Columbia University Department of Computer Science Tech Report CUCS-015-05

Multi-Language Edit-and-Continue for the Masses

Marc Eaddy Steven Feiner
Department of Computer Science
Columbia University
New York, NY 10027
+1-212-939-7000

{eaddy,feiner}@cs.columbia.edu

April 4, 2005

ABSTRACT

We present an Edit-and-Continue implementation that allows regular source files to be treated like interactively updatable, compiled scripts, coupling the speed of compiled native machine code, with the ability to make changes without restarting. Our implementation is based on the Microsoft .NET Framework and allows applications written in any .NET language to be dynamically updatable. Our solution works with the standard version of the Microsoft Common Language Runtime, and does not require a custom compiler or runtime. Because no application changes are needed, it is transparent to the application developer. The runtime overhead of our implementation is low enough to support updating real-time applications (e.g., interactive 3D graphics applications).

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Programming Environments—*Interactive environments*; Testing and Debugging—*Debugging aids*; D.1 [**Software**]: Programming Techniques—Dynamic Software Updating; D.3.3 [**Programming Languages**]: Processors—*Incremental compilers, dynamic compilers*

General Terms

Experimentation, Languages, Performance

Keywords

dynamic software updating, online reconfiguration, patching, edit-and-continue, hot-swapping, rapid application development, .NET

1. INTRODUCTION

Edit-and-Continue is a common debugging feature found in many integrated development environments (IDEs). It refers to the ability to pause an application, usually via a breakpoint, make changes to the source code that are recompiled in the background, and then continue execution with the modified application. These activities must be performed in the IDE because they are usually not supported by the application's runtime environment. Because using Edit-and-Continue in the IDE can be cumbersome, it is usually reserved for debugging activities. We desired a lightweight and efficient Edit-and-

Continue solution for rapid prototyping, extensibility, and dynamic reconfiguration that did not require the presence of an IDE, and would therefore be available to end-user and developer alike.

Our requirements for Edit-and-Continue were motivated by our target application class: *real-time 3D graphics applications*. Specifically, we are addressing interactive 3D user interfaces, including virtual reality (VR), augmented reality, and 3D games. These applications support real-time 3D rendering and require high-performance, highly configurable software architectures.

1.1 Our Solution

Our Edit-and-Continue allows any .NET language source file to be treated as a *high-performance, dynamic script*. It is fast because source code changes are compiled to native machine code instead of being interpreted. However, the user does not execute an explicit compilation step because compilation occurs in the background.

We have successfully tested Edit-and-Continue on applications written in C#, VB.NET, and JScript.NET. Users can even intermingle multiple languages in their application, although we only support dynamic updates to languages that support the .NET CodeDOM API (specifically, the ICodeCompiler API).

Unlike approaches that require that code be run within the IDE, our implementation is library-based and requires only the standard Microsoft Common Language Runtime. Therefore, it can be used by end-users who do not have an IDE, as well as by developers.

Our Edit-and-Continue implementation allows code changes (currently, only updates to method bodies) to be applied without having to restart the application. This shortens the normal edit–compile–restart cycle and enables interactive prototyping and debugging. However, if the compilation fails due to a syntax error, type violation, misspelling, or other compilation error, the change will be rejected.

In addition, our Edit-and-Continue solution has the following properties:

Transparent. The application developer does not have to make any code changes to support Edit-and-Continue. However, they do have to specify a compiler flag and three environment variables. For some .NET languages, the user must also provide a *build description file*, similar to a makefile.

Lightweight. Our solution is based on standard .NET application programming interfaces (APIs) and does not require an IDE. Code changes can be made using the user's favorite editor (e.g., Notepad, emacs, or *vi*).

Automatic. Changes are applied as soon as they are saved by the user. No other action is necessary. To make this possible, we developed a difference algorithm for determining changes to a method's byte code automatically (see Section 3). Changes can also be made remotely, which is especially useful for full-screen 3D applications. If multiple instances of an application are running in a distributed scenario, they will all be updated at (roughly) the same time.

Low overhead. Our solution is enabled using the low overhead .NET Profiling API. Asynchronous compilation and code patching are performed in the background. The overhead is low enough to be feasible for the performance-critical applications that we are targeting, which include 3D games. We evaluate the performance of our implementation in Section 4.

Our Edit-and-Continue solution is the first to support multiple languages in a generic way since it supports any .NET language that implements the standard .NET CodeDOM API. In contrast, other solutions support only one language or a pre-defined number of languages. Furthermore, some languages (e.g., JScript.NET, Eiffel, Cobol, Perl, Python, Lua, Haskell, to name a few) do not currently support Edit-and-Continue. Because our solution is based on the generic CodeDOM API, we can potentially support any language that implements the API (specifically, the ICodeCompiler interface). For example, ours is the only solution that supports Edit-and-Continue for JScript.NET.

Our solution supports certain types of unanticipated changes (see Section 5.2) to .NET applications that would be difficult or impossible to support using other techniques. Moreover, changes can be made interactively without requiring a restart of the application. We find Edit-and-Continue to be a very useful prototyping technique for our 3D applications.

2. Related Work

Prior to arriving at our Edit-and-Continue solution, we investigated other popular application-level techniques for enabling rapid application development, including *dynamic configuration files* and *embedded scripting languages*, which we describe below. We then compare our Edit-and-Continue solution with other *software updating* approaches.

2.1 Dynamic Configuration Files

In the VR Juggler [2] architecture for building VR applications, changing a configuration file causes the application to reinitialize automatically without requiring a restart. This can result in a substantial time savings for application developers, as the startup time required for VR applications can be quite long, due to loading large models, initializing devices, and communicating with servers. Other benefits of this approach are quick prototyping of new features and interactive performance tuning. The limitation is that dynamic configuration files are an inflexible application-specific solution because the application behaviors that can be configured are typically very limited and must be anticipated beforehand.

