A Transaction Definition Language for Java Application Response Measurement

J.D. Turner[†], D.P. Spooner[†], J. Cao[†], S.A. Jarvis[†], D.N. Dillenberger[‡], G.R. Nudd[†]
Dept. Computer Science, University of Warwick, Coventry, UK[†]
IBM TJ Watson Research Center, New York, USA[‡]
Email: {jdt,dps,junwei,saj,grn}@dcs.warwick.ac.uk[†] engd@us.ibm.com[‡]

Abstract

This paper describes a technique for automatically ARMing Java applications in accordance with the ARM 3.0 standard for Java (submitted to the Open Group February 2001). An XML-based Transaction Definition Language is defined, which allows a developer to semantically define performance critical components of Java applications. An application is instrumented with ARM method calls through a bytecode transformer prior to execution, providing ARM compliance while removing the necessity to modify (or even possess) the original Java source code.

Keywords: *Java, ARM, Transaction Definition Language, Performance*

1 Introduction

1.1 High Performance Computing

Ever since computing resources have been used to allow scientific and engineering problems to be addressed more efficiently, application developers have spent a great deal of time developing the infrastructures necessary to achieve the greatest performance possible. In the early days, the methodology and philosophy behind such high performance computing was to create massively parallel supercomputers, resulting in expensive, high maintenance systems with hundreds of processors and gigabytes of local storage. The software for such systems often included highly optimized applications written specifically for the architecture upon which they were to run.

It may be necessary for these computationally-intensive applications to be executed on another architecture other than that for which they were specifically written. Where a choice of architectures is available, it is often that which results in the greatest performance which is preferable, although this choice is not necessarily an easy one to make.

Over the last ten years, the High Performance Systems Group (HPSG) at Warwick have developed a wealth of tools allowing application developers to predict the performance of such a class of applications. The main contribution of this research, the Performance Analysis and Characterization Environment (PACE) [NUDD00, CAO00], allows a parallel application to be characterized and then mapped to a variety of architectures; the performance estimates which result allow a user (or scheduling system) to predict the optimum execution behavior over the available hardware.

PACE operates by constructing a theoretical time-line for a given application, based on the timings of individual machine code instructions and their organization as subtasks described within parallel templates. The resulting model is parameterized so that factors such as input data size, number of processors and available memory can be adjusted. Observations can then be made as to the predicted effect on the behavior of the application over a number of different architectures and under a number of performance-critical situations.

PACE adopts a layered approach for application characterization (see Figure 1, part (i)). PACE associates each layer with a set of specified interfaces, allowing objects of the same class to be exchanged without affecting the overall structure of the model. The sequential tasks, the parallelization strategy and the hardware elements of the given application are described using a modeling language known as CHIP^{3}S [PAPA95]. This is subsequently compiled and linked to form a single application object which is later evaluated to obtain performance estimates.

A distributed resource scheduler (named TITAN), has also been developed for the allocation of applications on a static set of known hardware resources. TITAN uses PACE performance models of scheduled applications to predict their execution time and behavior and therefore achieve greater scheduling efficiency. A genetic algorithm is employed to allow the choices involved in task allocation to 'evolve' over time, such

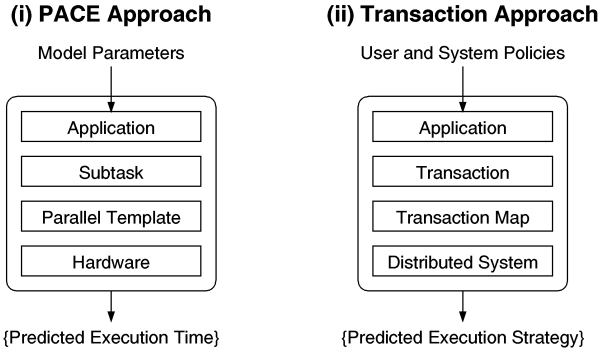

Figure 1: (i) The original PACE layered approach. (ii) The transaction-based approach, used for rapid application characterization.

that when the application is eventually scheduled, the most suitable target architecture is chosen.

1.2 Computational GRIDs

More recently the methodology behind high performance computing has changed. With the vast performance increase of the desktop computer, larger network bandwidth, and the decrease in price for such components, distributed environments have become the favored infrastructure for high performance applications. This interest in geographically dispersed networks such as GRIDs [FOST98, LEIN99] has stimulated the demand for high performance resource allocation services with the ability to adapt to the continuous variations in user demands and resource availability.

Building on the experience with using PACE for the prediction and scheduling of massively parallel scientific applications, a new scheduling environment [SPOO01] is being developed at Warwick to manage more data-intensive business applications running on these GRID-type architectures. Built on top of the Globus infrastructure [FOST01], applications are scheduled according to a wider set of metrics, including communication requirements, quality of service, and according to resource restrictions including software license domains. Resource discovery and advertisement is achieved using an in-house agent system [CAO01, CAO02], and an application's performance is predicted prior to its execution to increase the effectiveness of scheduling software.

PACE is well-suited to the prediction of computationally-intensive, scientific applications, however it is less well-suited to many GRID-type applications; many of the problems derive from the level at which the prediction data is gathered. It has been possible to abstract away from the more machine level details involved in performance prediction by defining the lower level atomic units of work as 'transactions' (see Figure 1, part (ii)). Applications consist of a number of transactions which are related to each other through a 'transaction map'. This approach to application characterization allows a broader class of problems to be addressed, including data- and input-driven business applications.

