Compiler Transformations to Enable Synchronous Execution in an RIA Runtime

Rich Internet applications (RIAs) provide an immersive experience that blends the Internet's expansive reach with the desktop computer's rich performance. Client applications typically connect to the server using asynchronous communication protocols, which lets users interact with the application without getting blocked due to server response latency. Although this programming model enables high user interactivity, it increases the complexity of synchronizing multiple pieces of program logic. This increased complexity can be alleviated if synchronous program execution is enabled without blocking the user interface. The authors' technique based on compiler transformations enables synchronous execution on an asynchronous RIA programming model.

Anantharaman P. Narayana Iyer, Arijit Chatterjee, and Jyoti Kishnani
Adobe Systems India

The advent of Web 2.0 and its associated read/write Web access patterns places a major emphasis on the user experience. Although this was a critical need even for traditional Web applications, emerging Web 2.0 and cloud applications extend it even further. The two primary factors that impact user experience on the Web with browser-based access are

- page-based metaphor, in which the page on view is refreshed while a new page gets loaded; and
- the synchronous nature of communication between the client and server, in which the client user interface (UI) is blocked until it receives the server's response, inhibiting user interactivity.

The rich Internet application (RIA)[1] architecture enables significant computing dedicated to presentation to occur on the client, supported by a sophisticated client RIA runtime (see the "Rich Internet Applications Architecture" sidebar for more background). The server exposes services that encapsulate the business logic while the client handles the presentation. This approach has a more scalable architecture due to separation of concerns and the use of a service-oriented architecture (SOA). It also leverages the client system's compute power, which

1089-7801/10/$26.00 © 2010 IEEE

Rich Internet Applications Architecture

Traditional Web applications are based on the popular protocol HTTP and interchange information with servers as HTML. HTML content (pre-HTML5) encapsulated data within presentation logic (for example, the data for a table was written within the table markup). This meant that every time the Web application needed to modify data or its presentation, the server would have to create a new HTML page with the new data embedded in the new presentation logic.

Over the past decade, computers and other client devices have achieved significant computation capability and present high-fidelity experiences. This capability has been leveraged by client-side runtimes that can read data from servers or services and create their own presentation logic on the client. Given the client's ability to render rich content, it leverages client runtimes to create high-quality presentations for data served from the back end. The added advantage of using client runtimes to create and render presentation is that, for a change in presentation, the application need not get new information from server back ends, thus making the client user interface (UI) far more responsive.

Web applications that leverage such rich runtimes to render Internet-based data (spanning everything from textual data to high-definition video and audio content) are known as rich Internet applications (RIAs). This richness is further reinforced by a repertoire of rich visual controls and support for various data formats. RIA runtime vendors have empowered developers to adopt their platforms by providing tools that let developers create the high-fidelity applications and creative assets that go into building RIAs.

RIAs are typically architected into multiple layers delegating data and content storage and mechanisms for accessing data to separate layers (RIAs add the paradigm of digital rights management, persistent connections, streaming data, and so on). Figure A shows the architecture of an RIA, as described here.

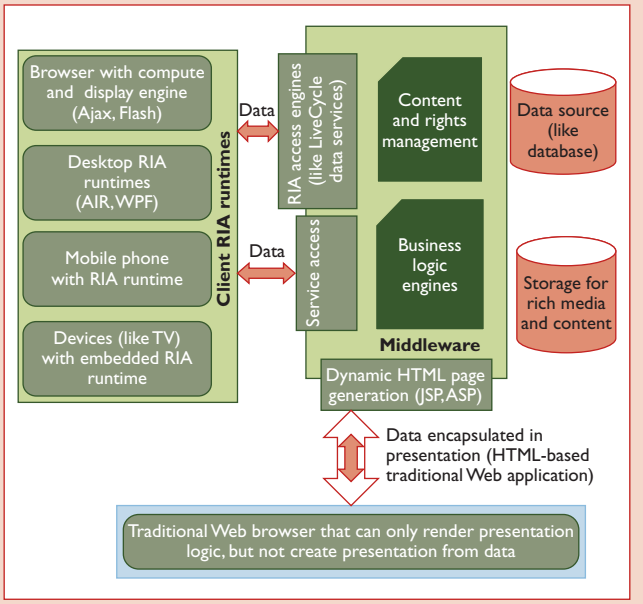

Figure A. Generic architecture of the infrastructure required to enable rich Internet applications.

can result in greater responsiveness because a superior application design can reduce server roundtrips. The separation of server and client logic also implies that architects can implement the server-side business logic using the languages and frameworks that are best suited for that purpose without worrying about the presentation. On the client side, this architecture has led to the evolution of specialized frameworks, programming models, and languages that aim to provide a better user experience on the Web. These client-side frameworks support a wide variety of rich UI controls and multiple protocols for accessing the server.

To address poor interactivity due to synchronous access, RIAs use asynchronous communication. The asynchronous communication model is supported at the browser level through Document Object Model (DOM) APIs such as XMLHTTP or plug-ins such as the Adobe Flash Player.

