

A Non-Invasive Approach to Dynamic Web Services Provisioning

Fei Cao, Barrett R. Bryant, Shih-Hsi Liu, Wei Zhao
University of Alabama at Birmingham
Birmingham, AL 35294 USA
1.205.934.2213
{caof, bryant, liush, zhaow} @cis.uab.edu

Abstract

Service-oriented computing has emerged as a new component-based software development paradigm in a network-centric environment. By using a standard description language and protocol, services can be used to wrap legacy software systems to be integrated beyond the enterprise boundary across heterogeneous platforms. Nevertheless, the challenges come in tandem with the opportunities because of the inherent dynamic characteristics within a distributed environment. In particular, there is a need for dynamic adaptation for provisioned services to accommodate the ever-changing business requirements externally as well as the computing resource status internally, while maintaining the continuousness of service provisioning. We present a dynamic Web Service provisioning approach based on .NET Common Language Runtime, one of the two primary Web Services platforms, exploring the runtime code manipulation at the Intermediate Language (IL) level rather than at the source code level. Meanwhile, we show how the service provisioning can be adapted in a modularized way by complementing the conventional Service-Oriented Architecture (SOA) with a repository of adaptation aspects. Moreover, we demonstrate how dynamic service provisioning can be used for non-functional property assurance.

1. Introduction

Web Services (WS) technology has continually evolved toward wide adoption in numerous business domains. Based on a standard description language and protocol, WS can be used as a common vehicle to wrap up enterprise software application for integration in a distributed environment. Just as conventional distributed component-based software systems, WS can

be built by amortizing the collective effort to independent service providers. Meanwhile, the distributed environment features dynamic status quo of resource consumption and provisioning. This brings forth the need of a *non-functional property assurance* (such as access control, transaction) mechanism to achieve service reliability, which is particularly desirable in finance and other mission-critical business domains. Another desirable need is the *guarantee of service availability*, which requires continuousness of service provisioning rather than shutting down services for maintenance or upgrade. The guarantee of service availability and the non-functional property assurance are at odds to each other in general, as the latter requires adjustment of service provisioning, for which the common practice is to shut down the old version service and bootstrap a new version service. The need for reconciliation between the guarantee of service availability and the non-functional property assurance calls for a dynamic WS provisioning mechanism, i.e., to accommodate non-functional concerns at runtime.

The non-functional property with WS is not a new thing. However, prior work such as IBM's Web Services Level Agreement (WSLA) [2] and HP's Web Service Management Language (WSML)[9] incorporate the notion at higher-level presentation, rather than address it at lower-level platform layer. We believe a platform layer treatment is necessary toward thoroughly addressing non-functional concerns for WS. As a proof-of-concept, this paper presents a dynamic WS provisioning approach based on .NET Common Language Runtime (CLR)[6]. We choose .NET because it is a thorough, fundamental re-architecting of a distributed computing platform based on WS, while other application server support for Web Services tend to be designed more as another client, or presentation tier for the back-end systems, with communication tier based on RMI or RIM/IIOP rather than strictly XML protocol based such as .NET [5].

This paper is organized as follows. Section 2 briefly

describes the background knowledge on .NET-based WS. Section 3 describes our design principle and implementation for dynamic service provisioning. Section 4 describes a use case. Section 5 evaluates the performance. Section 6 describes related work, followed by conclusion and future work in Section 7.