A system for performance prediction and scheduling of such jobs on GRID architectures has been developed (see Figure 2). On submission, the user presents an application stub to the scheduling environment that describes static performance information about the application. This information contains a transaction map and scheduling metrics, such as the priority of the application, the quality of service required and particular service policies, etc.. Using this information, the current resource load is collected from the environment and compared with the predicted resources deemed necessary for the application to run at its most efficient. Once an optimum resource has been found for a particular application, the application is scheduled for execution on the target architecture.

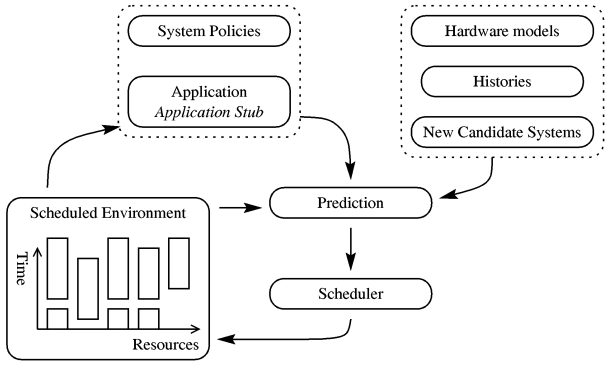

Figure 2: A metric-based scheduling environment incorporating prediction of applications, resource allocation, system policies, performance histories, and quality of service. Feedback of information during the application's execution aids the prediction and scheduling of similar applications in the future.

1.3 Application Response Measurement

Using the Application Response Measurement (ARM) standard [JOHN00, ARM01], applications are split into transactions corresponding to performance-critical components. The ARM standard usefully imposes a unified measure over the performance statistics recorded, these are calculated at runtime by instrumenting applications with ARM procedure calls. The use of application stubs and performance histories allows information regarding performance-critical components of applications to be bootstrapped and then 'tuned' over time.

In order to bridge the gap between service policies and performance-critical ARM measurements, a Transaction Definition Language (TDL) has been defined. This provides an

XML-based descriptor language in which developers can define the key performance-critical transactions in their applications, and then relate these components back to system service policies. The implementation of the TDL automatically ARMs the applications according to the TDL descriptors. Automating such a process has a number of advantages, including that of not needing to access to the source code, and not requiring the need to re-compile or re-link the application during the ARMing process.

The TDL description for each application forms part of the service policy document, which itself is a constituent part of the application stub shown in Figure 2. An ARM consumer implementation has been implemented for use with the scheduling environment, which stores the information in a local LDAP server [HOWE97].

The remaining sections of this paper describe:

- An ARM compliant Transaction Definition Language that allows a developer to semantically define performance critical components of Java applications (see Sections 2 & 3).
- An automated method for the manipulation of Java object code using a Java bytecode transformer, used in the implementation of the TDL (see Section 3).
- The demonstration of the feasibility of this approach on a number of Java applications (see Section 4).

2 Application Response Measurement

With the increasingly complex computer infrastructures present within the industry today, it has become more difficult to analyze the performance of modern applications. Such applications involve separate programs distributed across multiple systems, processes and threads. Each one of these transactions may communicate with a number of data storage devices, application servers or web services.

Analyzing the performance of such a distributed architecture can be problematic and application administrators are faced with a number of difficult questions:

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time experienced by the end user?
- Which sub-transactions are taking too long?
- Where are the bottlenecks?
- How many of which transactions are being used?
- How can the application and environment be tuned to be more robust and to perform better?

The Application Response Measurement (ARM) standard allows these questions to be answered. ARM allows the application developer to define individual units of work within an application as 'transactions'. The performance of these transactions are then measured during the application's execution; analyzing the results allows some of the above questions to be answered. Which transactions are monitored is up to the developer, although they are normally the areas of an application that are considered performance critical.

The standard is employed by ARMing an application; ie. by inserting ARM interface library calls within an application (see Figure 3). These calls define where a transaction begins and ends, and the performance of the application between those points is measured and reported through the consumer implementation of this interface. Via this interface, ARMed applications can communicate with a number of consumer implementations without any change to the source instrumentation. This allows a variety of information to be retrieved according to the implementation used.

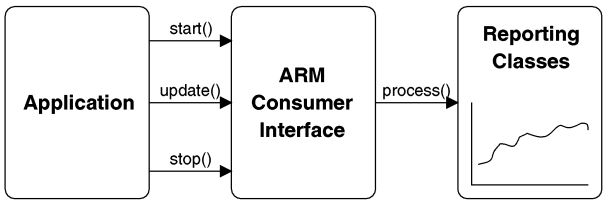

Figure 3: A schematic of the communication between an application, an ARM consumer interface and the reporting classes.

2.1 Java 3.0 Binding

In February of this year, a Java binding version of the ARM specification was proposed to the Open Group [OPNG]; this specification introduced a number of new features not found in previous versions of the standard. The specification defines a set of Java interfaces, allowing the methods defined within each interface to be invoked by a Java application, and a Java consumer implementation to be developed.

One of the most notable features of these interfaces is the 'ArmTransaction'. This interface allows the application to define where transactions occur, which is done by invoking 'start' and 'stop' methods at the beginning and end of each transaction. According to the specification, each transaction records a mandatory set of information, including how long it took for the transaction to complete (the response time), a time-stamp of when that transaction finished (the stop time), and whether the transaction was successful or not (the status). Other optional

information can be associated with all transactions, either by associating a User and Transaction Definition class, or by utilizing a number of metrics which may be passed to the transaction when instantiated. Figure 4 presents the 'ArmTransaction' data model in more detail.