Although the asynchronous communication model lets the client application be more interactive, it requires that the developer synchronize different pieces of his or her application logic using event handlers. This makes program development complex in many situations as compared to implementing the same functionality in a synchronous programming environment.[2] One mechanism for making programming easy for RIAs is to provide the programmer with a synchronous programming paradigm while developing applications for an asynchronous programming environment, thereby providing a high degree of interactivity without requiring complex programming. Our innovative solution based on compiler transformations addresses this issue without sacrificing scalability and performance.

Achieving In-Order Execution

With client runtimes becoming more capable and SOA growing popular, RIA programmers are building complex Web applications that typically use a combination of calls to various

services to get specialized data and manipulate client data. Most such mashups consume data from one service and pass it to another as input, potentially with some changes. A more generic area that inherently has requirements to sequence service calls is *service composition*,[3] which involves arranging several services in a deterministic workflow to provide a complete solution. All these scenarios are inherently ordered, and thus will directly benefit from a sequential execution pattern.

The requirement for sequentially executing calls to services has traditionally had various solutions. One example is making synchronous calls from the browser. This is most evident in Ajax support for making calls to services with an option to keep the calling function blocked until the service call returns. This is possible in Ajax because its engines are built into browsers (browser plug-ins follow the Netscape Plugin Application Programming Interface (NPAPI) architecture, which disallows plug-ins from placing blocking calls to prevent them from making the browser nonresponsive). This option has had limited success because blocking the browser UI makes for a very poor user experience.

Another approach is to use browser-plug-in-based RIA runtimes, such as Adobe Flash Player and Microsoft Silverlight, which are bound by the NPAPI architecture. They thus require programmers to manually handle any service call sequencing by using popular techniques such as placing code, which depends on the result of a service in a previous call's result handler. Although applications built with these platforms are rich, the programmers must be skilled enough to keep track of service call sequence requirements, and, in nontrivial industrial Web applications, this task can become significantly complex.

Finally, developers requiring more complex sequences of service invocation have relied on server-side programming to achieve their goals. The software industry has built and used several service-composition-based solutions[4] in this working model. However, this approach doesn't let the client programmer leverage mature and high-capability client runtimes to achieve rich user experiences.

Our mechanism achieves in-order execution of asynchronous calls that apply to rich client runtimes, which are inherently asynchronous. It removes the tedium of tracking call sequences from the RIA programmer. Although our concept is generic, we've used the Adobe Flash Player and Adobe Flex framework (www.adobe.com/products/flex/) to implement the solution. Thus, all code fragments in this article are based on MXML and ActionScript 3 (AS3; www.adobe.com/devnet/flex/quickstart/coding_with_mxml_and_actionscript/), the programming languages for Adobe Flex.

Problem Statement

Asynchronous server access is a key characteristic in building rich and responsive client experiences with RIAs. However, several situations exist in which an application must execute logic in order simply because the result of one operation is an input for the following operation. The case for asynchronous communication for RIAs is built on the premise that UIs shouldn't freeze when a service call is in progress — but not all service calls result in UI modification.

RIAs perform nontrivial computation on the client and handle presentation-logic concerns. Some of this computation might involve manipulating data and potentially calling data-oriented services asynchronously. Such services are those that serve data, potentially in a variety of formats, but don't serve any information about how the client UI must present that data.

Often, real-world applications require a sequence of operations in which they read data and subsequently use it. This data access can happen from a data-oriented service. To illustrate the need for such a sequential use of data accessed from a service, we use a simple example that retrieves the location of a nearby weather station, given an airport name. In this example, enabled by the Weather Underground API (www.wunderground.com), we first fetch the list of weather station codes given the airport name code and then use another service to get the name of the city where the weather station is located by passing to it the weather station code returned earlier. Although the first service (`service1`) returns an array of weather station codes for a given airport code, for simplicity, we use only the first code as input for the second service (`service2`); we can sufficiently explain the sequential execution concept with this assumption. The calls to the services must be in sequence because we can make the one to get the weather station's city name only after the service call to get the list of weather station codes completes and the

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
                        xmlns:s="library://ns.adobe.com/flex/spark"
                        xmlns:mx="library://ns.adobe.com/flex/mx>
  <fx:Script>
       <![CDATA[
               import mx.controls.Alert;
               import mx.events.FlexEvent;
               import mx.rpc.events.FaultEvent;
               import mx.rpc.events.ResultEvent;

               import service2_resultHandler
                                    (event:ResultEvent):void
               {
                     var res:XML;
                     var city:String;
                     res = event.result as XML;
                     city = res.location.city.toString();
                     mx.controls.Alert.show(city);
               }
               protected function service1_resultHandler (event:ResultEvent):void
{
                     var res:XML;
                     var resultList:XMLList;
                     var newURL:String =
"http://api.wunderground.com/weatherstation/WXCurrentObXML.asp?ID=";
                     res = event.result as XML;
                     resultList = res.nearby_weather_stations.pws.station;
                     newURL = newURL + resultList[0].id.toString();