2. Background

.NET framework is a platform for software integration, with Common Language Runtime (CLR) for integrating software at a single operating system process scale, while with XML WS for integration at internet scale. CLR is the .NET equivalent to the Java virtual machine, but offers more features such as using Common Intermediate Language (CIL) [6] to translate .NET languages before execution, which offers cross-language interoperability for .NET languages based on CIL. The code to be translated into CIL and then to be executed by CLR is also called *managed code*. Also CIL includes rich metadata information for describing software module contracts to achieve *managed execution*, with the benefits of security and scalability. The .NET XML WS is a layer built on top of CLR. This layer is not necessarily tied to .NET, as it can be provided over other application servers such as J2EE. However, .NET based XML WS can leverage the benefits of managed execution, as the invocation to .NET XML WS will be run at the CLR level. Therefore, by changing CIL derived from the XML WS implementation language, the behavior of WS can be adapted non-invasively without changing the WS at the source code level. The non-invasive change is often desirable as the WS vendor may deliver the software package in binary form. Also even though it is possible to derive CIL from .NET executable using some de-compilation tools, invasively changing either original source code or derived CIL code will require unloading, recompiling and redeployment of the original WS application, which compromises the availability of WS. Moreover, the invasive change of WS code will pollute the original application such that recovering the original application will become difficult, which introduces the common version control problems for software system.

Instead of providing a full description on .NET CLR and XML WS, the scope of this paper is to present our approach on how to adapt WS at the CLR CIL level to achieve non-invasive dynamic provisioning of WS.

3. The Design and Implementation

3.1 The Architecture of Non-Invasive Dynamic WS Provisioning Framework

Figure 1 illustrates the architecture for the non-invasive dynamic WS provisioning framework. The boxed part represents the underlying adaptation mechanism transparent to the upper-level XML WS application, while the PC monitor icon on the right hand side represents the unit for adjusting the adaptation policy.

Our work is built upon the ASP.NET, a WS implementation package based on the .NET framework. In ASP.NET, Internet Information Server (IIS) is used to accept the incoming WS SOAP message transported over HTTP (1). Upon acceptance of the WS request encoded as a SOAP message, an IIS filter will launch a work process (`aspnet_wp.exe`), which in turn will launch CLR to run the WS application in the mode of managed execution (2). At this point, the WS application is rendered as in CIL subject to be Just-In-Time (JIT) compiled into native code and executed (6). In order to adapt WS, a means is needed to intercept the WS call at CIL level before it is compiled. While it is reasonable to implement the expected functionalities in the CLR open source of millions of lines of code such as Rotor [10], we feel it too expensive for a prototype effort and would like to leave it for future work. Instead, we use CLR profiling API to implement a *Profiler* as event handlers, and register them as listeners for the events generated from the CLR (3). In contrast to the conventional publisher/listener model, which is often of a client-server relationship, the profiler here will be mapped into the same address space for the profiled application as an in-process server.

The events generated from the CLR are the result of managed code, execution, including but not limited to garbage collection, class load/unload, CLR startup/shutdown and JIT compilation. The event of our interest is JIT compilation, for which we implement in-memory CIL manipulation for the event handler.

The adapted CIL will then be JIT compiled and executed resulting in changed WS behavior. A one-shot change to CIL will reduce the traceability of adaptation, impede the removal of the imposed adaptation, and restrict the flexibility of future adaptation. Therefore, we interpose a *Hook* code in the WS application to be