2.2 Embedded Scripting Languages

Another common approach is to embed a script language interpreter or *script engine* inside the application and then graft on a scripting or command language interface. Tcl [23] is one of the best known examples. Other popular choices are Perl, Python, JavaScript, and Lua. Script-based interfaces, intended for the *developer-as-end-user* audience, are widely recognized for supporting rapid prototyping. However, there are many disadvantages to scripting, especially for high-performance 3D applications, including the following:

Slow performance. Script languages are by-and-large interpreted, although what is being interpreted may be an intermediate representation or byte code. Regardless, interpreted code is much slower than compiled machine code [3, 12]. Just-in-time compilers (e.g., [25]) help alleviate this problem, but do not eliminate it entirely. While the slower speed may be adequate for some applications, it becomes a bottleneck for high-performance ones. Therefore, only a portion of application behavior is typically scripted (i.e., the *script interface*) and the rest of the application is written in a compiled language [22].

Object model discontinuity. The objects, variables, and functions of the compiled portion of the application and those of the script form two separate object models, written in two different languages. The script interface is often implemented using a *reflection API* or some other "glue code." *Reflection* refers to the ability to programmatically obtain type information or call methods at run-time. It can be used to provide a bridge that allows communication between two languages or object models.

Providing a one-to-one bridge for every object and function in the object model is infeasible since it can require a major development effort and entails a significant runtime performance penalty [24]. This requires the application developer to tailor the script interface to meet their anticipated application needs, further limiting the power of the scripting solution and making it application-specific.

Unlike a scripting solution, whose expressive power is limited by the script interface, Edit-and-Continue allows arbitrary, and possibly unanticipated, type-safe changes to method bodies. Moreover, object-oriented language features such as class and member access control, reuse, and data hiding can be leveraged for free.

Multi-language issues. Different languages usually have separate and sometimes conflicting threading, networking, GUI and memory management models. In addition, few tools allow simultaneous cross-language debugging [4, 22].

Our approach avoids object model and language discontinuity by leveraging features common to all .NET languages. Users can intermingle multiple languages in their application, although we only support dynamic updates to languages that support the .NET CodeDOM API (specifically, the ICodeCompiler API).

Need for a restart. Aside from some application-specific solutions, script changes do not take effect until the application is restarted, preventing interactive prototyping and debugging. This is particularly painful for applications with long startup times, common in VR and gaming, or when trying to reproduce a bug.

It should be noted that several scripting languages, including Perl, Python, JavaScript, and Lua, have been ported to .NET. All .NET languages share a unified type system (the .NET Common Type System), and use the same class library (the .NET Framework Class Library). This means that integration of a .NET-based scripting language will not suffer from object model or language discontinuity and will not incur cross-language communication overhead. However, none of these languages allow the user to make updates to a running application. Ideally, these languages would expose a compiler interface via the CodeDOM API, which would enable us to apply our Edit-and-Continue solution to applications written in Python.NET, for example.

2.3 Dynamic Software Updating

Dynamic software updating (also known as *online reconfiguration*, *hot-swapping*, and *code patching*) refers to the ability to update currently executing software, at the machine code or byte code level. *Edit-and-Continue* (also known as *fix-and-continue*, *develop-and-continue*, and *incremental build*, as well as by many other names) is a special form of dynamic software updating in which updates (1) are specified by directly modifying the original source files, (2) are compiled in the background, and (3) are applied immediately.

Our solution is library-based, as opposed to language-, IDE-, runtime-, or proxy-based. We will explain the tradeoffs of each technique below. As we are focused on general-purpose Edit-and-Continue solutions, we do not consider the large number of solutions that are application-specific.

2.3.1 Language-Level Support

Ideally, the language execution environment itself will natively support Edit-and-Continue, as do Lisp [15], Prolog, Forth[1] and Smalltalk [8]. However, we do not consider these languages viable for the development of 3D applications, for reasons of performance, lack of support for 3D graphics APIs, and the difficulty of finding developers proficient in these languages.

2.3.2 Integrated Development Environments

Most modern IDEs support Edit-and-Continue, although the details about *how* they do it are not in the public domain:

- Microsoft Visual Studio™ (C/C++ [29], VB, and C# [31])
- Sun™ Java Studio[2] (Java)
- SGI ProDev™ WorkShop[3] (C/C++, Fortran, Ada)
- HP wdb[4] (C/C++, Fortran)
- Eclipse[5] (Edit-and-Continue is only supported for Java)
- Apple Xcode[6] (C/C++, Objective C, Java)
- Functional Developer[7] (Dylan)

Most of these IDEs provide more Edit-and-Continue functionality than our solution, such as the ability to make certain type-safe design changes, such as adding new functions and fields [29, 31]. The main benefits of our solution are that it supports any .NET language, for example, ours is the only solution that supports Edit-and-Continue for JScript.NET, is completely library-based, and supports remote updates.

A library-based solution is essential in a setting in which the IDE is not installed (e.g., a customer, demonstration, production, or developer-as-end-user setting), and in the *game mod* community [4, 22]. The user can take advantage of Edit-and-Continue without rebuilding the executable explicitly. Even when the IDE is available, its Edit-and-Continue feature may not be convenient because it requires that the IDE be running and "attached" to the target process, and that changes be made from within the IDE editor. If the IDE is not running, the user will need to start it and attach to the process, which may take several

[1] http://www.forth.org
[2] http://developers.sun.com/prodtech/javatools/jsstandard
[3] http://www.sgi.com/products/software/irix/tools/prodev.html
[4] http://h21007.www2.hp.com/dspp/tech/tech_TechSoftware DetailPage_IDX/1,1703,1662,00.html
[5] http://www.eclipse.org
[6] http://www.apple.com/macosx/features/xcode
[7] http://www.functionaldeveloper.com

seconds. In contrast, our Edit-and-Continue solution is *always on*. Changes can be made in any editor and the effects are visible at interactive (sub-second) speeds.