                     service2.url = newURL;
                     service2.send();
               }
               protected function service1_faultHandler(event:FaultEvent):void
               {
                   mx.controls.Alert.show(event.fault.faultString);
}
               protected function textInput_enterHandler(event:FlexEvent):void
{
                     var url:String =
"http://api.wunderground.com/auto/wui/geo/GeoLookupXML/index.xml?query="+textInput.
text;
                     service1.url = url;
                     service1.send();
               }
      ]]>
   </fx:Script>
   <fx:Declarations>
               <s:HTTPService id="service1" resultFormat="e4x"
result="service1_resultHandler(event)" fault="service1_faultHandler(event)"/>
               <s:HTTPService id="service2" resultFormat="e4x"
result="service2_resultHandler(event)" fault="service1_faultHandler(event)"/>
   </fx:Declarations>
   <s:TextInput id="textInput" widthInChars="20"
enter="textInput_enterHandler(event)" text="enter airport code and hit enter" />
</s:Application>
```

This is the call to service2 from within the result event handler of result1 to ensure sequentiality between the two

Figure 1. Code listing for the weather station city-name-retrieval application. This figure shows the traditional source code that a rich Internet application developer must write to achieve in-order execution of services by manually including invocation of subsequent services in the result handler for the preceding service call. The code is written in MXML and ActionScript 3.

input parameter for the weather station city-name-retrieval service is available.

Figure 1 shows a sample implementation of the application that retrieves the weather station city name, written in MXML with embedded AS3 code, which accomplishes the following steps:

1. The application displays a text input (named `textInput`) that takes the input of an airport name as text.
2. Hitting enter invokes the `textInput_enterHandler`, which in turn initiates a call to the REST-based service, `service1`, defined in the `<fx:Declarations>` section.
3. The result event handler `service1_resultHandler` handles a successful result from the service `service1`, whereas a fault will result in the fault event handler `service1_faultHandler`'s invocation.
4. On getting valid data from successfully invoking `service1`, the application invokes

the REST-based service `service2` (again declared in the `<fx:Declarations>` section) from `service1`'s result handler.
5. If the call to `service2` succeeds, the system invokes the result event handler for it, `service2_resultHandler`.

Note that the developer accomplishes in-order execution of the access to `service1` and `service2` by placing the call to `service2` in `service1`'s result event handler.

This simple example has complexity stemming from the following characteristics:

- This code has utilized 45 lines of developer-written source code.
- The developer must manually detect all in-order execution sequences and ensure that service call ordering is maintained by moving the dependent code's invocation to the previous call's event handler (as described in step 4).
- In Figure 1, with just two sequential services and a depth of two functions in the call stack, the code to ensure in-order execution is fairly involved. In typical enterprise or consumer RIAs, the call pattern will have more service calls, and the depth of the call stack is likely to be much higher, making the code that the RIA developer must write significantly more complex.

We can achieve this same result with ease if we have framework support for ordering the calls to the service. RIA developers' lives would be much simpler with a compiler that compiles the RIA source code in the format that runs on the client runtime, automatic transformation of the source code to maintain service calls' sequential nature without worrying about call graph analysis, and correctly written sequenced event handlers. Our work provides a basis for the RIA programmer to write the application using synchronous logic, which we transform automatically to a pattern that the underlying asynchronous environment supports.

In Figure 2, we achieve the same functionality as in Figure 1 without any intricately sequenced event handlers while also ensuring no freezing occurs in the UI. We accomplish this using the compiler techniques we describe in the next section. This code fragment (Figure 2) clearly shows a far simpler and more intuitive programming model for achieving the task at hand using a two-service sequence. Not only does it use fewer lines of developer-written code (30 lines, in this case), this mechanism also obviates the need to track the service execution sequence beyond writing the service calls in sequence. The code in Figure 2 is divided into sections marked B1, B2, and B3 for code that executes in sequence, and C1 and C2 for code that makes an asynchronous call.

Solution

We developed our solution using a compiler transformation-based approach that enables sequential service calls in an underlying asynchronous environment. We demonstrate a mechanism that makes the compiler detect the calls to asynchronous services and transforms the code to automatically ensure sequential execution. This ensures, for example, that the source code written in the pattern Figure 2 describes will indeed work in sequence.

Our solution identifies the function to be transformed and splits it into multiple code blocks. The compiler then chains these blocks using event handlers such that completing one block results in events that invoke the subsequent block. It chooses where to split the code block such that each code block ends when an asynchronous call occurs. The compiler automatically performs all this transformation and is thus completely transparent to the RIA programmer.

In the most general case, the entry conditions for our solution are as follows:

- Let's suppose the basic program unit in which we want to enforce in-order execution to be a function.
- This function (in the Figure 1 example, `textInput_enterHandler()`) may contain any arbitrary number of invocations of other asynchronous functions (in our example, there is an invocation of the function `service1.send()`).
- The asynchronous functions getting invoked might require and depend on the variables created or initialized by an earlier service call (in our example, the call to `service1` generates the input parameter for `service2`).

The detection of the entry conditions in the RIA source code triggers the compiler transformation.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
                        xmlns:s="library://ns.adobe.com/flex/spark"
                        xmlns:mx="library://ns.adobe.com/flex/mx"
                           minWidth="955" minHeight="600">
   <fx:Script>
       <![CDATA[
               import mx.controls.Alert;
               import mx.events.FlexEvent;

               import textInput_enterHandler(event:FlexEvent):void
               {
                     var url1:String =
"http://api.wunderground.com/auto/wui/geo/GeoLookupXML/index.xml?query="+textInput.text;
                     var url2:String =
"http://api.wunderground.com/weatherstation/WXCurrentObXML.asp?ID=";
                     var res:XML;
                     var resultList:XMLList;
                     var city:String;                                          B1

                     service1.url = url;
C1                   res =  service1.send();//CALL TO ASYNC SERVICE

                     resultList = res.nearby_weather_stations.pws.station;
                     url2 = url2 + resultList[0].id.toString();                B2
                     service2.url = url2;
C2                   res = service2.send();//CALL TO ASYNC SERVICE

                     city = res.location.city.toString();
                     mx.controls.Alert.show(city);                             B3

               }
       ]]>
   </fx:Script>
   <fx:Declarations>
       <s:HTTPService id="service1" resultFormat="e4x"/>
       <s:HTTPService id="service2" resultFormat="e4x"/>
   </fx:Declarations>
   <s:TextInput id="textInput"  widthInChars="20"
   enter="textInput_enterHandler(event)" text="enter airport code and hit
   enter" />
</s:Application>
```