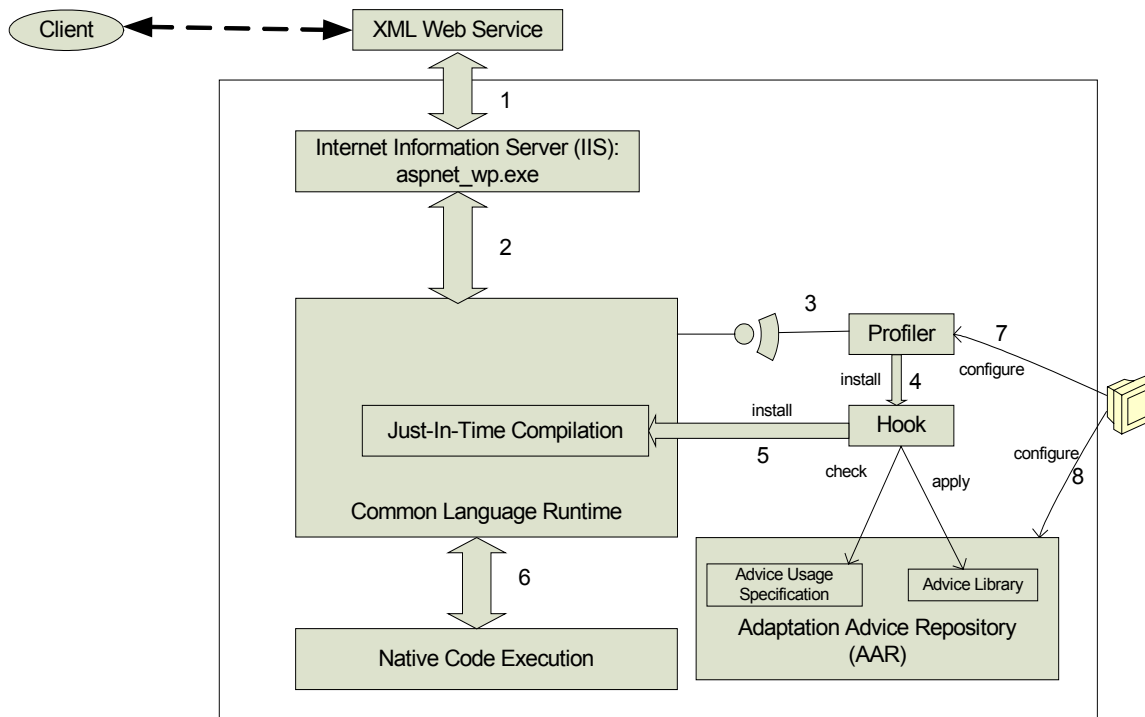


Figure 1: Architecture of non-Invasive dynamic WS provisioning

adapted, which will check the *Adaptation Advice Repository* (AAR) for applicable adaptation advice. The AAR includes an Advice Library storing predefined reusable advice, which is compiled into the managed code form, as well as an Aspect Usage Specification (AUS) component to indicate applicable advice for WS. The Profiler and the AAR are subject to external configuration (7,8): for the former, the configuration is used to narrow down the scope of profiling; for the latter, the configuration is used to dynamically set up adaptation policy, which is detailed in the next section.

3.2 Modularized WS Adaptation

3.2.1 Using hook for dynamic weaving and unweaving adaptation advice

At a WS provisioning site, multiple WS applications may be hosted under uniform domain requirement management. Consequently, the code handling a specific domain requirement may be scattered around

all the WS applications. As such, it is not possible to specify adaptation for every individual WS application upon changing of requirements. Instead, there need a means to abstract the adaptation in a modularized way. Aspect-Oriented Programming (AOP) [7] offers a means to abstract cross-cutting concerns in a modularized way called an *aspect*, and the concerns can be weaved using weaver technology into the base program based on the *join point model*, which specifies the destination to weave concerns. In the same vein, we specify the adaptation advice in the AAR in a modularized way following AOP style. To weave and unweave the specified advice, we instrument the hooks at both the entry (*pre-hook*) and exit point (*post-hook*) of the WS method to be adapted, which are used to check into the AAR to see if corresponding *before advice* and *after advice* is applicable: the former performing some pre-processing before the actual WS method execution, while the later performing some post-processing immediately before the WS method execution returns. Such pre- and post- processing