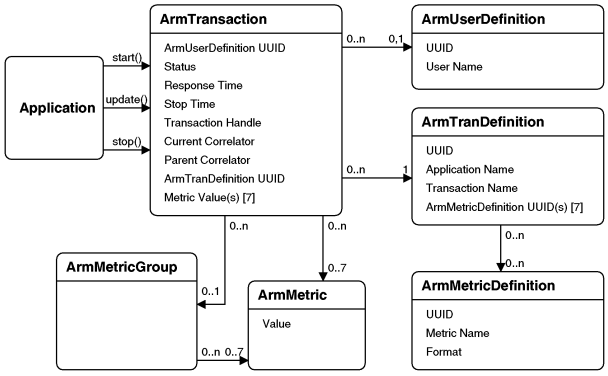

Figure 4: The ARM 3.0 Transaction Data Model.

It is possible to create an understanding of how transactions relate to each other within an application with the use of correlators. Current and parent correlators can be associated with each transaction. Many transactions can have the same parent and therefore a transaction map can be created from the relation of correlators reported by the ARM implementation during execution. It is this transaction map which is created and used within the scheduling environment to aid in the distribution of applications over GRIDs architectures.

2.2 An Example

The following example describes how an application communicates with an ARM consumer interface, and how descriptive information concerning the performance-critical transactions defined within an application is processed.

In a web server application, a transaction is defined in such a way that every page request that occurs during the web server's lifecycle is monitored according to the consumer implementation that is used. The developer instruments the application source code with method calls to the ARM consumer interface as shown in Figure 5.

The developer associates the optional transaction and user definition objects with the ARM transaction. Association of such information within ARM is achieved with the use of Universally Unique Identifier (UUID) 16-byte arrays, which uniquely identify each Java object used. Using UUIDs also allows the descriptive information to be kept separate from the actual transaction measurements until the information is reported at some time after the transaction completes its execution.

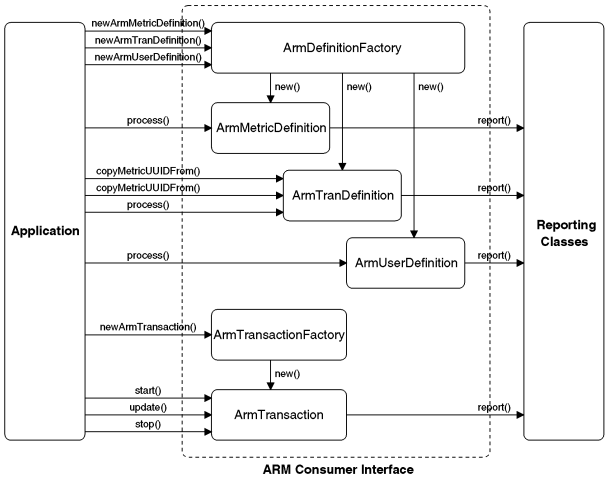

Figure 5: All ARM transactions have (optional) definition information associated with them via ArmMetricDefinition, ArmTranDefinition, and ArmUserDefinition objects. Each of these objects are populated with information describing the transaction; when the application invokes a process call, the consumer interface reports the data to the reporting classes. The application then creates a new instantiation of an ArmTransaction object, and defines the location of the transaction by invoking the start and stop transaction calls.

The three definition objects are each associated a UUID by the application. When the application invokes a process call to the definition objects used, report information is then passed by the consumer interface and handled by the reporting classes. As extra metric information is held within the metric definition object, whose UUIDs are referenced by the transaction definition, the application has to associate those metrics which are connected with the transaction definition before its information can be processed.

Once the transaction definition information has been completed and processed, a new transaction is created using the transaction factory, which associates the definition UUIDs with the transaction. A 'start' is sent to the instantiation of the transaction when a transaction begins and a 'stop' sent when the transaction ends. Ending the transaction then automatically reports the measurements made during the transaction to the reporting classes.

The reporting classes generate a number of tables, (see Tables 1, 2, 3, and 4): these show the transaction measurements made by the consumer implementation, and also the transaction, user and metric definitions processed by the application. Mapping the transaction measurements back to the definition objects by way of the UUIDs provides detailed information as to the transaction's name, the application name, the user, and the name and type of the measurement metrics used during the monitoring process.

TranUUID	UserUUID	Status	Response Time	Metric 1
64E14	72536	GOOD	5.638	632

Table 1: Transaction Measurements.

TranUUID	Appl Name	Transaction Name	MetricUUID 1
64E14	Web Server	Page Request	B6115

Table 2: Transaction Definitions.

UserUUID	User Name
72536	Administrator

Table 3: User Definitions.

MetricUUID	Name	Format
B6115	Server Load	Guage32

Table 4: Metric Definitions.

2.3 Asynchronous Data Reporting

Version 3.0 of the ARM specification allows an application to measure performance data and report the data asynchronously via the 'ArmTranReport' interface. Instead of the ARM implementation measuring the performance information of transactions, the application measures the information and populates an 'ArmTranReport'. When complete, the application sends a 'process' call to the interface which then reports the data at that time.

For most distributed applications it is possible to use either method of measuring the performance information. However, in situations where it is not feasible to place the 'start' and 'stop' calls at the exact location at which the transaction is defined, it is necessary for an application to populate the 'ArmTranReport' instance in order for accurate results to be reported.