The call to service2 is just written in sequence after the call to service1 and the compiler transformation ensure sequential execution

Figure 2. Code fragment. We rewrote the code from Figure 1 based on the compiler's ability to transform it to achieve in-order execution. B1, B2, and B3 define synchronous code sections, while C1 and C2 are asynchronous calls.

Our Algorithm

Providing a synchronous programming paradigm on top of an asynchronous programming environment and runtime involves two broad steps.

The first is detecting asynchronous functions. We define an asynchronous function as one that calls a system or library function that's known to be asynchronous (for example, the network connection APIs for HTTPService and Webservice from Adobe Flash Player, or asynchronous I/O calls that operating systems provide) or that calls another function that the algorithm has previously detected to be asynchronous.

The second step is transforming each asynchronous function into a sequence of functions such that the application invokes the newly created functions in a sequence that forms a synchronous execution chain. This chain's execution results in the same functionality as the original function that was transformed.

Step 1: Detecting asynchronous functions. Figure 3a depicts the call graph for the code in Figure 2, but this graph isn't sufficient to explain our methodology's concepts. Instead, we use the call graph Figure 3b depicts to describe this step of the algorithm. We refer to the Figure 2 example to draw parallels with the sample.

We considered all the functions in the application for analysis; for example, let's form a set of functions $F = \{f_1, f_2, \ldots f_6\}$ (see Figure 3b). An example of f_n defined here is the function `textInput_enterHandler` in Figure 2.

Mark the function that makes an asynchronous call — in our example, it's f_6, identified in green in Figure 3. So f_6 is similar to the function `service1.send()` call in Figure 2. Form a graph such that the nodes are function definitions, and the edges represent a function call. In Figure 3b, each arc (for example, the arc from f_3 to f_6) is a function call. An example is the call made from `textInput_enterHandler` to `ser-`

`vice1.send()` in Figure 2.

For each node in the call graph, evaluate the node's reachability[5] to the asynchronous node. In this case, the function call sequences are $f_4 \rightarrow f_3 \rightarrow f_6$ and $f_5 \rightarrow f_6$. In the sample in Figure 2, only one such call exists, the one from `textInput_enterHandler` to `service1.send`.

Mark all function nodes that are reachable to the asynchronous node (f_6 in our example) as asynchronous and the rest of the nodes as synchronous nodes. This will cause `textInput_enterHandler` to be marked as asynchronous in the sample code in Figure 2.

The algorithm iterates through every function node in the application and ends after it evaluates all the functions, as described, to be synchronous or asynchronous. In this case, f_3 and f_5 get marked as asynchronous because they're directly connected to f_6; by induction, f_4 also gets marked asynchronous because it directly calls f_3, which the algorithm already marked asynchronous.

Step 2: Transforming an asynchronous function. After the algorithm has identified all user-defined functions to be either synchronous or asynchronous, the compiler progresses to creating the transformed blocks and generating the code. We refer to Figure 4 for this step.