Finally, since our user interface is essentially the file system, our solution allows updates to be applied remotely, which is not possible using an IDE. This is essential for applications that must use the full screen or that capture all mouse and keyboard input, including VR applications and 3D games.

2.3.3 Runtime-based Approaches

Dynamic software updating systems usually have loftier goals than supporting the simple updates required for Edit-and-Continue. These systems are designed to update entire classes or modules. Supporting this scale of updating usually places constraints on the design that render the system unsuitable for Edit-and-Continue. For example, many systems require a custom class loader [1, 28], compiler [10], operating system [27], or runtime [17], and thus are non-portable. Other systems support only research languages [10], suffer from performance problems [1, 11, 16], or are not transparent [7, 11, 26].

Although it appears that many dynamic software updating systems could meet the interactivity requirements of Edit-and-Continue, we found no mention of this in the literature. We also found no system that supports updating applications written in multiple languages.

2.3.4 Proxy-based Approaches

We initially considered (and ultimately rejected) implementing Edit-and-Continue using .NET Remoting Proxies [16]. A major disadvantage to a proxy- or wrapper-based approach is the need for class renaming. This necessitates changing the client code to use a special class factory [26, 28] or derive from a special interface [11, 16], or requires a way to convert client code to use the proxy [17]. Consequently, the identity of the classes is not preserved, which introduces subtle bugs in client code that uses reflection, serialization, casting, or run-time type identification.

Another issue is that state transfer must be used when replacing object instances to guarantee consistency across updates [27]. This is difficult to do in the general case, given that OS-level data structures (e.g., file handles and sockets) usually cannot be transferred between processes [10]. This also requires the proxy to manage its own object instances to update them, requiring extra data and code [11, 17].

Furthermore, proxy-based approaches require at least one, and sometimes several, extra indirections per function call, a prohibitive cost for performance-critical applications [11, 17]. In addition, many solutions require the class being updated to enter a state of *quiescence*, in which none of its methods are currently active [27]. This can prevent some updates from being applied [17], or require that all threads be suspended or all method calls into the class be blocked during the update [10]. This is not a desirable solution for us, since pauses can adversely affect the execution of a VR application; for example, disorienting the user or causing the application to miss network or device updates.

2.3.5 Library-based Approaches

Our Edit-and-Continue solution makes extensive use of the .NET Profiler API, which provides traditional profiling services as well as the ability to replace method bodies [18, 20]. CLAW [14] and AOP-Engine [6] take advantage of this feature to implement aspect-oriented programming. However, ours is the first system to use the method replacement facilities of the Profiler API to implement Edit-and-Continue.

The architecture of the AOP-Engine [6] most closely resembles our system. Frei and colleagues mention how method replacement could be used to apply bug-fix patches to running applications; however, their solution is currently not interactive, which prevents it from being used in an Edit-and-Continue fashion.

The Java HotSwap API provides a method replacement facility similar to that of the .NET Profiler API. However, HotSwap is designed for replacing entire classes, which requires more work and resources than replacing just a single method. There are plans to resolve this issue in a later version of the API [5]. The HotSwap Client Tool[8] demonstrates how Java classes can be replaced at runtime. However, it does not meet our definition of Edit-and-Continue, as it does not perform background compilation and automatic method replacement.

3. Implementation

The key enabling technologies needed for our Edit-and-Continue system are the .NET CodeDOM and Profiler APIs and the *build description file*. The CodeDOM API provides an interface for compiling source code. The Profiler API allows a developer to "hook" specific application events (e.g., module loading/unloading, memory allocation/deallocation, function entry/exit, and just-in-time (JIT) compilation events). These events are of obvious interest to a CPU or memory profiler. The build description file lists all source files and compiler options used when compiling the original application, allowing us to use the CodeDOM API to reproduce the compilation exactly.

3.1 Overview

Our system is composed of two subsystems. The *Patcher* subsystem is a *dynamic link library* (DLL) written in standard C++. When the application is started, Patcher determines where to find its source files and how to build them. When a module is updated, Patcher determines which methods have changed and updates them.

The *Watcher* subsystem is a separate executable (EXE) written in C#. It is responsible for detecting changes to the source files and recompiling them on-the-fly.

3.2 Background: CLR Technology

To explain our implementation, it is important to first review some fundamentals of the Common Language Runtime.

A NET application consists of modules built by a .NET compiler that compiles source files into Microsoft Intermediate Language (MSIL) *byte code*. This is termed *source compilation*. To avoid the cost of interpreting byte code, right before a function is exe-

