This presentation covers SDO data binding in the WebSphere® Application Server Feature Pack for Service Component Architecture.

Key features of SDO data binding

- Serialization/deserialization of data between the application and wire
- Scope Management
  - Basic
  - Shared scopes
- Programming model
  - API to access default scope
  - JAX-WS Based

The SDO data binding support includes serialization and deserialization of data between the application and the wire; scope management, which is either basic or shared; and the JAX-WS based programming model.

Each of these features is covered in detail in the next couple of slides.

Serialization/deserialization of data

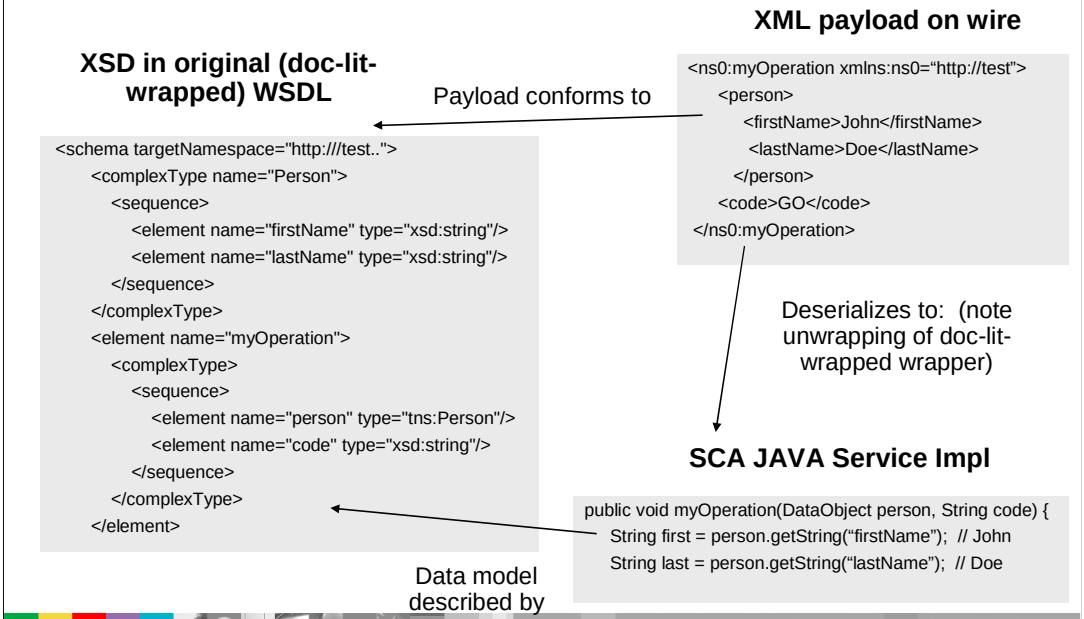
- Used to marshall/unmarshall service interface input/output/fault values to/from the application service client/implementation
- SDO defines a mapping between XML document and an instance of SDO's **commonj.sdo.DataObject**
  - XML document = wire data format
  - DataObject = application data format

An important aspect of SDO is that it is used to marshall and unmarshall service interface input, output, and fault values to and from the application service client or service implementation. In a manner parallel to JAXB, SDO defines a mapping between an XML document and an instance of SDO's **commonj.sdo.DataObject.**

An XML document is used for the wire data format, and a DataObject is the application data format.

This happens in the scope of the application's default HelperContext.

Example – XML document, XSD, SDO Java™



Here is an example of an XML document, XSD, and SDO Java.

Scope management

- In SDO, a scope = an instance of **commonj.sdo.helper.HelperContext**
- SCA defines SDO scopes within boundaries meaningful from SCA-application perspective
- SCA/SDO runtimes together provides SCA applications with a default HelperContext

In SDO, a scope corresponds precisely to an instance of commonj.sdo.helper.HelperContext. As with any scope, this defines visibility boundaries for SDO types. In the Feature Pack for SCA, the SCA layer defines HelperContext on SCA application boundaries, which are meaningful boundaries from an SCA perspective. Together, SCA and SDO define a default HelperContext for a given application and enable the SCA application to access this programmatically.