Let's assume that each function consists of a set of statements numbered S_1 to S_n. Partition the statements into blocks of code B_1 to B_{k+1}, such that a given block B_i consists only of statements that are guaranteed to execute synchronously ending with an asynchronous call as the last statement in the block. The asynchronous calls are numbered C_1 to C_k, where the function has a total of k asynchronous calls.

Next, transform each block B_i such that the entire function is now transformed into a set of blocks in which i ranges from 1 to $k + 1$ (where k is the number of asynchronous calls in the function). Each block is constructed into a function (*block functions*), and these functions should have access to the context of the initial pre-transformation function (in our example, `textInput_enterHandler`). Each block ends with a call to an asynchronous function except for the last block, which ends with the return statement. Introduce the code to register result and fault handlers[6] for the asynchronous calls C_1 to C_k (see http://livedocs.adobe.com/flex/3/html/help.html?content=16_Event_handling_1.html).

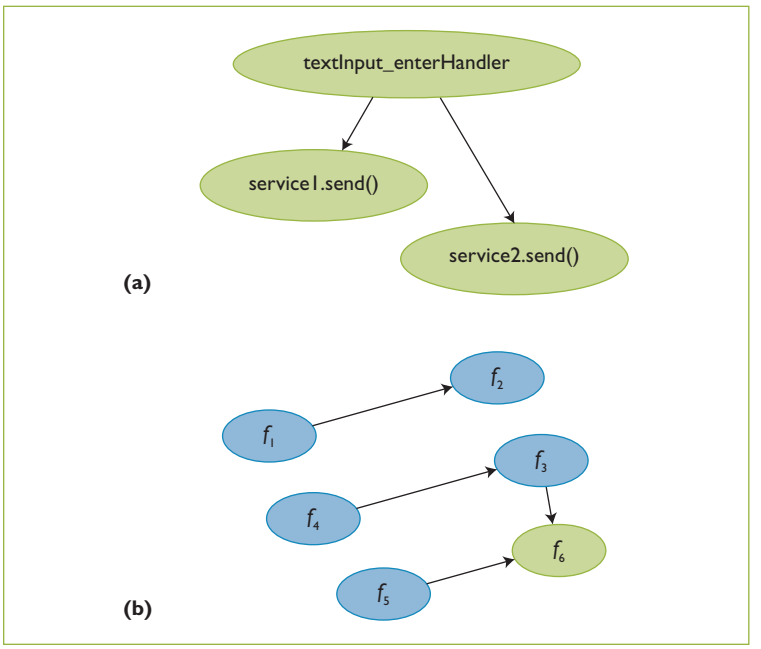

Figure 3. Call graphs for the code in Figure 2. (a) This trivial case, in which all user-written functions are asynchronous, isn't sufficient to explain our methodology's concepts. Instead, we use (b) a graph with two disjoint call paths, one with an asynchronous call and one without.

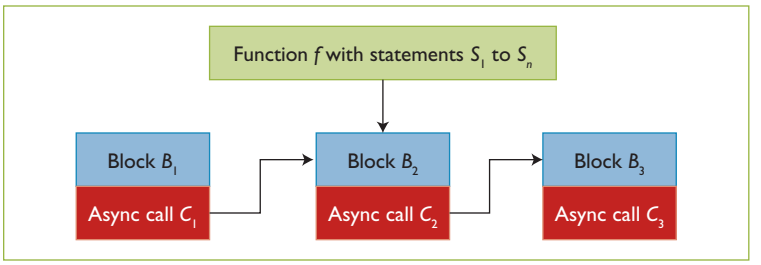

Figure 4. Transformation of the function into a chain of blocks. The runtime executes these blocks sequentially based on event triggers.

Now, chain the blocks together as a logical linked list, such that the result handler of the asynchronous function in block B_i is set to the function block B_{i+1}. The final block B_{k+1} returns the expected return value to the calling function. In the sample code in Figure 2, the XML result is returned to the XML variable `res` defined in the function `textInput_enterHandler`.

If another function calls this function, the caller function is also marked as an asynchronous function, and the compiler applies the same algorithm to it. However, if the calling function is a library or system function defined to be asynchronous, or the root node, the algorithm ends. In Figure 2, the function `textIn-`

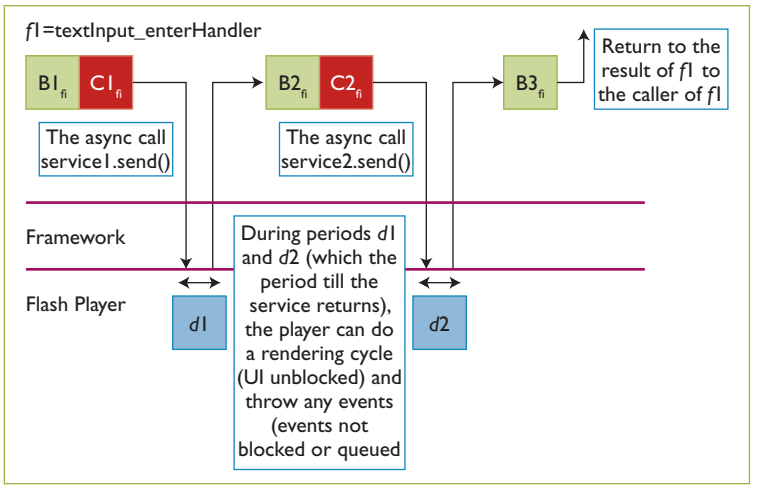

Figure 5. The time-sequence diagram depicting the partitioned code blocks' execution, post-translation. The time segments d1 *and* d2 *let the runtime keep the UI interactive even though the server call is in progress because the control has shifted from the function back to the framework code.*