[8] http://developers.sun.com/dev/coolstuff/hotswap/more.html

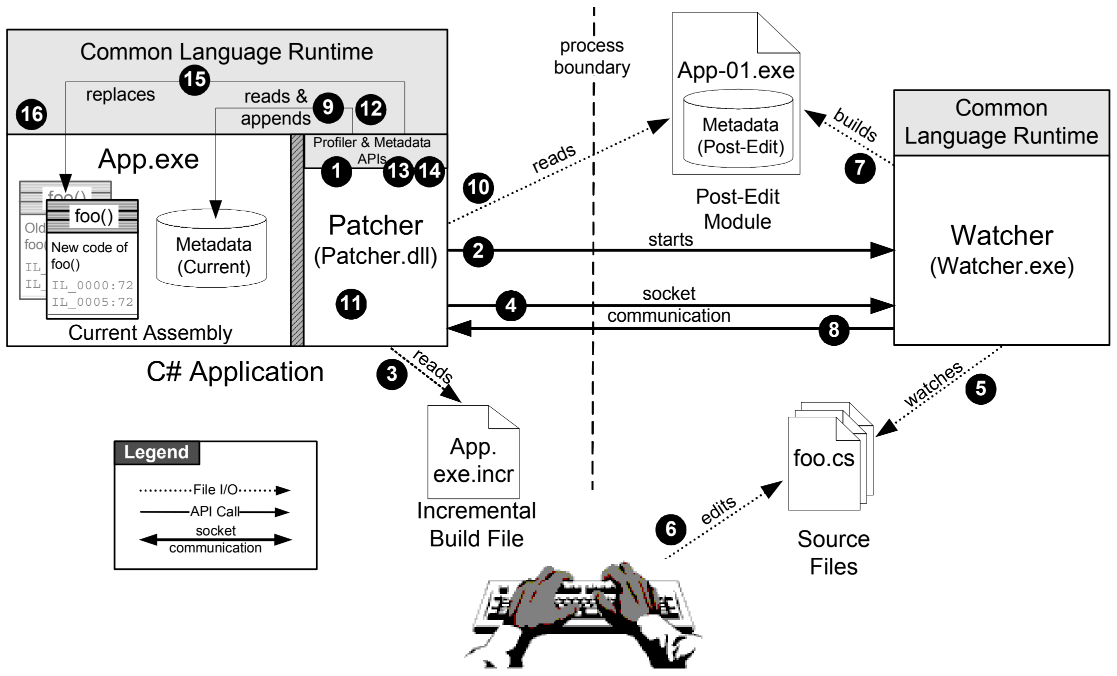

Figure 1. Edit-and-Continue walkthrough for an application written in C#.

cuted it is compiled to native machine code via a process called *just-in-time (JIT) compilation*.

Languages that compile to byte code are called *managed* languages because the execution environment and memory management is handled by the Common Language Runtime. All other languages are considered *unmanaged*.

A *module* is a binary file, usually a DLL or EXE, which contains the byte code for the methods and the metadata for the data types implemented by the module. Modules are analogous to Java class files.

Metadata is binary data that describes an application and is stored either in an executable file or in memory. Metadata describes every data type used by the application, as well as other characteristics of the application. Metadata is stored in *metadata tables,* which are global structures shared by all methods. A change to a method body that affects a metadata table will likely affect other methods as well. Applications can use the Metadata API [19] to query the metadata.

A *token* is a handle to a metadata data table entry or memory "blob" that can refer to a string constant, data type, method, or some other metadata. Tokens are used in .NET instead of pointers so that the byte code is memory-model independent and to simplify garbage collection.

3.3 Edit-and-Continue Walkthrough

Figure 1 shows a walkthrough of our Edit-and-Continue scenario, with numbers indicating the interaction sequence, which we reference here. We assume that the user, prior to starting the application, has already installed our Edit-and-Continue files, built their application (they must also specify the /bugreport compile option), and set the profiler-related environment variables. Although the walkthrough is shown for an application written in C#, the walkthrough is exactly the same for all .NET languages.

When the application is started, the CLR loads Patcher.dll and calls *ModuleLoadFinished* for each module (1). Patcher spawns the Watcher (2), and then reads the build description file associated with each module to obtain the source files, referenced assemblies, and compiler options (3). Patcher sends the module compilation information to Watcher (4), which then starts watching the module's source files (5).

At some point in the future, the user edits the foo.cs file, changing the *foo* method, and saves their changes (6). Watcher receives a file change event for foo.cs and uses the module compilation information to perform a source compilation (source code to byte code) using the CodeDOM API. This creates the App-01.exe *post-edit module* (7). (The number is appended to the module name to ensure that it is unique.) If the compilation succeeds, Watcher sends a message to Patcher telling it the post-edit module path (8). Watcher then goes back to watching the source files for more changes.

Using the Metadata API, Patcher reads the *pre-edit* (current) metadata and method addresses associated with the running application (9). It also reads the post-edit metadata and method addresses from the post-edit module (10). It uses the *module diff* algorithm (described later) to compare the two modules (11). If one or more methods have changed, Patcher emits new metadata as necessary (12) and calls *SetFunctionReJIT* (13).

Sometime afterward, *foo* is called, and Patcher receives the *JITCompilationStarted* event from the CLR (14). Patcher then calls the *SetILMethodBody* Profiler function to replace the old byte code for *foo* with the new byte code (15). The CLR JIT compiles the new byte code to native machine code and executes *foo* (16).

3.4 Patcher

Using the Profiler API to hook an application is tricky. The Patcher DLL must be written in raw (unmanaged) C++, must not block inside an event handler, and is not allowed to make calls into managed code. The reason for the last restriction is given in the documentation [18] and widely echoed by other users: If the DLL is handling an event, and then subsequently calls into managed code, which in turn generates the same event, the CLR can deadlock because it is not designed to be reentrant. Note that these are only the restrictions placed on the Patcher DLL; the actual .NET application being hooked has no restrictions.

Because our Edit-and-Continue solution uses the managed CodeDOM API, we need a separate managed process to safely call the API. This is the motivation behind the Watcher process, which we describe later. (In the future, we would like to explore calling managed code from a separate thread, which may circumvent the deadlock issue.)

3.4.1 Startup

When the target application starts, the CLR immediately loads the Patcher DLL into the process and starts sending it application events. Patcher monitors the *ModuleLoadFinished* events to determine the file paths of the modules that compose the running application.

Patcher must determine all the information that Watcher needs to recompile the module, essentially recreating the command-line used to compile the original module. This includes the source files, the referenced assemblies, as well as other compilation options (e.g., /debug and /unsafe). See Table 1 for an example.

The Microsoft Visual C++ implementation of Edit-and-Continue stores this compilation information directly inside the C++ object files [29]. In contrast, Microsoft Visual C# and VB.NET stores the information in a separate file called the *build description file*. Patcher parses this file and sends the compilation information to Watcher.

Table 1. Compilation information example.