Scope mgmt – based on deployable composite and contribution

- SCA creates a unique HelperContext (scope) for each deployable composite
- Each deployable composite is contributed to the SCA domain by a single contribution (JAR)
  - All schema definitions in WSDL/XSD files packaged within the same contribution JAR is added (on-demand) to a given deployable composite's HelperContext
- No programmatic API calls like the SDO XSDHelper.define() API

The service component architecture creates a unique HelperContext, or scope, for each deployable composite. Each deployable composite is contributed to the SCA domain by a single JAR. All schema definitions in WSDL/XSD files packaged within the same contribution JAR are added, on-demand, to a given deployable composite's HelperContext. This happens without any programmatic API calls.

Scope mgmt - registration of schema definitions

- SCA registers schema definitions from your SCA application's WSDL/XSDs to your application HelperContext
- Registered definitions is used by:
  - The runtime, to establish the SDO Type of DataObject instantiated by deserializing service request/response data from the wire
  - The application, to create DataObject of application-specific Type

The SCA run time provides mechanisms to register schema definitions from your SCA application (for example, in WSDL/XSD files packaged with your application) to your SCA-meaningful scopes, such as your application default HelperContext.

Schema registration is important because when using SDO to interact with XML data, the exact details of your application's SDO programming model can be affected. They can be affected by whether the relevant SDO scope recognizes the schema definitions in the original XML document (such as the wire format data coming in over the binding invocation).

For example, the commonj.sdo.DataObject method Object get(int propertyIndex) might return a single DataObject in cases where get() is invoked on an object corresponding to a registered schema element or type. It might return a list<DataObject> in cases where the schema definitions corresponding to the payload are unrecognized.

Scope mgmt – shared scopes

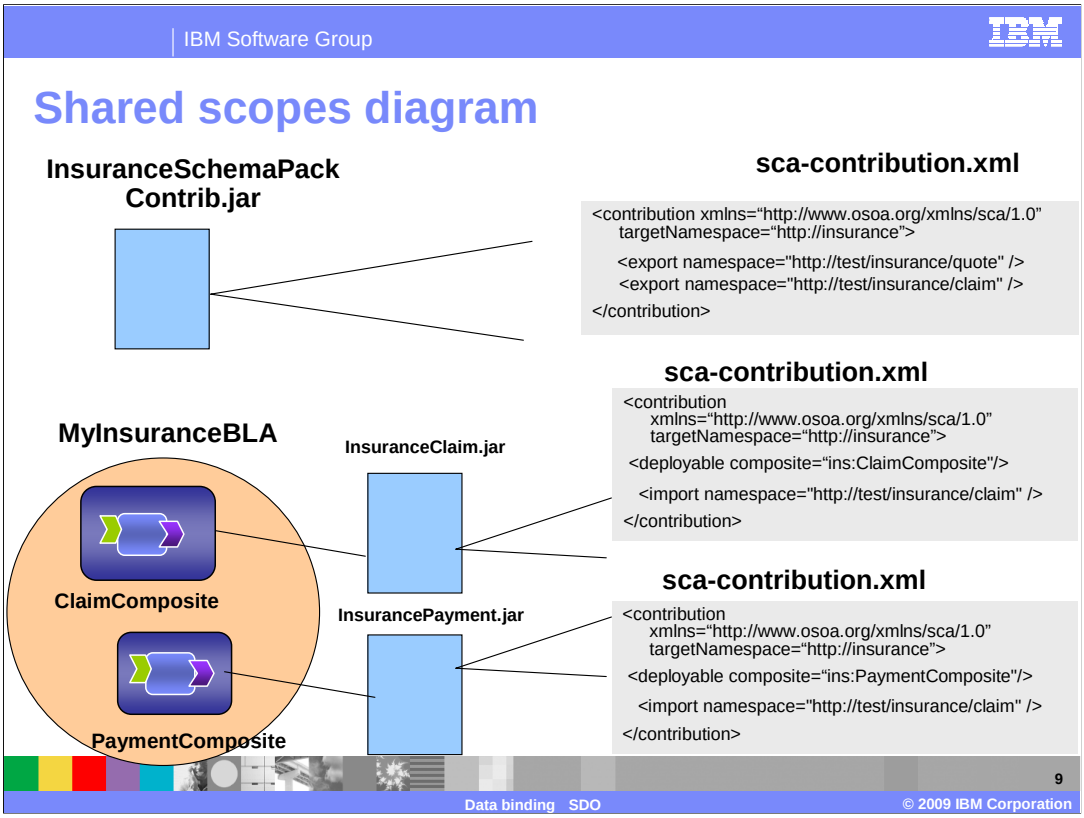
- Multiple components/composites can load schema definitions into a shared SDO scope
  - reduces memory footprint
- Uses SCA contribution import and export
- Integrated with pass-by-reference optimization
- Shared scopes also work within a business level application (BLA)

Multiple components or composites can load commonly-used schema definitions into a shared SDO scope. This reduces memory footprint.