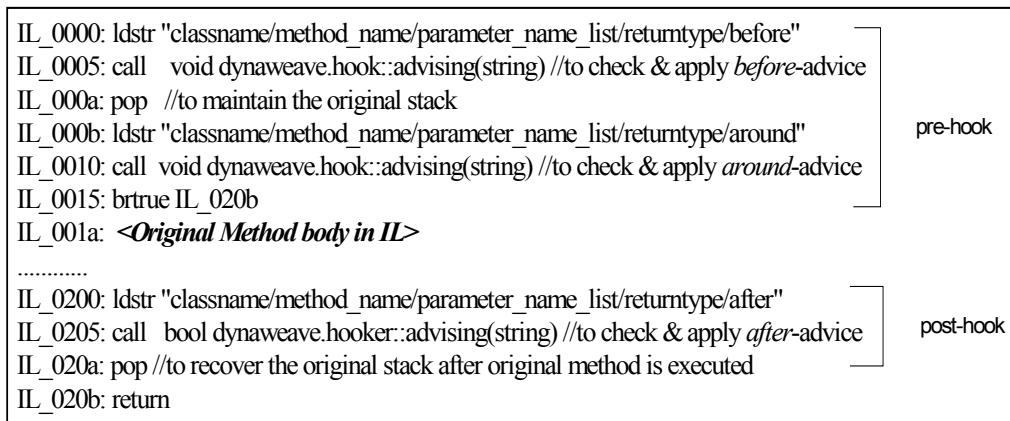


Figure 2: Instrumentation for the WS Method in IL

capacity can be used to instrument codes for addressing non-functional concerns, such as applying access control upon the entry into the WS method, or applying state persistency service for the executed WS application upon the end of the WS call. Also included in the pre-hook are the instructions to check if an *around advice* is specified or not, and a jump instruction to redirect the execution to the exit point of the WS application. The jump instruction is to be activated if an around advice is found valid in the AAR. With around advice, the original WS will be replaced with new behavior specified in that around advice. Consequently, not only the original WS can be decorated, it can also be overridden completely, which is necessary when a buggy WS is identified and need to be removed, or an old service module need to be updated. By using hooks for weaving, advice can be applied dynamically and proactively. Meanwhile, unweaving advice can be realized by dis-activation of the corresponding AUS in AAR. Figure 2 is the CIL manipulation template for adapting a WS method.

3.2.2 Specifying WS Adaptation

The type system used in the AUS in AAR can be based on the object-oriented Common Type System (CTS) [6] of CIL, for which each CLR hosted language is translated to before being JIT compiled. Therefore, such specification is applicable to all WS applications running in CLR, which provides a language-neutral approach for AUS. However, writing adaptation AUS based on low level CTS is error-prone and not necessary for high-level AUS. Fortunately, there is another language neutral type system at a high level that Web Service Description Language (WSDL) is based on, the XML Schema¹, that can be used to

specify the adaptation advice. The AUS in AAR accords with XML Schema as illustrated in Figure 3.