Module Name	*MyApp.exe*
Source Files	*MyApp.cs, MyHelper.cs*
Referenced Assemblies	*System.dll, mscorlib.dll*
Icon	*MyApp.ico*
Resource	*MyApp.res*
Defines	*MyLogging=1*
Main Class	*MyNamespace.MyClass*
Compile Options	*/optimized, /checked*

Later, the user modifies a source file. This causes Watcher to build the new module and to send the file path to Patcher. To avoid requesting that every method in the module be re-JITed, Patcher determines which methods have changed by using a two-pass *module diff* algorithm.

3.4.2 Module Diff

We implemented a "smart" *module diff* algorithm that determines when two methods are "equivalent." Method equivalence confounds a simple binary comparison because of the presence of tokens in the byte code. For example, if the user only changes a string literal from "foo" to "bar", the string token in the new module will be the same as the token in the old module, resulting in method bodies that are equivalent under binary comparison. Similarly, two bitwise identical tokens may refer to different string values.

Figure 2 illustrates this by showing an example of the metadata embedded in the hello.exe application before and after an update. Figure 2(a) shows the original hello.cs source file. Note that the string literals, "Hello" and ", world!", are stored in the User

```
namespace HelloNS {
   class Hello {
      static void Main() {
         Hello h = new Hello();
         h.Foo();
         h.Bar();
      }
      void Foo() {
         System.Console.Write("Hello");
      }
      void Bar() {
         System.Console.WriteLine(", world!");
      }
   }
}
```

User String Table

```
[70000001] "Hello"
[7000000D] ", world!"
```

Local Variable Signature Table

```
[11000001] Class HelloNS.Hello
```

a) hello.cs (original version)

```
namespace HelloNS {
   class Hello {
      static void Main() {
         Hello h = new Hello();
         h.Foo();
         h.Bar();
      }
      void Foo() {
         int i;
         System.Console.Write("Hi");
      }
      void Bar() {
         System.Console.WriteLine(", world!");
      }
   }
}
```

User String Table

```
[70000001] "Hi"
[70000007] ", world!"
```

Local Variable Signature Table

```
[11000001] Class HelloNS.Hello
[11000002] int
```

b) hello.cs (new version)

Figure 2. Hello.exe and metadata example. (a) The original hello.cs file. (b) The new hello.cs file after updating *Foo* by adding a local variable `i` and changing a string literal from "Hello" to "Hi". The arrows indicate the changes to the source code and the string and local variable metadata tables.

String metadata table. This is similar to how Java stores constants in a constant pool. The local variable `h` in the *Main* method has an associated type signature (`Class HelloNS.Hello`), which is stored in the Local Variable metadata table.

In Figure 2(b), the user has modified the method *Foo* by adding a local variable `i` (which is never used) and changing the string "Hello" to "Hi". What is interesting about this example is that the underlying byte code instructions for *Foo* are exactly the same. Only the metadata has changed. Detecting that the local variable signatures are different requires us to dereference the local variable signature token associated with the old and new method and perform a binary comparison for each signature.

Unlike local variables, detecting that a string token in the byte code refers to a different value is more involved. To detect this, we must iterate over the old and new method bodies, decoding and comparing each byte code instruction. This constitutes the first pass of our two-pass algorithm, and is quite tedious, as there are more than 250 different instruction codes to consider [20]. We perform a simple binary comparison until we encounter an instruction with a token argument. For example, the LDSTR instruction loads a string token onto the stack. We must dereference the old and new tokens to compare the metadata values to which they refer. Dmitriev [5] describes a similar approach in Java HotSwap.

Before replacing the method body of *Foo*, we must *emit* the new string and local variable signature metadata using the Metadata API [19]. Otherwise, we will get an invalid metadata exception that halts the application or the application will behave incorrectly (e.g., the wrong string value will be used). Figure 3 shows how the emitted metadata is appended to the existing metadata to form the merged metadata. This actually causes a one-time memory leak, since after the update, the application no longer refers to the original "Hello" string, yet it remains in the metadata. We consider this inconsequential because the leak is very small and it happens only once for each metadata value (i.e., it is not a persistent leak).

A further complication is that the string token for "Hi" in the final metadata shown in Figure 3 ("70000020") is different from the token in the post-edit metadata shown in Figure 2(b) ("70000001"). This occurs because the newly emitted metadata will have different tokens than the ones used in the new *Foo* method byte code. This requires Patcher to "fix-up" tokens in the new byte code to match the new tokens. To do this, Patcher performs a second pass over the method body, using the same technique described earlier, this time to correct the tokens.

User String Table

```
[70000001] "Hello"
[7000000D] ", world!"
[70000020] "Hi"
```

Local Variable Signature Table

```
[11000001] Class HelloNS.Hello
[11000002] int
```

Figure 3. Merged metadata for the final hello.exe module.

Slightly more subtle is the fact that method *Bar* will *not* need to be updated. The user string table in Figure 2(b) shows that the token value for ", world!" was changed from "7000000D" in the original module to "70000007" in the new module. The reason has to do with how string tokens are generated, but for our purposes, the only thing that matters is that a binary comparison of the old and new method bodies will indicate that the methods are different.

We solve this problem in the same way we did for *Foo*, by dereferencing the string tokens to compare the old and new string values. In this case the string values are the same (i.e., ", world!"), so we do not need to update *Bar*.

Ours module diff algorithm uses a physical comparison as opposed to a logical comparison, which presumably would perform some type of static and/or dynamic analysis of the methods to determine equivalence. It is possible that our physical comparison will result in a false positive, where the byte codes are different but the methods perform exactly the same computation. However, no comparison technique can guarantee that two functions perform the same computation, as this is undecideable in the general case [9]. Since the cost of patching a function due to a false positive is low, as our performance metrics show, we argue that a more sophisticated comparison technique could actually degrade performance.