3 Transaction Definition Language

ARM provides an efficient and convenient way of measuring the performance of distributed applications during their execution. The ARM standard therefore provides an effective mechanism for the performance monitoring of applications within the PACE-based scheduling environment described in Section 1. In particular, the performance information which the ARMing of applications provides can be used to increase the accuracy of performance prediction and provide the basis for efficient task allocation by the scheduler.

The ARMing of an application can however prove difficult. It requires that an application developer define the performance-critical components of application and therefore where transactions will take place. The application need then be instrumented so that ARM calls to the consumer interface are made appropriately. We will show that it is possible to reduce the overhead involved in ARMing applications and in so doing provide the application developer with a high-level mechanism by which application service level agreements can be modeled.

Another potential disadvantage to the current scheme is the necessity for the developer to possess and be able to modify the application's source code. With modern business environments and the complexity of license agreements this may not be possible. Even worse, the source code may not be available. It is quite possible that a remote GRID location submits just an application object code into the distributed GRID infrastructure. In the past, ARMing an application without source code has only been made possible through the use of management monitoring techniques [HAWO97].

An effective solution to these problems is for a user to provide as part of the service policy a description of where performance-critical components of an application are likely to exist. To achieve this, an XML-based Transaction Definition Language (TDL) has been developed. Transactions can be defined at a number of abstract levels in the application, whether it be at a low-level through method calls and areas delimited by source code line numbering, or at a high level in terms of operational units such as database accesses or remote procedure calls. An application is then ARMed accordingly, invoking transaction method calls to a pre-defined consumer interface, which reports the data as appropriate.

The TDL provides a useful way of bridging the gap between a GRID-type scheduling environment and the collection of application performance-critical measurements. The TDL also provides a useful way of interfacing Java applications with the ARM specification. One use of ARM is to detect where bottlenecks appear within an application, and the TDL could be used to ARM all methods in all classes within an application with the aim of locating such bottlenecks. Once found, the abstraction mechanism of the TDL allows more specific portions of the application to be automatically ARMed, until the exact area in the application where performance is lacking is targeted and can be changed.

3.1 Specification: TDL DTD

The DTD in which the TDL syntactic rules are defined can be found in Figure 6. The root element of all TDL files is a `tdl` element, this contains one attribute that defines the Java application to be ARMed. Within the scheduling environment all Java applications are represented as a single Java .jar file. These .jar files contain all the classes necessary for the application to execute, excluding system classes, which are contained within the Java distribution being used[1].

[1] It is possible to ARM these classes with the TDL as the system classes are usually held within 'rt.jar'.

```
<!-- Transaction Definition Language DTD (v1.0) -->

<!ELEMENT tdl (transaction+)>
<!ATTLIST tdl jarfile CDATA #REQUIRED>

<!ELEMENT transaction (location, line_number?, metric*)>
<!ATTLIST transaction type (method_source | method_call |
                            line_number) #REQUIRED
                      fail_on_exception (yes | no) "yes"
                      user_name CDATA #IMPLIED>

<!ELEMENT location EMPTY>
<!ATTLIST location class CDATA #REQUIRED
                   method CDATA #REQUIRED
                   call CDATA #IMPLIED>

<!ELEMENT line_number EMPTY>
<!ATTLIST line_number begin CDATA #REQUIRED
                      end CDATA #REQUIRED>

<!ELEMENT metric EMPTY>
<!ATTLIST metric type (Counter32 | Counter64 | CounterFloat32 |
                       Guage32 | Guage64 | GuageFloat32 |
                       NumericId32 | NumericId64 |
                       String8 | String32) #REQUIRED
                 name CDATA #REQUIRED
                 value CDATA #REQUIRED>
```

Figure 6: Transaction Definition Language DTD.

Each `tdl` element contains one or more transaction elements which define where transactions occur within an application. The current TDL specification (version 1.0) allows for three types of transaction:

- *Method Source:* A method source transaction states that the transaction starts at the beginning of that method and ends when the method returns, as depicted in Figure 7. For this transaction type to be permitted, the class and method described by the location element associated with this transaction must be found within the jar file being ARMed. If the method being ARMed calls itself anywhere throughout the method, ie. the method is recursive, then the method is not ARMed, because the same transaction instantiation can not be in use more than once according to the ARM specification. In this situation, the method call type must be used.

- *Method Call:* A method call transaction states that the transaction starts where that method is called, as depicted in Figure 8. If this type of transaction is used, then the optional call attribute within the location element must be used. This transaction type has many advantages over the method source type, including the ability to report the performance of recursive methods and also methods not included within the jar file specified by the tdl element attribute. However, the overhead incurred by the Java Virtual Machine (JVM) to invoke the method is included within the transaction's response time.

- *Line Number:* A line number transaction states that the transaction begins at a specific line number within a method and ends at a line number greater than the first, as shown in Figure 9. If this type is used then both the location and line number elements are required within that transaction element. If the developer has access to the source code and would like to define a transaction within two specific line numbers of the source, then this type will allow this to be possible. However, this type is only usable if the line numbers have been included within the class file by the Java compiler.

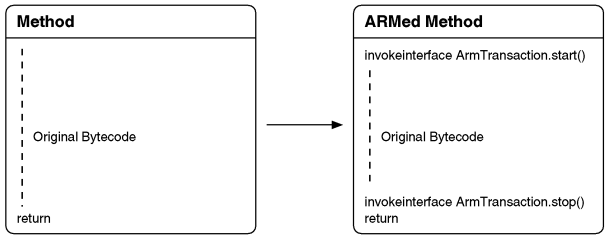

Figure 7: *Method Source* Transaction Type.