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="tns:advice_type"/>
  <wsdl:input message="tns:return_type"/>
  <wsdl:input message="tns:classname"/>
  <wsdl:input message="tns:methodname"/>
  <wsdl:input message="tns:parameter_list"/>
  <wsdl:input message="tns:advice_name"/>
</wsdl:operation>
```

Figure 3: WS adaptation AUS in WSDL

The specification is represented as WSDL so that the configuration can also be exported as WS, which enables adaptation advice to be specified remotely. Our work has only been applied to the local configuration of adaptation advice at present. The XML Schema based specification is parsed and translated to CTS to be matched against the string provided by the hook such as described in IL_0000, IL_000b, IL_0200 in Figure 2.

Associated with each *advice_name* is the path information for actual advice in the form of managed code stored in the AAR. All the advice code is defined as a template with the tuple <Classname, Methodname, Parameter_List> as parameters, which offers reusability of advices. Such advice can be pre-built in any .NET language and compiled into managed code. If a matching advice is found, then the advice code will be loaded from the corresponding path and called. In our work, the wild-card characters are also supported for AUS.

¹ <http://www.w3c.org/2001/XMLSchema>

4. Example

Figure 4 provides an adaptation example for a college student credit authorization WS to demonstrate the use of this dynamic WS provisioning framework for a non-functional concern: access control. Figure 4-A provides a simple WS application written in C#, which provides a WS method for authorizing credit card application based on the Social Security Number (SSN) and the expected credit line. The corresponding WSDL in Figure 4-B can be automatically generated from the source code in Figure 4-A based on existent tool support, which in turn is to be exported and used as the basis for AUS as well. Figure 4-C is an AUS with an around advice to apply credit history checking before any credit card application request is processed. The wild card specification in `credit_*` represents all credit application with the request name preceded with "credit_". Figure 4-D is the source code for the pre-built credit history checking advice, which can be written in any .NET language (here C#) and is compiled and persisted in the managed code form. The type systems in Figure 4-A, Figure 4-C, Figure 4-D are being translated into CIL and matched up in CLR. Once a match holds, the advice in Figure 4-D will be called by the hook instrumented at runtime. This shows the form of cross-language capacity. The WS application source detail is transparent to AUS in Figure 4-C, and advice definition advice in Figure 4-D.

5. Performance Evaluation

Using the profiler to handle all the events generated from all managed execution in CLR is expensive and will degrade system performance significantly. Therefore, we apply optimization at three levels through configuring the profiler as indicated in (7) in Figure 1:

- 1). As the CLR can be launched from a shell, Internet Explorer, ASP.NET, and other customizable CLR hosts for managed execution, we configure the profiler to skip profiling for all non-ASP.NET modules hosted in CLR, which can be filtered easily based on the name of the module that launches the CLR.