3.4.3 Method Replacing

After Patcher detects that a method has changed, emits the required metadata, and fixes up the byte code, it uses *SetFunctionReJIT* to schedule the function to be re-JITed. The next time the target method is called, Patcher's *JITCompilationStarted* event handler is called. Then, Patcher can replace the current method body's byte code with the new one.

3.5 Watcher

The Watcher process watches for changes to source files and recompiles them on-the-fly. We employ the .NET FileSystemWatcher and CodeDOM APIs for this purpose. After receiving the list of modules to watch, along with their corresponding compilation information, such as the list of source files, list of assembly references, and compile options, Watcher monitors the source files for changes (i.e., a changed modification time), using the event-based FileSystemWatcher API. Watcher maintains an MD5 hash of each source file and ignores situations when the user saves a file that has not changed. Malabarba and colleagues [17] provide a more reliable solution by computing a hash of each method's byte code which allows them to ignore situations when the source code has changed but the byte code remains the same.[9]

When a source file is changed, the associated module is recompiled using the CodeDOM API. As described earlier, Watcher then sends the new module path to the Patcher via a socket and Patcher updates the changed methods accordingly.

4. Evaluation

We have evaluated the performance of our Edit-and-Continue solution by quantifying the overhead and update latency using three benchmarks.

4.1 Experimental Setup

[9] Similar to Malabarba and colleagues[17], we consider the remote possibility that the hash will cause a false positive acceptable.

The experiment was run on a Dell Dimension 8100 Workstation with a single Pentium IV 1.3 GHz processor, a 400 MHz front-side bus, and 256MB of RAM. The platform was Windows XP with .NET Framework version 1.1.4322.

We used three separate C# benchmarks: SciMark,[10] CLI-Grande, and Text3D.

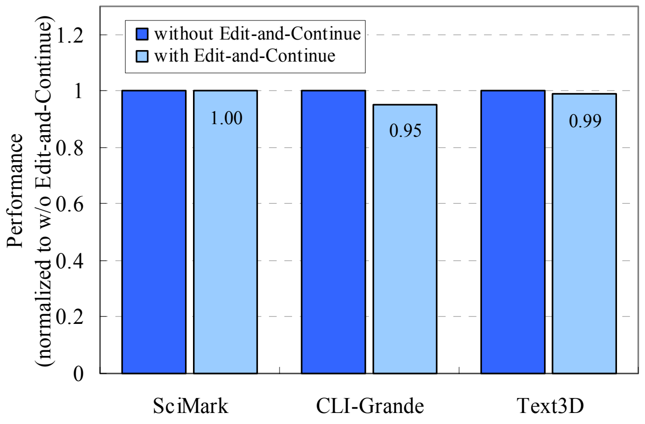

Figure 4. Performance with and without Edit-and-Continue.

SciMark (1,605 lines of code) is a benchmark for numerical and scientific computing. It includes Fast Fourier Transform, Jacobi Successive Over-Relaxation, Monte Carlo Integration, Sparse Matrix Multiply, and Dense LU Matrix Factorization.

CLI-Grande (4,834 lines of code) is a C# port of the Java-Grande benchmark. It exercises low-level functionality such as arithmetic, assignment, casting, arrays, object creation, loops, method calls, and serialization [30]. We modified the code slightly by reducing the number of iterations to shorten the trial time to around 12 minutes. This is not problematic, since we are comparing relative values, not absolute performance.

Text3D (14,731 lines of code) is not a benchmark per se. It is a simple Direct3D demo included with the Microsoft Managed DirectX 9 SDK. We included it to quantify the performance impact of Edit-and-Continue on 3D rendering speed. This benchmark holds special significance for us, since it represents the 3D application class we are actively targeting for Edit-and-Continue.

4.2 Runtime Overhead

Our system uses the Profiler API to selectively intercept module loading and JIT compilation application events. Our event handlers merely add work items to thread queues, so they are very fast and never block. All the real work is done in separate threads. However, simply enabling profiling implies an overhead. The overhead cost is mostly paid during startup, which is when the majority of module and JIT compilation events occur. The steady-state overhead is attributed to new execution paths that cause new methods to be JIT compiled.

Figure 4 shows our measurements of the runtime overhead for the three benchmarks with and without Edit-and-Continue. We performed five test runs for SciMark and Text3D, and three for CLI-Grande. SciMark and CLI-Grande ran to completion; however, Text3D is a windowed application, so we modified it slightly to exit automatically after three minutes. All executables under test, Patcher, Watcher, and the three benchmarks, were optimized release (retail) builds (compiled with switches /incremental /optimize /checked-). For each benchmark, the bar on the left shows the performance, normalized to one, of the benchmark without Edit-and-Continue enabled. The bar on the

[10] http://rotor.cs.cornell.edu/SciMark

Component	Milliseconds (StdDev)	Contribution
C# Compilation	723.00 (22.80)	95.51%
Patch Creation	32.00 (16.43)	4.23%
JIT Compilation	2.50 (5.00)	0.33%
Total Latency	**757.00 (30.05)**	**100.00%**

Table 2. Update latency for the *FFT.transform_internal* method. The average (arithemetic mean) time of each operation is listed in milliseconds along with its standard deviation.

right shows the performance with Edit-and-Continue enabled. As this test was designed to measure runtime overhead, no code patching was performed.

Our measurements show that Edit-and-Continue introduces a 1–5% runtime overhead, which we consider negligible. We had heard reports from outside sources that profiling incurred an overhead of around 10%, so we were pleasantly surprised with our results. As a comparison, the runtime overhead introduced by the dynamic updating system for Java developed by Malabarba and colleagues. [17], which uses a customized version of the Java Virtual Machine (JVM), is 6–10% slower than the original JVM.

We also measured the memory overhead to be static at around 4.5 MB.

4.3 Update Latency

We measured update latency using the SciMark benchmark. The *update latency* is the time it takes for an update to be applied. We also broke out the *update pause* portion of the update latency to distinguish the time during which the application must be paused to apply the update. This pause time is what is most noticeable to the user, especially when an update is applied to a continuously rendered 3D application.