`put_enterHandler` is called from the Flex framework library, and thus the algorithm ends after evaluating it.

Transformation Advantages

Although the compiler is making significant changes to the original code, its functionality remains unchanged. We can prove this not only through practical testing but also theoretically.

We can assert that the transformed code is functionally accurate as follows. Let's assume the program consists of a set of sequentially executing statements S_1 through S_n, and that the application is making a set of asynchronous calls A_1 to A_m from within the program. So, the RIA programmer can mix S_i (where i ranges from 1 to n) and A_j (where j ranges from 1 to m) in any logical sequence. Each statement S_{i+1} executes only after all preceding statements (statements S_1 to S_i) have completed execution and results produced due to the execution are available.

Every block consists of sequential synchronous code and ends at the boundary of an asynchronous call. Hence, all the statements in the block satisfy the correctness requirement as described in the algorithm. The compiler converts the subsequent statements after an asynchronous call into new blocks as functions (the new block functions B_1 to B_{m+1}). Each succeeding block is called through a result handler registered for the preceding block's terminating asynchronous call. Hence, the programming environment guarantees execution correctness.

Although we have explained this solution in the context of the Flash Player and the Flex framework, the underlying compiler technique is applicable to all RIA platforms, whether compiled or scripted. Compiled RIA platforms, such as Flex, perform the compiler transforms during the compilation process that converts the source to the platform-independent bytecode (Actionscript Byte Code, in the case of Flex). This ensures that the generated bytecode includes the transformed code. In the case of scripted runtimes, by scanning the script, the interpreter can ensure whether an asynchronous call needs to be in sequence (the script's interpreter could scan ahead to analyze whether the service call's result will be used as input for any subsequent code or service invocation) and make changes to its execution path to ensure in-order execution.

In a traditional asynchronous programming environment, the developer writes the event handlers explicitly to attain synchronization between different parts of the application. Our methodology abstracts event-handler development and makes it implicit through the use of compiler transformations. This process doesn't use any busy waits or blocking calls. The system handles the events with the same responsiveness, and no execution — either of the UI or the event subsystem — is held up due to our generated transformations. Hence, an application's performance and interactivity is as good as the nontransformed application that uses explicit event handlers to synchronize.

Execution Sequence

The diagram in Figure 5 describes the execution pattern in a temporal sequence. Figure 2 describes the sections of code, which are split into blocks. The compiler makes each block into a new function based on the algorithm described earlier. The service calls are inherently asynchronous, so after the call has been dispatched but before the result from that call has returned, the runtime and framework are free to keep the UI alive — that is, the application remains completely interactive. The periods during which this situation occurs are $d1$ and $d2$ (as Figure 5 shows), where $d1$ and $d2$ represent the periods when the runtime has control of the execution because the application has