Shared scopes also use SCA Contribution import/export. You can package a set of shared schemas into a single contribution, export certain XML namespaces, and import those namespaces from your composites' contributions.

Another aspect of shared scopes is that it is also integrated with pass-by-reference optimization.

It also works within a business level application. For each shared contribution exporting schema definitions (XSD), any two composites in the same business level application that import those definitions will load from a single SDO scope.



Here is an example of shared scopes. Note how ClaimComposite and PaymentComposite from MyInsuranceBLA load from a single SDO

API to access default scope

@com.ibm.websphere.soa.sca.sdo.DefaultHelperContext

- The runtime will inject a public or protected field/setter with the application-specific HelperContext

```
import com.ibm.websphere.soa.sca.sdo.DefaultHelperContext;
…
@DefaultHelperContext
protected HelperContext defaultHC;
….
DataFactory dataFactory = defaultHC.getDataFactory();
DataObject person =
       dataFactory.create("http://test", "Person");
```


As you saw in some earlier slides, SCA and SDO define a **default HelperContext** for a given application and enable the SCA application to access this programmatically. The default HelperContextFactory is accessible through the interface commonj.sdo.helper.SDO. The runtime will inject a public or protected field or setter with the application-specific HelperContext.

In the example shown, the XSD type with QName **{http://test}Person** is defined in an XSD file packaged in the same contribution as the deployable composite using this Java code in a component. The annotation gives you access to this scope.

JAX-WS based programming model

- JAX-WS defines interface-level mapping, between WSDL portType operation and Java interface method
- JAX-WS annotations significant in SCA applications using SDO
  - @RequestWrapper, @ResponseWrapper, @WebParam, @WebResult
- For individual input parameter and return types, SDO defines mapping between XSD and Java, rather than JAXB
- Exception/fault handling defined by JAX-WS
  - Mapping between the “fault bean” and schema defined by SDO, rather than JAXB

SCA feature pack uses JAX-WS to define the operation-level mappings between WSDL/XSD and the Java interface method, and uses SDO to define the mapping between XSD types and the corresponding Java parameter types. Thus, there is a single SCA programming model based on JAX-WS, with the ability to "plug in" either the JAXB data binding or SDO data binding.

One consequence of this is that the JAX-WS annotations like those shown here are significant in SCA applications using SDO.

Another important consequence is that the product uses JAX-WS to define the mapping between a Java exception thrown or caught in Java clients and implementations, and the "fault bean" that is serialized "on the wire”.

The "fault bean" can be an SDO of type commonj.sdo.DataObject, in which case the SDO XMLHelper is used to serialize or deserialize the fault bean to or from the wire format.

SDO known versus. unknown type

- The (XSD/SDO) type of the DataObject can either be known or unknown
  - "unknown" type scenario is treated as the SDO-equivalent of xsd:anyType
  - can affect the programming model used to interact with DataObject's properties
- In SCA, top-down approach used to program with "known" types, bottom-up approach results in "unknown" types

When working with a DataObject in your SCA client or SCA implementation, the (XSD/SDO) type of the object can either be known or unknown.

The "unknown" type scenario is treated as the SDO-equivalent of xsd:anyType. This can affect the programming model used to interact with your DataObject's properties. In SCA, top-down approach (starting from WSDL), can be used to program with "known" types, whereas bottom-up approach (starting from Java) typically results in "unknown" types.

Example: Schema definition unknown

XML

```
<schema targetNamespace="http://person"
      xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="Person">
      <sequence>
        <element name="company"
      type="comp:Company"/>
        <element name="firstName" type="string"/>
        <element name="lastName" type="string"/>
      </sequence>
    </complexType>
</schema>
```

Since SDO has no knowledge of the Person XSD type, it must assume its child property might correspond to something like this: maxOccurs="unbounded" element returning a List instead of a singular object

```
DataObject personDO = ...; // corresponds to 'Person' XSD type
List<Object> child1 = (List<Object>) personDO.get("company");
```


This is an example where Schema definition is unknown. In this scenario, since SDO has no knowledge of the Person XSD type, it assumes its child property might correspond to an unbounded element. It therefore returns a list instead of a singular object.