4.3.1 Setup

For our test, we opened the FFT.cs file from the SciMark directory in Notepad and made updates to the *transform_internal* method. Our preliminary experiments showed that this function was called frequently during the early part of the SciMark benchmark. We modified the method after it had been JIT compiled (which happens the first time the function is called), but before the last time it is called. This ensured that the function would be called again, forcing another JIT compilation.

Because the SciMark application was built as an optimized release version, we were initially thwarted by our efforts to modify the method using only trivial changes, such as adding or updating a variable that is never used, since these were being “optimized away” by the compiler. We finally settled on the following C# code snippet, which we added to the method:

```
if (false) {
   Thread.Sleep(0);
}
```

Changing the condition from false to true (and vice versa) generates byte code that differs by one byte, which is enough to trigger Patcher to replace the method.

4.3.2 Update Latency

Table 2 shows that the average update latency is around 757 milliseconds. The table also shows that the background C# compilation time dominates the latency (95%). Although not shown in the table, C# compilations after the first one were slightly faster by around 100 ms, probably due to the JIT compilation that occurs for the CodeDOM module the first time it is used. We could avoid this first-time penalty by exercising the CodeDOM API beforehand, "priming the pump" as it were.

Patch creation took anywhere from 20–50 ms and JIT compilation took anywhere from 0–10 ms.[11] The C# compilation, patch creation, and JIT compilation times will increase with the number of methods and the size of the method bodies in the updated module.

4.3.3 Update Pause

All the work takes place in a separate process (C# compilation) or a separate thread (patch creation), except for JIT compilation, which takes place in the application's thread. This step forces the application to "pause" for a small amount of time (0–10 ms) while it waits for the JIT compilation to complete. Note that only the calling thread is affected. This is in contrast to systems that suspend all application threads when performing an update [17].

We obtained some empirical results by experimenting with making updates to the Text3D application, to observe the visual effect of the update overhead. After modifying a rendering method to increase the scale factor of a 3D object, we noticed a fraction of a second delay before the update took effect visually; however, the update was accomplished smoothly, without any appearance of "freezing."

4.3.4 Effects of Scaling on Update Latency

Only update latency is affected by the module (DLL/EXE) size. As the size of the module increases, the update latency will increase, as it is dominated by the module's compilation time. The largest module we tested was Text3D at 14,731 lines of code (includes blank lines and comments), which is representative of the sizes of the applications we are writing. As we reported in the previous sections, the update latency was negligible at under one second.

To support larger projects, we will need to look at traditional compilation time optimization techniques such as splitting the projects into smaller modules or performing incremental compilation, and pre-loading the CodeDOM libraries as mentioned earlier.

5. Limitations of Edit-and-Continue

Our Edit-and-Continue solution has some limitations, some of which are implementation-related and some which are problems common to the field of dynamic software updating.

5.1 Implementation Issues

Build description file. The build description file allows us to reproduce the compilation information exactly. This eliminates guesswork or the need to create build configuration files manually. We currently support the bug report file format (`.bugreport`) produced by the C# and VB.NET compilers when the /bugreport flag is used. Unfortunately, not all compilers produce a build description file. For example, we have to manually create the build description file for the JScript.NET compiler. This can potentially

[11] This range makes it appear that sometimes JIT compilation takes no time. However, our timer resolution is only 10 ms, which means measurements 10ms or less cannot be fully trusted (i.e., they range anywhere from 0 to 10ms).

lead to the build description file being out of sync with the actual build settings. We are currently looking at supporting the JScript project file format as well as using command-line utilities for automatically determining build settings.

Method inlining. Method inlining is a common and very effective compiler optimization. Unfortunately, the Profiler API does not allow inlined methods to be replaced. When Edit-and-Continue is enabled, we are forced to disable method inlining for the target process. This contributes to the performance overhead of our solution. In contrast, Java HotSwap supports updating inlined methods by first "de-inlining" them, while leaving other inlined methods alone [5].

Enabling Edit-and-Continue requires the user to set some profiler-related environment variables (`Cor_Enable_Profiling=1`, `COR_PROFILER=`*CLSID of Patcher*, and `WATCHER_PATH=`*Path to Watcher.exe*) and to compile the application with the `/bugreport` compiler switch. If the user forgets these steps, Edit-and-Continue will not be enabled. This is not obvious, since our Edit-and-Continue solution works entirely in the background. A useful feature would be to provide some type of visual indication of status, including errors (e.g., with a system tray icon). Hopefully, a future version of the .NET Common Language Runtime will support a more convenient interface for enabling profiling; for example, to allow the user to specify which applications to profile in the Windows Registry, or to attach a profiler to a running application.

5.2 Open Issues

We must address issues that are common to dynamic software updating systems, including what types of updates are allowed, when updates are applied, and how active functions are handled.

Types of updates allowed. By definition, the only changes an Edit-and-Continue implementation should support are those that can be automatically and quickly applied without extra help from the user. These include *implementation* changes (i.e., changes to method bodies), and *type-safe design* changes (e.g., adding fields and methods and removing/renaming *private* fields and methods). Supporting arbitrary design changes would complicate the user's workflow because special instructions, transformation or conversion rules are needed to ensure the system remains in a consistent state [10]. This defeats our goal of interactive development. This is the same position taken by other IDE-based Edit-and-Continue implementations, including `gdb` and Microsoft Visual C++ [29].

Currently, our solution only supports changes to method bodies (i.e., implementation changes). If the user wants to make design changes, they have to compile them outside of Edit-and-Continue. In the future, we would like to support type-safe design changes.