- 2). We further trim off unnecessary profiling based on class name, method name based on CTS-based CIL. This is possible because all managed code is translated to CI, and the CIL level information can be derived from the corresponding WSDL for the WS; this is also necessary to avoid profiling system classes and methods.
- 3). We mask off all unnecessary events except JIT compilation events, which is needed for handling CIL manipulation.

To evaluate the influence of CLR profiling-based WS adaptation on performance, we implemented a simple WS server application with 100 loops for calling a method, which contains only a single plus calculation in its body. We host this WS application on a Dell Workstation with Intel XEON CPU 2.2GHz, 1.00GB RAM, which is installed with Win XP professional version 2002 with IIS 5.1, .NET framework version 1.1.4322. We configure profiler so that the method is to be profiled and adapted with a log advice to write to a file a line of string. A WS stub is generated by compiling the corresponding WSDL for this simple WS application. The WS stub is instrumented together with a simple client application for the client application to call the server-side WS. The client side is hosted on a Dell PC with Intel Pentium 4 CPU 1.80 GHz, 512 MB RAM which resides on the same LAN environment as the server so as to minimize the network influence during the server side performance benchmarking.

Note that the CLR profiling-based approach only applies to managed code to be loaded and JIT compiled. Therefore, we run ASP.NET in the managed mode for profiling WS to realize dynamic adaptation. ASP.NET can load one worker process to handle a pool of WS requests. Once the worker process is launched to serve the first WS request in the pool, it continues to serve other WS requests in the same pool until the end of its lifecycle without itself being reloaded into CLR, thus it fails to profile the other WS applications in the same pool. Therefore, we adjust the setting for ASP.NET so that a new worker process will be created for each WS request so that each WS call can be captured by the Profiler and thus is adaptable. The goal of our tests is to evaluate how the adjustment of worker process lifetime (Figure 5-a), and the adoption of profiling-based dynamic adaptation (Figure 5-b) affect the performance WS provisioning.

For the case in Figure 5-a, we did not provide any adaptation advice when adjusting the worker process life between *zero life* (a new worker process is created for each WS request) and *infinite life* (the same worker class is used for multiple WS requests). The absence of advice execution will help clarify the influence of the changing life of worker processes on the system performance.

There are significant differences between the first call and the remaining calls for an infinite life case as the first call involves the creation of a new worker class, thus incurring more overhead than the remaining WS calls which use the original worker process. Also the presence of profiling does not affect performance much in the case of infinite life, as the worker process is no longer to be reloaded for new WS requests, the new WS will not be adapted, and the event handler in the profiling API is ignored. In comparison, the worker

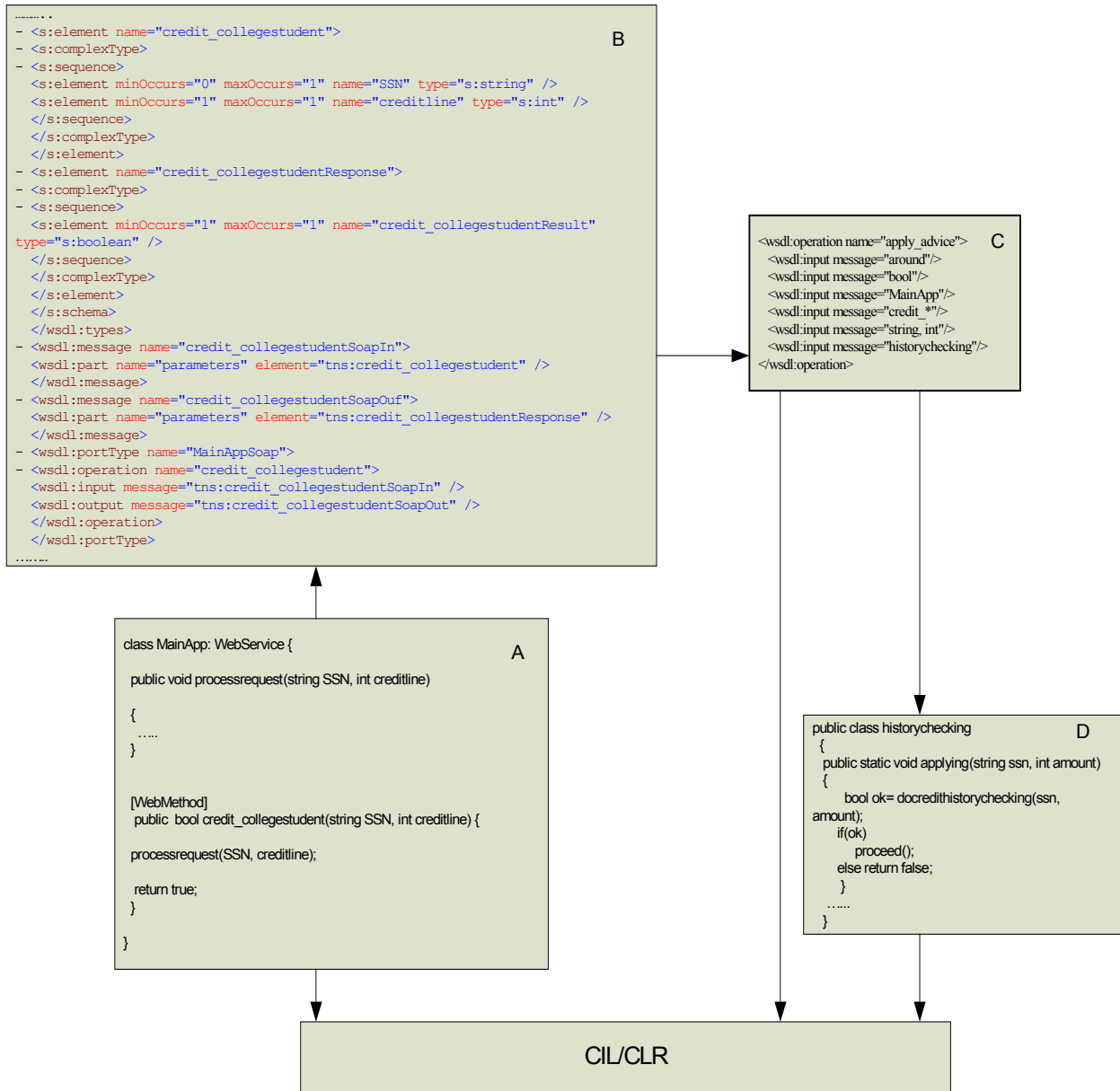


Figure 4: Adaptation of credit authorization WS

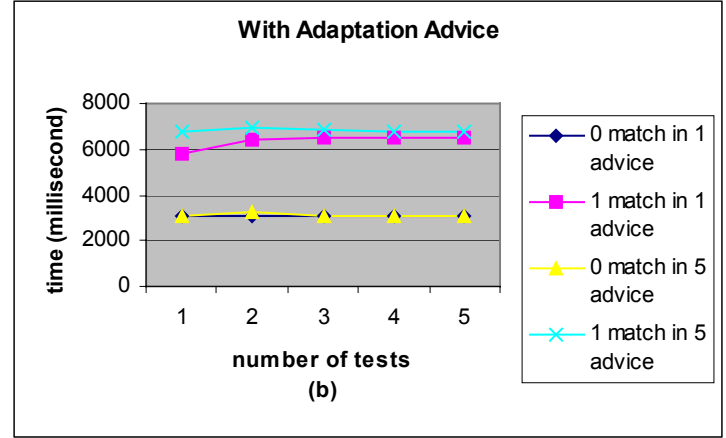
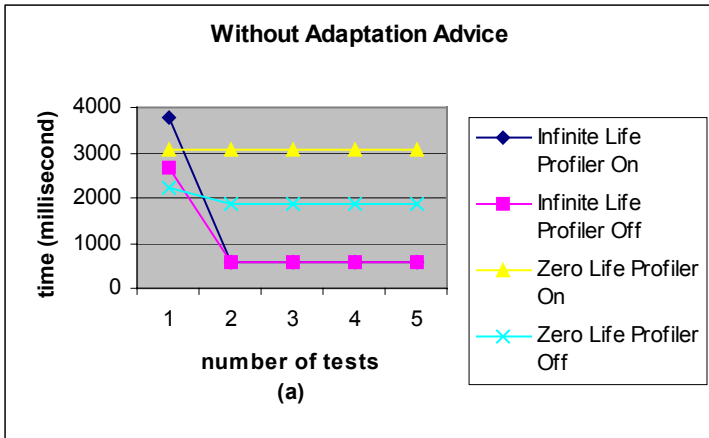


Figure 5: Benchmarking dynamic WS adaptation