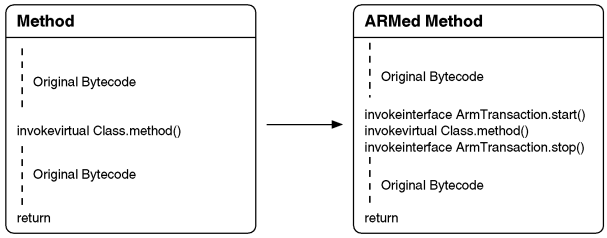

Figure 8: *Method Call* Transaction Type.

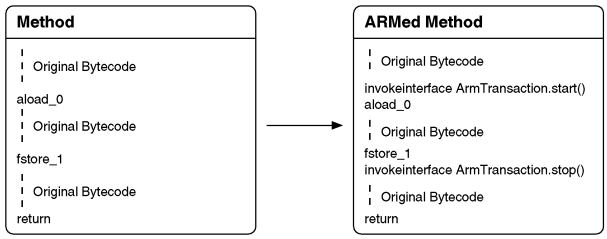

Figure 9: *Line Number* Transaction Type.

Each transaction can include 'fail on exception' and 'user name' attributes. Detecting whether a transaction has failed or not is done by observing whether an exception is thrown by the area of code in which the transaction is defined. If an exception is thrown, then by default the transaction is reported

as failing. The 'fail on exception' attribute is used when transactions, which despite throwing exceptions, are reported as succeeding.

The 'user name' attribute allows the optional ArmUserDefinition object to be associated with the corresponding named transaction. This attribute is reported to the implementation together with the ArmTranDefinition prior to starting the transaction. ArmUserDefinition objects are not created by default, but if the developer requires a user name to be reported then this attribute should be used.

Up to seven optional metrics can also be associated with each transaction, with their name, format and start value defined within a metric element. If used, an ArmMetricDefinition object is created and associated with the ArmTranDefinition before being reported.

There is also a keyword ANY which can be substituted into all attributes of the location class. If this keyword is used, all classes or methods that satisfy the location requirement are defined as transactions and ARMed accordingly.

3.2 Implementation: Bytecode Transformer

The implementation of the TDL specification described above is achieved with the use of an automated instrumentation technique [LEE96, DAHM99, COHE98]. Automated instrumentation involves the manipulation of either source or object code in a pre-defined fashion, such that the execution of the instrumented application fulfills a number of requirements. Such a technique can be used to instrument an application so that it directly interfaces with an ARM consumer implementation.

The automated instrumentation of Java applications is made possible through the use of a Java bytecode transformer. Each Java class consists of a number of fields, methods and attributes and a constant pool (a lookup table of strings and method references etc., which is used within the class to improve efficiency). The bytecode transformer reads the data encoded within a class file and parses the information into a number of Java objects, each of which represents one of these specific elements of the class. These objects can then be processed, modified as required, and then written back to a file, resulting in a Java class whose execution is different from the original.

By using the bytecode transformer to parse objects within a Java application, it is possible to match transactions defined within the TDL to the location within the application. The application can then be transformed so that ARM method invocations are made at the appropriate locations within the bytecode. Once instrumented, the application is executed; the execution behavior of the application being exactly that of the original other than the reporting of the performance statistics of the defined transactions via the ARM consumer interface.

3.2.1 The Automated ARMing Process

At the start of the automated ARMing process the TDL file is processed and checked for errors. This includes checking for errors in the TDL DTD and any other errors that may be present even though the TDL file itself is syntaxically correct. The TDL file is parsed using a standard XML parser [XML].

When the application is processed, the manifest of the jar file is decoded so that the entry level main class of the jar file is located. Using the transformer, a new static field array of type ArmTransaction is inserted. The static main method (the entry method to the application) is then instrumented in such a way that the array is initialized at the beginning of the method, with a size equal to the number of transactions defined within the TDL file. This creates the pool of ArmTransaction objects necessary for all the ARM transactions which may be added throughout the execution of the application[2].

After the initialization is achieved, all the transactions that are defined within the TDL file are processed and mapped to specific methods within the application. If a current mapping is achieved, the method is instrumented with the transaction initialization code and start and stop calls surrounding the relevant bytecode are inserted.

The initialization code is dependent on the definition of the transactions within the TDL file. If a user name and metrics are associated with a transaction, then both ArmUserDefinition and ArmMetricDefinition objects are created and a process call is sent to the ARM implementation. An ArmTranDefinition object is associated with every transaction: the application name set to the path and file name of the jar file and the transaction name set to a description of the location of the transaction.

When instrumenting a method with ARM calls, the ARM package is fixed (`comp.opengroup.arm3.application`) as stated within the ARM specification. With this assumption, the only doubt as to the location of a class is with the four ARM factory classes. This information is normally retrieved from the JVM system properties, which an administrator maintains. However, the philosophy behind the TDL is that it bridges the gap between GRID scheduling middleware and the gathering of performance metrics. To solve this, the ARMing process requires that an ArmFactory class is created in the same package as the other ARM interfaces, which points the four ARM factories to the ARM consumer implementation being used. An example of such a file can be seen in Figure 10.