Example: Schema definition registered

```
<schema targetNamespace="http://person"  xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="Person">
        <sequence>
          <element name="company" type="comp:Company"/>
          <element name="firstName" type="string"/>
          <element name="lastName" type="string"/>
        </sequence>
      </complexType>
</schema>
```

This is the same XSD as in the last example, but this time SCA has registered the XSD in the HelperContext

Here SDO knows about the Person XSD type and knows its first child property corresponds to the 'company' element. Since for 'company', the maxOccurs="1", SDO returns a single DataObject rather than a List

```
DataObject personDO = ...; // corresponds to 'Person' XSD type
DataObject company = personDO.getDataObject("company");
```


This is an example where schema definition is registered. This is the same XSD as in the last example, but this time SCA has registered the XSD the HelperContext class. In this example, SDO knows about the Person XSD type and knows its first child property corresponds to the 'company' element. Since for 'company', the maxOccurs="1", SDO returns a single DataObject rather than a list.

Top-down versus bottom-up

- Top-down approach, the <interface.wsdl> allows the payload to be deserialized into the “known” type.
  - defined by the XSD type of the doc-lit WSDL’s message element.
- In bottom-up approach, there is no particular XSD type associated with a Java DataObject.
  - It is treated as an “unknown” type, that is xsd:anyType
  - Note that if you generate Java from a WSDL and do not use <interface.wsdl> the XSD type information is lost

In the top-down approach, the <interface.wsdl> describing the service in SCDL allows the payload to be deserialized into the “known” type defined by the XSD type of the doc-lit WSDL’s message element.

In the bottom-up approach, there is no particular XSD type associated with a Java DataObject, and so it is treated as an “unknown” type, that is xsd:anyType. If you do a publish WSDL, you will also see this mapped to xsd:anyType.

Note that even if you generate the Java from a WSDL (for example, in Rational® Application Developer) and do not use <interface.wsdl> the XSD type information is lost. This is unlike when using JAXB.

Steps for using SDO 2.1.1 in SCA feature pack

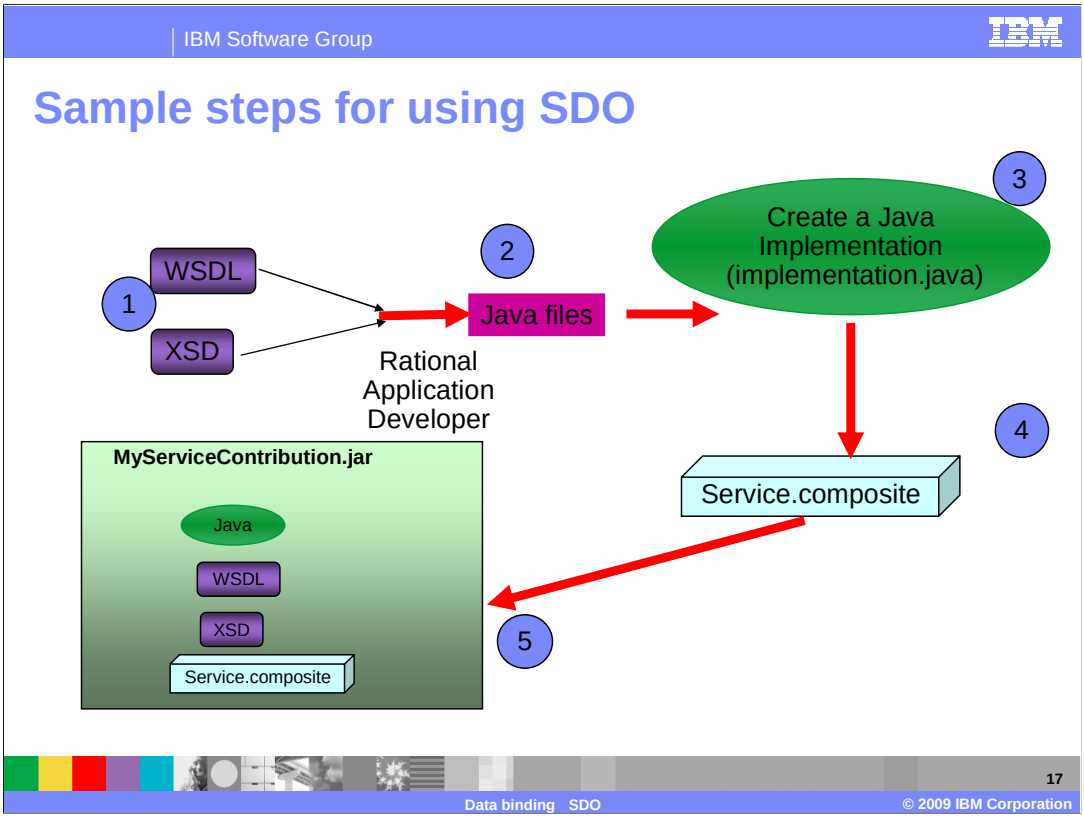
1. Start from your WSDL and XSD files describing your service interface (as a WSDL portType)
2. Produce a corresponding Java interface
3. Write your SCA Java client or component implementation
4. Write the composite definition defining the reference or service interface in terms of your original WSDL portType
5. Package the WSDL and XSD files in the same contribution JAR as the deployable composite