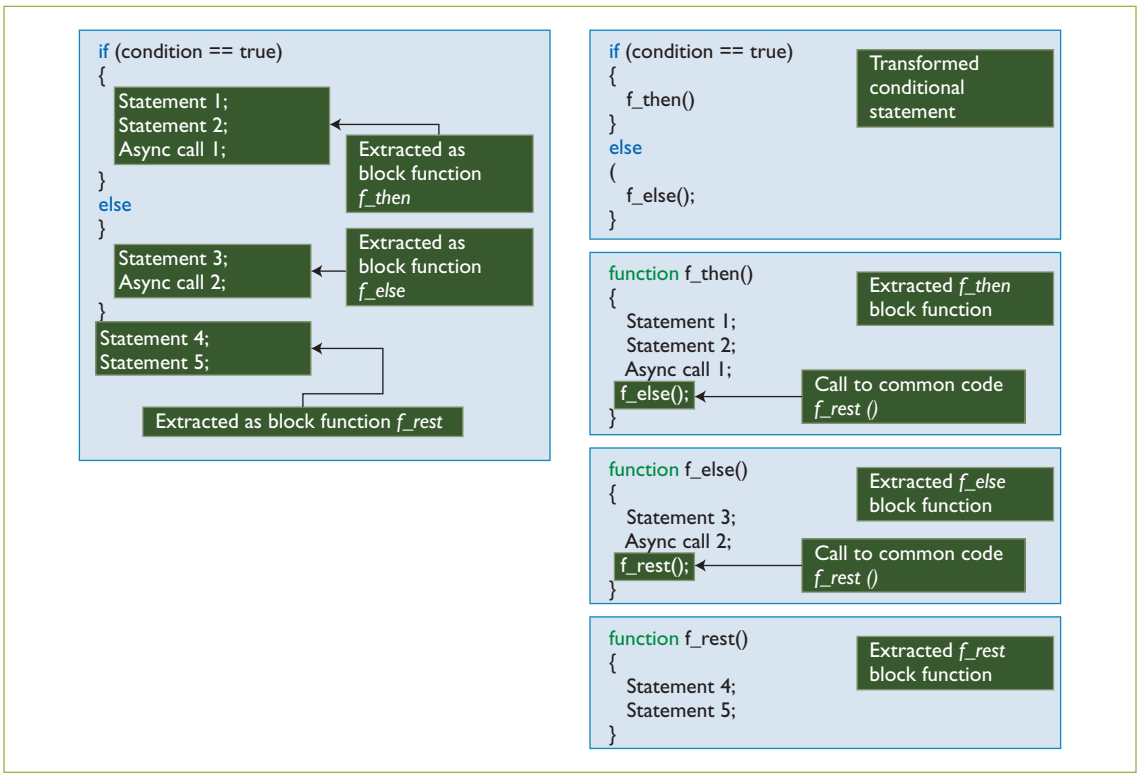

Figure 6. Transformation of conditional explained using the `if-then-else` *construct. Transformation of conditionals ensures synchronous execution in an asynchronous environment.*

made the asynchronous call A_i but the results of that call aren't yet available.

Once the service calls (at the end of any block B_i) complete, the result handler for the service is the block function B_{i+1}, which the compiler created by splitting out the original code (in Figure 2). The temporal sequence also shows, conclusively, that the transformed application maintains the order of execution the developer prescribed, even though the underlying runtime is asynchronous and thus nonblocking.

Handling Loops and Conditionals

The transformation we previously described handles transformation of code that's meant to execute in sequence. However, real-world applications contain execution-sequence changing constructs, such as `for` loops, `if-then-else` statements, and `while` loops. We can largely categorize these constructs as *loops* and *conditionals*. When a function makes an asynchronous call within the body of a loop or conditional, additional transformations are necessary. We've developed mechanisms to handle loop and conditional transformations. The mechanisms are applicable for situations that might have an arbitrary combination of these programming constructs.

Transforming conditionals. Let's consider the `if-then-else` construct as the basis for all conditionals (see Figure 6). Several additional steps are required to detect block functions.

Let's call the set of statements that execute when the tested condition is true the `then` block and the set of statements that execute when the condition is false the `else` block. Extract the `then` block into a separate block function named the `then_function()`. Similarly, extract the `else` block into a block function named the `else_function()`. The compiler extracts common code that executes after either the `then` block or `else` block completes as a block function named `f_rest`.

Because each extracted block function could contain an arbitrary number of asynchronous calls, conditional statements, and loops, the compiler again passes the extracted functions through the same transformation process. This process repeats for every new block function generated and terminates when all block functions contain only one asynchronous call or return call at the end of the block.