When instrumenting Java bytecode care has to be taken so that the execution of the original bytecode is not affected. The execution of bytecode within the JVM is based on an operand stack and a number of local variables, and the majority of opcodes defined by the JVM specification [JVM99] alter the current state of the memory owned by the method. It is beneficial

[2] It was decided not to initialize the field array within the class init method due to the extra overhead and larger class file which would result.

```
package org.opengroup.arm3.application;
import org.opengroup.arm3.implementation.*

public abstract class ArmFactory
{

  public static ArmDefinitionFactory createArmDefinitionFactory()
  {
    return new ImplDefinitionFactory();
  }

  public static ArmTransactionFactory createArmTransactionFactory()
  {
    return new ImplTransactionFactory();
  }

  public static ArmTranReportFactory createArmTranReportFactory()
  {
    return new ImplTranReportFactory();
  }

  public static ArmMetricFactory createArmMetricFactory()
  {
    return new ImplMetricFactory();
  }

}
```

Figure 10: An example ArmFactory class.

that instrumented code does not alter the data held within the method's stack and local variables. To achieve this, a number of local variables are associated with the method and used by the instrumented code. This means that the original local variables remain unaffected, and any objects left on the stack are popped prior to re-entering the original bytecode.

Once a match is found for all transactions defined within the TDL file, and each transaction is instrumented accordingly, the jar file is output, resulting in an application which is ARMed as the TDL file describes.

3.3 Using the TDL

The following three examples show how a developer would use the TDL to ARM specific parts of an application. They range in complexity, so as to present the range of ARM functionality that is available to a developer using the TDL.

3.3.1 TDL Example 1

Figure 11 shows a simple TDL XML file which defines one transaction of type method source, held within `example1.jar`, which surrounds the source of the method `example1method` contained within the class `example1class`. The fail on exception attribute is not defined and so is preset by default. No user name or metrics are associated with the transaction.

```
<?xml version="1.0"?>
<!DOCTYPE tdl SYSTEM "tdl.dtd">

<tdl jarfile="example1.jar">

  <transaction type="method_source">
    <location class="example1class" method="example1method"/>
  </transaction>

</tdl>
```

Figure 11: TDL Example 1 XML file.

3.3.2 TDL Example 2

Figure 12 shows a more complex TDL XML file defining two transactions of type method source and line number for the jar file `example2.jar`. The first transaction has the user name `admin` associated with it, and defaults to failing if the method `example2method` contained within the `example2class` class throws an exception. Two metrics are also associated with the transaction.

The second transaction is of type line number, and places a transaction around the bytecode compiled between lines 32 and 208 of the original source of the method `example2method2` in class `example2class2`. The transaction does not fail if an exception is thrown during the execution of the transaction, and has a user name of `root` associated with it, along with one metric of type `String8`.

```
<?xml version="1.0"?>
<!DOCTYPE tdl SYSTEM "tdl.dtd">

<tdl jarfile="example2.jar">

  <transaction type="method_source" user_name="admin">
    <location class="example2class" method="example2method"/>
    <metric type="GuageFloat32" name="example2metric1" value="4.1"/>
    <metric type="String8" name="example2metric2" value="Load"/>
  </transaction>

  <transaction type="line_number" fail_on_exception="no"
                                  user_name="root">
    <location class="example2class2" method="example2method2"/>
    <line_number begin="32" end="208"/>
    <metric type="String8" name="example2metric3" value="MemUsage"/>
  </transaction>

</tdl>
```

Figure 12: TDL Example 2 XML file.

3.3.3 Using the TDL within a Service Policy

Figure 13 shows an example service policy document which includes a reference to an external TDL description along with other quality of service requirements. Notable features include a number of service classes that describe the policy requirements of the application for which this service policy is associated. These include a priority level, a constraint on the

response time, and a reference to the TDL file defining the performance-critical transactions within the application.

```
<AServicePolicy PolicyName="PACE Scheduling Application Submission"
                PolicyVersion="Version 1.0">
    <Workloads>
       <Workload WorkloadName="SCIMARK"
                 WorkloadDescription="SCIMark Java Benchmark Suite"/>
       <Workload WorkloadName="ADMIN"
                 WorkloadDescription="Environment Maintenance"/>
       <Workload WorkloadName="SYSTEM"
                 WorkloadDescription="System work"/>
       <Workload WorkloadName="OTHER"
                 WorkloadDescription="Unimportant work"/>
    </Workloads>
    <ReportClasses>
       <AReportClass ReportClassName="ARMREPORT"
                     ReportClassDescription="ARM Reporting Classes"/>
    </ReportClasses>
    <ServiceClasses>
       <ServiceClass ServiceClassName="SCN01"
                     ServiceClassDescription="SCIMark Small Dataset"
                     WorkloadName="SCIMARK" Priority="2">
          <Application JarFile="jnt.jar" Arguments=""
                       TDLFile="jnt.tdl"/>
          <Goal GoalType="Average" ResponseTime="30000"/>
       </ServiceClass>
       <ServiceClass ServiceClassName="SCN02"
                     ServiceClassDescription="SCIMark Large Dataset"
                     WorkloadName="SCIMARK" Priority="3">
          <Application JarFile="jnt.jar"
                       Arguments="-large" TDLFile="jnt.tdl"/>
          <Goal GoalType="Discretionary" ResponseTime="75000"/>
       </ServiceClass>
       <ServiceClass ServiceClassName="SYSTEM"
                     ServiceClassDescription="System"
                     WorkloadName="System" Priority="0">
          <Goal GoalType="System"/>
       </ServiceClass>
    </ServiceClasses>
    <ClassificationRules>
       <Subsystem Platform="Linux" InstrumentationType="ARM"
                  ProcessOrSubsystemType="GRID"
                  DefaultServiceClassName="SCN1"
                  DefaultReportClassName="ARMREPORT">
          <ClassificationRule TransactionClass="Gold"
                              ServiceClass="SCN01" ReportClass="">
          <ClassificationRule TransactionClass="Silver"
                              ServiceClass="SCN02" ReportClass=""/>
          <ClassificationRule>
             <Filters UUID="" URL="" ObjectName="jnt/scimark2/kernel"
                      MethodName="" TransactionalQOSToken=""
                      ApplicationProfileName="" TransactionClass=""
                      TransactionName="" SubsystemInstance=""
                      NetworkQOSClassName="" SourceIPAddress=""
                      SourcePort="" TargetIPAddress="" TargetPort=""/>
          </ClassificationRule>
       </Subsystem>
    </ClassificationRules>
</AServicePolicy>
```

Figure 13: An example service policy.