process with zero life will incur a performance degradation of 1.7 times slower with profiling on than with profiling off. With the absence of the profiler, the overhead incurred by adjusting from infinite life to zero life will be 3.0 times. With the absence of advice, the overall performance degrade (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class) for this WS provisioning is $3.0 \times 1.7 = 5.1$. Figure 6 illustrates the performance degradation.

In Figure 5-b, we focus on evaluating the influence of active advice on the overall performance. Therefore, the worker process is set with zero life. We found the number of active advice will not affect the performance linearly, as the AUS are stored in a paging file to be shared by hooks, which constitutes a minor overhead in comparison to that incurred by hook instrumentation and calling of advice. The weaving of a matching advice in the case of zero life in Figure 5-b incurs a performance degrade of 2.2 times. Therefore, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class), by synthesizing the result described in the preceding paragraph, will be $2.2 \times 5.1 = 11.2$.

In a real world deployment, we can reduce the overhead by setting the worker class to zero life at the adaptation time, then resetting it to infinite time after adaptation is done. Of course this assumes a predictable adaptation process.

6. Related Work

Our work on dynamic service adaptation is based on runtime intermediate code manipulation by intercepting the JIT compilation event. The interception-based

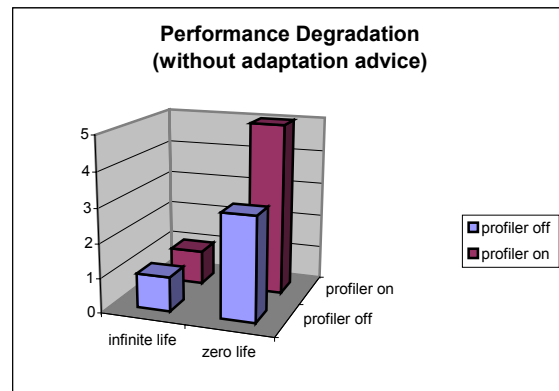


Figure 6: Performance degradation with no adaptation advice

approach is not new. Java servlets provide a handle of `HttpServletRequest` to intercept remote request, and CORBA provides Portable Interceptor. In particular, the QuO project [3] leverages system condition objects (sysconds) to probe CORBA system information and then apply Quality of Service (QoS) aspects. None of the aforementioned work is based on binary code manipulation, and adaptations are incorporated into the application code in an invasive fashion. Also the adaptation can only be used to decorate rather than to override original application. Besides non-.NET platform interception-based work, there are also some .NET interception based work using profiling API such as CLAW [8], which is a cross-language aspect weaver. However, in contrast to our instrumentation of hook into base applications at JIT time, CLAW generates a dynamic proxy at runtime, which lacks the flexibility of adjusting advice weaving decision proactively and retroactively. Moreover, our work is the first to offer a solution in the middleware context based on CIL code manipulation.

Our work on **dynamic WS provisioning** also nicely complements the existent work on **dynamic WS consumption** [11] and **dynamic WS orchestration** [1]. All three of these apply the AOP principle for modularizing dynamic adaptation. Specifically, in [11], a Web Services Management Layer (WSML) is introduced using dynamic AOP implementation language JAsCo to enable hot-swapping and runtime management of services. WSML is a Java-based, client-side software layer for consumption of WS, while our work is for server-side dynamic WS provisioning. Also, even though the server side is .NET platform based, there is no language and platform restrictions for the client side. In [1], dynamic service orchestration is realized with AO4BPEL, an aspect-oriented extension to BPEL4WS², which is a static WS composition model. Though our work is not intended for service orchestration, the adaptation advice can also be implemented as a Façade [4] for service composition dynamically.