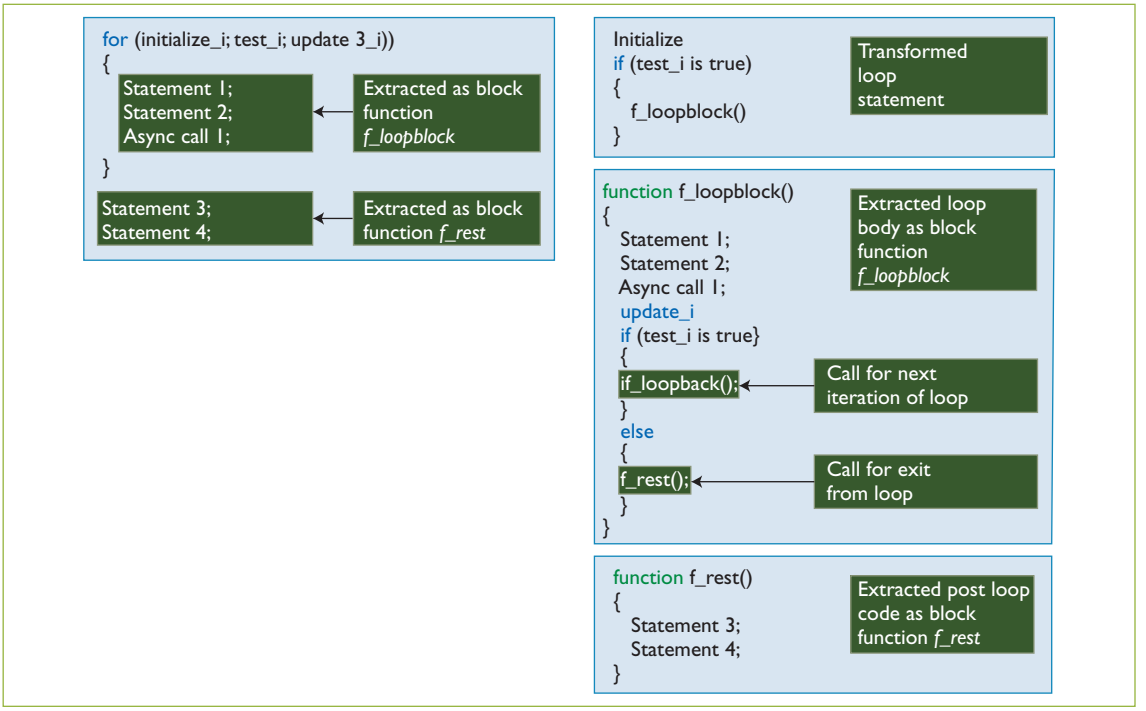

Figure 7. Transformation of loop explained using the `for` construct. Transformation of loops ensures synchronous execution in an asynchronous environment.

Loop transformation. Let's look at the transformations required to handle loops, using the `for` loop as the basis construct (see Figure 7).

First, the compiler extracts the loop's entire body as a block function `f_loopblock`. It also extracts the part of the code that follows the loop block as a separate block `f_rest`. Finally, the compiler removes the original source of the condition evaluation that controls the loop block and replaces it as follows:

- If the loop variable has an initialization (for example, the initialization in a `for` loop), the compiler transforms that part into an initialization statement.
- The compiler then inserts the condition evaluation that decides whether the loop block needs to be entered for the first time as an `if` condition, with the `then` part having a call to the `f_loopblock` function and the `else` part having a call to the `f_rest` function.

At the end of the code in the `f_loopblock` function, the compiler adds code to update the loop variable and test whether the loop must be repeated. The `then` part of that condition will make a call to `f_loopblock` while the `else` part will be a call to `f_rest` because the loop no longer needs to repeat.

Because the extracted block functions `f_loopblock` and `f_rest` can both have an arbitrary number of asynchronous calls, loops, and conditional statements, the algorithm again transforms these functions. The compiler applies this transformation to every new block function created until it doesn't result in the creation of a new block function.

Our methodology has the potential to alleviate the complexity inherent in achieving in-order execution in environments that are asynchronous. RIA runtimes are common examples of such environments and thus present a platform in which developers and RIA environments can use our technique often and effectively.

We've successfully implemented the compiler transformation techniques and methodology we described in this article in the Flex 4 SDK compiler. We've also verified this implementation with the Flex framework on the Flash runtime environment. Currently, work is in progress to integrate this modified compiler with Adobe's developer tooling and server/services infrastructure.

We also intend to modify the developer productivity tools such as the debugger and the profiler such that they run with the transformed

code. As the code gets transformed, reported results should be such that developers see them associated with the untransformed code they originally wrote.

References

1. "Macromedia Flash MX — A Next Generation Rich Client," Macromedia white paper, chapter 2, 2002; http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf.
2. S. Katsoulis, "Synchronous vs. Asynchronous Method Calling," *MDM Developer Network*, www.multidmedia.com/support/articles/?action=detail&tid=1.
3. F. Curbera et al., "Bite: Workflow Composition for the Web," *Proc. Int'l Conf. Service-Oriented Computing* (ICSOC 07), LNCS 4749, Springer, 2007, pp. 94–106.
4. X. Liu et al., "Towards Service Composition Based on Mashup," *Proc. IEEE Congress on Services*, 2007, pp. 332–339.
5. T. Reps, "Program Analysis via Graph Reachability," *Proc. 1997 Int'l Symp. Logic Programming*, 1997, pp. 5–19.
6. D. Caromel and G. Chazarain, "Robust Exception Handling in an Asynchronous Environment," *Proc. European Conf. Object Oriented Programming* (ECOOP 05), *Workshop on Exception Handling in Object Oriented Systems*, A. Romanovsky et al., eds., tech. report 05-050, Dept. Computer Science, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, Montpellier-II Univ., 2005.

Anantharaman P. Narayana Iyer is the director of engineering at Adobe Systems India, working for the Platform Business Unit. His research interests include application of languages and compiler technologies to the domain of Web 2.0 and rich Internet applications. Iyer has an MS in electronics from the Indian Institute of Science, Bangalore. Contact him at ananth@adobe.com.

Arijit Chatterjee is a program manager at Adobe Systems India. His research interests include rich Internet application platforms, compilers, and related infrastructure. Chatterjee has a BS in computer science from Bangalore University and an MBA from the Indian Institute of Management, Bangalore. Contact him at arijit@adobe.com.

Jyoti Kishnani is a quality engineering manager at Adobe Systems India. Her research interests include Web 2.0, rich Internet applications, and Web design. Kishnani has a BS in commerce, economics, and business law from Gujarat University. Contact her at jkishnan@adobe.com.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*