) = \frac{h^{-1} \sum_{i=1}^n K_h(U) Y_i}{h^{-1} \sum_{i=1}^n K_h(U)} \quad (7)$$

$$u = \frac{(X - X_i)}{h} \quad (8)$$

$$K_{Epanechnikov}(u) = 0.75 * h^{-1} * (1 - (u)^2) \text{ for } |u| \leq 1, \text{ else } 0 \quad (9)$$

$$K_{Triangular}(u) = h^{-1} * (1 - (u)^3)^3 \text{ for } |u| \leq 1, \text{ else } 0 \quad (10)$$

$$K_{Uniform}(u) = h^{-1} * e^{-u^2} \text{ for } |u| \leq 1, \text{ else } 0 \quad (11)$$

$$K_{Uniform}(u) = 0.0$$

5.1 Method Models

The operational analysis model in Figure 9, shows an operational analysis model. When running a single class the Sort method required approximately 0.128-0.141 seconds, whereas running three classes concurrently the Sort required 0.399-0.415 seconds, or roughly a 300 percent increase.

Next, to illustrate the usage of the nonparametric model the Sort method shown in Figure 10 does not appear to be bottleneck. However, by running the nonparametric model using a regressor value of 0.06 CPU seconds the model shows that the Sort consumes roughly 0.05 elapsed time seconds. Intuitively this result appears reasonable. Nevertheless, Figure 11 shows that the only other active method ConvertToInt requires only 0.008 seconds. Therefore, nearly 86 percent of the total elapsed time is directly attributable to the Sort. Reducing the overhead of the Sort is clearly desirable objective.

As another example, performance data from Jain [10] was used to compare the least squares regression analysis using Excel to the Non-Parametric model described in this paper. Modeling CPU and I/O data from several samples, the Excel prediction was 10.44 Cpu seconds. In comparison the nonparametric regression using a bandwidth of 0.8 and 0.1 produced a prediction of

9.217 and 10.0 seconds, respectively. The measured Cpu was 9 seconds.

The modeling software discussed in section 5 will be made available by emailing the author at Finwe@yahoo.com.

6 Future Work

Future work should focus on developing an approach to dynamically ARM instrument Java byte code. One approach being considered solicits the “JVMPI_EVENT_CLASS_LOAD_HOOK” event. This event requests the JVM to dispatch a raw Java class to a JVMPI agent whenever a Java class is loaded [11]. Once the JVMPI agent has received this event, it has full access to the class’s methods via byte code [12].

7 Commentary and Conclusions

Due to the large number of methods embedded in any Java application method data provided using the ARMTransactionWithMetrics object is collected using a sampling technique that is activated during periods of interest. The Java ARMTransaction object, which supports transaction correlation, elapsed time measurements, application management, is always enabled.

Clearly, obtaining operational quantities (method measurements) is a difficult aspect of modeling Java applications. However, once a Java application is ARM enabled the operational quantities i.e.; method measurements can be gathered throughout the software development life cycle and used to develop incremental application models that can:

- Identify poorly performing methods or hotspots
 - during development
 - during performance testing
 - during production
- Identify frequently executed user flows
- Better understand hardware required to run the application
- Delay hardware procurement

The methodology presented in this paper will help software engineers easily identify application hotspots and quickly correct production performance problems.

8 References

- [1] J. Buzen, *Analysis of system bottlenecks a using queueing network model*. Paper presented at the Proceeding of the ACM SIGOPS Workshop on System Performance Evaluation (1971).
- [2] F. Baskett, M. Chandy, R. Muntz, and F. Palacios, Open, closed, and mixed network models of queues with different classes of customers. *Journal of the ACM*, 22(2), 248-260.
- [3] P. Denning, and J. Buzen, The operational analysis of queueing network models. *Computer Surveys*, 10(3), 225 - 261 (1978).
- [4] The Continuing Silicon Technology Evolution inside the PC Platform [Online]. Available: <http://www.intel.com/update/archive/issue2/feature.htm> (2002).
- [5] The Java virtual machine specification [Online]. Available: <http://java.sun.com/docs/books/vmspec> (1997).
- [6] ARM Issue 4.0, V2 Technical Standard – C041 C Language Binding; C042 Java Language Binding; Available: <http://www.opengroup.org/onlinepubs> [2004, Dec].
- [7] Byte Code Engineering Language. *Jakarta* [Online]. Available: <http://Jakarta.apache.org/bcel>.
- [8] D. Montgomery, E. Peck, & G. Vining (2001). *Introduction to Linear Regression Analysis*. New York: John Wiley.
- [9] E. A. Nadaraya (1964), On estimating regression. *Theory of Applied Probability*, 9,141-142.
- [10] R. Jain, (1991). *The Art of Computer Performance Analysis*. p225, New York: John Wiley & sons.
- [11] Java Virtual Machine Profiler Interface (JVMPI) [Online]. Available: <http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.htm> (1999).
- [12] C. De Pasquale (2004) Collecting Java Performance Measurements using Byte Code Engineering. CMG2004 International Conference on Computer Measurement.