4 Case Study: Java Benchmark

To illustrate the use of the TDL in a real world example, the SciMark benchmarking suite [SCI] is selected as an example application. SciMark 2.0 is a composite Java benchmark which measures the performance of five floating-point intensive kernels which are popular within scientific and engineering applications. These include fast fourier transforms, jacobi successive over-relaxation, sparse matrix-multiply, monte carlo integration, and dense LU matrix factorization.

On execution of the benchmark, the individual calculations are performed one after the other, with the results of each kernel and a composite score reported at the end in mega floating point operations per second (Mflops). Each kernel performs its calculation against a random data set, the size of which is determined by the user. Larger data sets can be chosen such that the memory accesses are guaranteed to be out of cache space, so that the performance of the virtual machine's memory allocation can be discussed.

4.1 Transaction Definition

The benchmark application is ARMed so that five transactions surrounding the source of each kernel are measured by the ARM implementation. A simple ARM consumer implementation is used that measures the data necessary to meet the ARM specification. The data reported includes the response time of each transaction. Each transaction is defined as a `method source` type by the TDL, and a number of different user names are associated in order to illustrate the changes that occur within the reporting of the ArmUserDefinition objects. The TDL file is shown in Figure 14.

```
<?xml version="1.0"?>
<!DOCTYPE tdl SYSTEM "tdl.dtd">

<tdl jarfile="jnt.jar">

  <transaction type="method_source" user_name="James">
    <location class="jnt/scimark2/kernel"
              method="measureFFT"/>
  </transaction>

  <transaction type="method_source" user_name="James">
    <location class="jnt/scimark2/kernel"
              method="measureSOR"/>
  </transaction>

  <transaction type="method_source" user_name="Dan">
    <location class="jnt/scimark2/kernel"
              method="measureMonteCarlo"/>
  </transaction>

  <transaction type="method_source" user_name="Steve">
    <location class="jnt/scimark2/kernel"
              method="measureSparseMatmult"/>
  </transaction>

  <transaction type="method_source">
    <location class="jnt/scimark2/kernel"
              method="measureLU"/>
  </transaction>

</tdl>
```

Figure 14: Scimark Benchmark Transaction Definition File.

TranUUID	UserUUID	Status	Response Time
[B@45d741aa	[B@45d441aa	GOOD	5100
[B@4960c1aa	[B@4961c1aa	GOOD	4284
[B@7dac01aa	[B@7dad01aa	GOOD	6038
[B@7d7241aa	[B@7d7341aa	GOOD	7557
[B@605e81aa	NULL	GOOD	5449

TranUUID	Appl Name	Transaction Name
[B@45d741aa	jar/jnt.jar	jnt/scimark2/kernel/measureFFT
[B@4960c1aa	jar/jnt.jar	jnt/scimark2/kernel/measureSOR
[B@7dac01aa	jar/jnt.jar	jnt/scimark2/kernel/measureMonteCarlo
[B@7d7241aa	jar/jnt.jar	jnt/scimark2/kernel/measureSparseMatmult
[B@605e81aa	jar/jnt.jar	jnt/scimark2/kernel/measureLU

UserUUID	User Name
[B@45d441aa	James
[B@4961c1aa	James
[B@7dad01aa	Dan
[B@7d7341aa	Steve

Table 5: Transaction measurements and associated definitions from the Scimark small dataset.

TranUUID	UserUUID	Status	Response Time
[B@5c2141dc	[B@5c2641dc	GOOD	21429
[B@26f801dc	[B@26f901dc	GOOD	7265
[B@26e401dc	[B@26e501dc	GOOD	6048
[B@2083c1dc	[B@2080c1dc	GOOD	6452
[B@587501dc	NULL	GOOD	30706

TranUUID	Appl Name	Transaction Name
[B@5c2141dc	jar/jnt.jar	jnt/scimark2/kernel/measureFFT
[B@26f801dc	jar/jnt.jar	jnt/scimark2/kernel/measureSOR
[B@26e401dc	jar/jnt.jar	jnt/scimark2/kernel/measureMonteCarlo
[B@2083c1dc	jar/jnt.jar	jnt/scimark2/kernel/measureSparseMatmult
[B@587501dc	jar/jnt.jar	jnt/scimark2/kernel/measureLU

UserUUID	User Name
[B@5c2641dc	James
[B@26f901dc	James
[B@26e501dc	Dan
[B@2080c1dc	Steve

Table 6: Transaction measurements and associated definitions from the Scimark large dataset.

4.2 Results

The TDL ARMed Scimark benchmark is executed using both the small and large benchmark datasets[3]. The reported results from the ARM implementation can be seen in Table 5 for the smaller data set, and Table 6 for the larger set. Metric definition data was not reported because no metrics were associated with any of the transactions.