Top-down approach steps for using SDO 2.1.1 in SCA feature pack. First, start from your WSDL and XSD files describing your service interface as a WSDL portType.

Then produce a corresponding Java interface. At the moment, this is not an easy task because of the absence of a supported WSDL-to-Java tool. The information center has manual instructions on how to do this.

Next, write your SCA Java client or component implementation using the dynamic SDO programming model. Then write the composite definition defining the reference or service interface in terms of your original WSDL portType

Finally, package the WSDL and XSD files in the same contribution JAR as the deployable composite that deploys your client or implementation component.



Here is a graphical view of the steps. A good example of this process is given in the information center. You can use Rational Application Developer to generate Java files or a semi-automated process of modifying the generated output of wsimport. To reiterate, start with WSDL, generate your Java classes, write your Java client, write your composite definition, then package the WSDL and XSD files in the same contribution JAR as the deployable composite.

SDO and JAXB comparison

- SDO defines a Java binding framework of its own
- SDO offers a dynamic API, analogous to Java reflection but with more capabilities
- SDO does not require generating and packaging Java classes corresponding to XSD definitions (like JAXB)
- SDO provides uniform access to data of various types in addition to XML (this feature is not used in SCA, however)

SDO defines a Java binding framework of its own, but it goes one step further. While JAXB is only focused on a Java-to-XML binding, XML is not the only kind of data being bound to SDO. SDO provides uniform access to data of various types, only one of which is XML. SDO also offers both a static and dynamic API, whereas JAXB only provides a static binding.

SCA SDO limitations

- Only doc-lit, BasicProfile-compliant WSDL is supported
  - Note that: this is an overall SCA feature pack restriction.
- Static/generated SDOs corresponding to complex types not supported
- java.lang.Object as a Java interface or implementation parameter type not supported

Here are a few limitations. First, WSDL must be BP compliant. The assumption here is that you are dealing with doc-lit WSDL (not necessarily doc-lit-wrapped though), which is WS-I Basic Profile compliant.

Second, in the SCA feature pack, any use of static or generated SDOs corresponding to complex types is not supported.

Finally, java.lang.Object as a Java interface or implementation parameter type is not supported. There is no intention to mix the dynamic aspects of java.lang.Object with dynamic SDO.

Section

Summary

In summary…

Summary

- SDO provides a dynamic programming model for writing SCA Java applications working with schema-derived data
  - Dynamic programming model eliminates the need to generate Java classes from schema definitions
- Provides convenient and advanced methods for accessing data at run-time by index, property name, or XPath-like expression"

SDO provides a dynamic programming model for writing SCA Java applications that work with schema-derived data. This dynamic programming model eliminates the need to generate Java classes from schema definitions (like JAXB). It abstracts over certain XML details, and provides convenient and advanced methods for accessing data at run-time by index, property name, or XPath-like expression.

And from the SCA perspective, SDO adds the dynamic API.

References

- IBM Education Assistant:

http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wasfpsca/wasfpsca/1.0/JAXB.html

- Introduction to service data objects
  http://www.ibm.com/developerworks/java/library/j-sdo/
- SDO 2.1 specification:
  http://www.osoa.org/display/Main/Service+Data+Objects+Home
- SDO JSR235 Data objects
  http://www.ibm.com/developerworks/java/library/specification/j-jsr235/
- SCA feature pack information center

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.soafep.multiplatform.doc/info/welcome_nd.html

Here are some reference links for more information.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASv7SCA101_SDO_databindng.ppt

This module is also available in PDF format at: ../WASv7SCA101_SDO_databindng.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM Rational WebSphere

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Rational is a trademark of International Business Machines Corporation and Rational Software Corporation in the United States, Other Countries, or both.

Java, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.