7. Conclusion and Future Work

This paper presents a novel approach for dynamic WS provisioning based on modularized adaptation, which has two non-invasive properties: 1) By manipulating applications at the in-memory intermediate code level, it is non-invasive to WS applications, and also has no need for binary code transformation before runtime; 2) By using a pluggable profiler, it is non-invasive to the platform hosting the WS application. Moreover, not only the adaptation can be dynamically applied, it can be dynamically configured as well. Among other distinguished features are advice reuse, advice unweaving, and language neutral weaving specification based on an XML schema. The experimental result shows the profiling-based approach for WS adaptation is encouraging because of the easy control over the profiling scope in the scenario of WS provisioning. Even though the approach presented in this paper is .NET based, the principle also applies to other platforms with adequate software vendor support.

Future work will include the autonomic configuration of the profiler and AAR to make WS provisioning not only dynamic, but also autonomic. Another direction will be to evolve the advice library for reuse, and to explore how AAR can be incorporated into UDDI.

8. References

- [1] Charfi, A., Mezini, M., "Aspect-Oriented Web Service Composition with AO4BPEL," *Proc. of the European Conference on Web Services 2004 (ECOWS 2004)*, September, 2004, pp. 168-182.
- [2] Dan, A., Franck, A. R., Keller, A., King, R., Ludwig, H., "Web Service Level Agreement (WSLA) Language Specification 2002," <http://dwdemos.alphaworks.ibm.com/wstk/common/wstkdoc/services/utilities/wslaauthoring/WebServiceLevelAgreementLanguage.html>.
- [3] Duzan, G., Loyall, J. P., Schantz, R. E., Shapiro R., and Zinky, J. A., "Building Adaptive Distributed Applications with Middleware and Aspects," *Proc. 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, March, 2004, pp. 66-73.
- [4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [5] Newcomer, E., *Understanding Web Services*, Addison Wesley, 2002.
- [6] Gough, J., *Compiling for the .NET Common Language Runtime (CLR)*, Prentice Hall PTR, 2002.
- [7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., "Aspect-Oriented Programming," *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, June, 1997, pp. 220-242.
- [8] Lam, J., "Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime," Demo at 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), April, 2002.
- [9] Sahai, A., Machiraju, V., Sayal, M., Jin, L. J., Casati, F., "Automated SLA Monitoring for Web Services," <http://www.hpl.hp.com/techreports/2002/HPL-2002-191.pdf>
- [10] Stutz, D., Neward T., Shilling, G., *Shared Source CLI – Essentials*, O'Reilly Press, 2003.
- [11] Verheecke, B., Cibrán M. A., Vanderperren, W., Suvéé D., and Jonckers, V., "AOP for Dynamic Configuration and Management of Web services," *International Journal on Web Services Research (JWSR)*, 2004, pp 25-41.

² BPEL4WS - Business Process Execution Language for Web Services - <http://www-128.ibm.com/developerworks/library/specification/ws-bpel>