The reported data from the ARM consumer implementation includes the response time of each transaction in milliseconds. Each transaction completed successfully and the transaction and user definition data describes the optional data associated with each transaction. The overhead incurred by ARMing the application over the original benchmarking suite was in the order of 0.1%.

5 Conclusion

This paper describes a new environment for the scheduling of distributed applications running on GRID architectures. Applications are submitted to the environment with an application stub, which includes a service policy describing workload, performance and quality of service requirements. Efficient task allocation is achieved with the use of a resource discovery and advertisement agent hierarchy, and by predicting the performance of an application prior to its execution. This is achieved by comparing performance-critical components of the application with previously executed transactions.

Transactions within an application are semantically defined by the user including a Transaction Definition Language document with the application's service policy at submission. The TDL is processed and the application is automatically instrumented so that performance information can be retrieved during its execution. This is made possible by instrumenting the application with Application Response Measurement method invocations, allowing an ARM consumer implementation to measure the performance of the defined transactions, and reporting this data to a set of reporting classes. Java bytecode is instrumented with the use of a Java bytecode transformer.

The TDL specification provides a method of bridging the gap between the service policies associated with GRID-based applications and lower-level ARM-based performance monitoring. The TDL allows developers to experiment with performance monitoring techniques without the necessity to recompile or even possess the application source code. The performance overhead incurred in automatically ARMing and monitoring applications was measured and found to be in the order of 0.1%.

[3]The benchmark is executed on a Pentium III 450 MHz with 256 Mb RAM, running Redhat Linux 7.1 and kernel 2.4.9, IBM JVM 1.3.7.

6 Acknowledgments

The authors would like to express their gratitude to IBM's T J Watson Research Center, NY. for the contribution towards this research.

The work is sponsored in part by a grant from the NASA AMES Research Center, and administered by USARDSG, contract no. N68171-01-C-9012.

References

[ARM01] The Open Group, "Application Response Measurement (Issue 3.0 - Java Binding)", Open Group Technical Specification, (2001). Available from regions.cmg.org/regions/cmgarmw/index.html

[CAO00] J. Cao, D.J. Kerbyson, E. Papaefstathiou and G.R. Nudd, "Modeling of ASCI High Performance Applications using PACE", 19th IEEE International Performance, Computing and Communication Conference, Phoenix USA. 485-492 (2000).

[CAO01] J. Cao, D.J. Kerbyson, G.R. Nudd, "Peformance Evaluation of an Agent-Based Resource Management Infrastructure for GRID Computing", Proceedings of 1st IEEE/ACM Int. Symposium on Cluster Computing and the Grid, Brisbane Australia. 311-318 (2001).

[CAO02] J. Cao, D.P. Spooner, J.D. Turner, S.A. Jarvis, D.J. Kerbyson, S. Saini and G.R. Nudd, "Agent-based Resource Management for Grid Computing", Invited paper at the 2nd IEEE/ACM Int. Symposium on Cluster Computing and the Grid, Berlin. 21-24 May (2002).

[COHE98] G.A. Cohen, J.S. Chase, D.L. Kaminsky, "Automatic Program Transformation with JOIE", Proceedings of the USENIX Annual Technical Symposium (1998).

[DAHM99] M. Dahm, "Byte Code Engineering", Proceedings of JIT (1999).

[FOST98] I. Foster, C. Kesselman, "The Grid : Blueprint for a New Computing Infrastructure", Morgan Kaufmann. 279–290 (1998).

[FOST01] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", To be published in Intl. J. Supercomputer Applications, (2001).

[HAWO97] M. Haworth, Service Management Using The Application Response Measurement API Without Application Source Code Modification (1997). Available from regions.cmg.org/regions/cmgarmw/shortarm.html

[HOWE97] T. Howes, M. Smith, "LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol", Macmillan Technical Publishing (1997).

[JOHN00] M.W. Johnson, J. Crowe, "Measuring the Performance of ARM 3.0 for Java", Proceedings of CMG2000 International Conference, Orlando USA. (2000).

[JVM99] SUN Microsystems, Java Virtual Machine Specification, 2nd Edition (1999). Available from java.sun.com/docs/books/vmspec/

[LEE96] H.B. Lee, "BIT : Bytecode Instrumenting Tool", Thesis for Masters of Science, University of Washington (1996). Available from www.cs.colorado.edu/~hanlee/BIT/index.html

[LEIN99] W. Leinberger, V. Kumar, "Information Power Grid : The New Frontier in Parallel Computing?", IEEE Concurrency **7(4)**, (1999).

[NUDD00] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, D.V. Wilcox, "PACE - A Toolset for the Performance Prediction of Parallel and Distributed Systems", International Journal of High Performance Computing Applications, Special Issues on Performance Modelling. **14(3)**, 228-251 (2000).

[OPNG] The Open Group. www.opengroup.org

[PAPA95] E. Papaefstathiou, D.J. Kerbyson, G.R. Nudd, T.J. Atherton, "An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems", 8th ISCA Int. Conf on Parallel and Distributed Computing Systems, Florida USA. 527–533 (1995).

[SCI] SciMark v2.0 Java Benchmark Suite. Available from math.nist.gov/scimark2/

[SPOO01] D.P. Spooner, J.D. Turner, J. Cao, S.A. Jarvis, G.R. Nudd, "Application Characterisation using a Lightweight Transaction Model", Proceedings of 17th Annual UK Performance Engineering Workshop (UKPEW '01), Leeds UK. 215-225 (2001).

[XML] XML4J: IBM Java XML Parser with DTD validation. Available from www.alphaworks.ibm.com/tech/xml4j