Update timing. We apply changes as soon as a file is updated (i.e., when the user saves the file). This overloads the save operation and may result in updates being applied unintentionally. In contrast, IDE-based solutions typically provide a separate way to apply code changes. The benefit is that our solution automatically applies updates, which is useful when applying updates remotely, and does not require a special editor. Another consideration is that the timing of the update may compromise program correctness. This is an open problem that has been shown to be undecideable [9].

Handling of active functions. An *active* function is a function that has a stack frame currently on the call stack. This includes the function that is currently executing on the top of the stack and those further down the stack. Some IDE-based Edit-and-Continue implementations, such as Microsoft Visual C++ [29], allow the user to modify the code of the currently executing function. For example, this is very helpful if the user needs to update the *main* function. This requires a more sophisticated method patch procedure, as well as adjusting the current instruction pointer. Similar to the Java HotSwap solution [5], our Edit-and-Continue solution cannot support updating the currently executing function because this is not supported by the Profiler API.

In our solution, the user is allowed to modify a function that is already on the stack, but the update is not actually applied until the function is called again (i.e., the existing stack frames are not updated). This is a consequence of the *SetFunctionReJIT* function. This can result in instances of both the old and new function existing on the stack at the same time. The Java HotSwap API provides a mechanism for updating stack frames; however, as mentioned earlier, we are not aware of an Edit-and-Continue implementation that uses it.

6. Conclusions and Future Work

Our Edit-and-Continue system allows the user to edit the source files of their running application and see the updates applied immediately. The system incurs a modest 1–5% runtime overhead and the update latency is less than one second. Moreover, the system runs on any Windows platform with the standard implementation of the Microsoft Common Language Runtime and does not require an IDE, thus making the solution more generally applicable.

Our system allows the user to treat source files as *high-performance, dynamic scripts*. This makes it particularly well-suited for dynamic reconfiguration, performance tuning, interactive development, and online debugging of high-performance and real-time applications.

Our initial experiences with using Edit-and-Continue on a large 3D application written in C# indicate that our solution is quite useful. For example, we used Edit-and-Continue to continually refine the field-of-view for a head-worn display and to tweak the matrix transformations for our head-tracking device. This would have been impossible using an IDE, because our application used the full screen and captured all mouse and keyboard input. We were able to tweak the settings by modifying the source files remotely using a separate machine.

By creating a library-based implementation of Edit-and-Continue, we have elevated it from a development-time only feature to a general application feature. Whereas in the past the application designer was limited to configuration files or scripting interfaces, now they can select situations in which Edit-and-Continue will provide a superior solution. It is especially useful for real-time applications, such as VR applications, and 3D games, that need powerful yet high-performance reconfiguration capabilities. However, Edit-and-Continue is a general application feature that can be transparently applied to any .NET application.

There are a number of areas that we are interested in addressing in future work, which we review here.

Source-level debugging. This will not work after a method update, because the debug information will be out of sync. As we believe source-level debugging is critical, we would like to find a way to support it.

Improved security. Dynamic software updating techniques can be used to circumvent security [21]. It would be very difficult ("security through obscurity"), but not impossible, to use our Edit-and-Continue system for this malicious purpose. We would like to study the security implications of our solution to determine what security improvements can be made (e.g., encrypting source files, namespace partitioning [17], or using the .NET Code Access Security API).

Aspect-Oriented Programming. We would like to leverage our ability to dynamically replace method bodies, even those that have already been JITed, to implement low-overhead dynamic aspect-oriented programming [6, 13, 14]. This would allow us to add monitoring and profiling support dynamically to a running application. We can support the ability to modify a function at runtime, while still ensuring that the function stays weaved properly (i.e., by reweaving whenever a method is replaced).

Arbitrary patching. We require an incremental build file and source files for Edit-and-Continue to work. Removing these restrictions, at the cost of a greatly increased security risk, would allow us to perform arbitrary patches of running methods, including system methods, to support applying security, performance, and bug fix patches [6, 28] and adaptive optimizations [27].

7. ACKNOWLEDGMENTS

We would like to thank Rean Griffith and Hrvoje Benko for reviewing this paper, Mark Lewin from Microsoft Research for his initial help, and John Lefor from Microsoft Research for his help on bug report files. This research is funded in part by a grant from Microsoft Research.

8. REFERENCES

[1] Ben-Shaul, I., Holder, O., and Lavva, B. Dynamic Adaptation and Deployment of Distributed Components in Hadas. *IEEE Trans. on Softw. Eng.*, 27(9): 769–787, Sept 2001.

[2] Bierbaum, A. and Cruz-Neira, C. Run-Time Reconfiguration in VR Juggler. *Proc. 4th Immersive Projection Tech. Workshop (IPT '00)* (Ames, Iowa, June), 2000.

[3] Cowell-Shah, C. W. Nine Language Performance Round-up: Benchmarking Math & File I/O. *OSNews Web Site*. 2004. http://www.osnews.com/story.php?news_id=5602.

[4] Dawson, B. Game Scripting with Python. *Proc. Game Dev. Conf. (GDC '02)* (San Jose, CA, March 21–23), 2002.

[5] Dmitriev, M. Application of the HotSwap Technology to Advanced Profiling. *Proc. 1st Int. Wkshp. on Unanticipated Softw. Evol. (USE '02)* (Malaga, Spain, June 10–14), 2002.

[6] Frei, A., Grawehr, P., and Alonso G. *A Dynamic AOP-Engine for .NET*. Tech. Rep. 445, Dept. of Comp. Sci., ETH Zürich, 2003.

[7] Gilmore, S., Kirli, D., and Walton, C. *Dynamic ML without Dynamic Types*. Tech. Rep. ECS-LFCS-97-378, Lab. for the Foundations of Comp. Sci., U. Edinburgh, December 1997.

[8] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[9] Gupta, D., Jalote, P., and Barua, G. A formal framework for on-line software version change. *IEEE Trans. on Softw. Eng.*, 22(2): 120–131, February 1996.

[10] Hicks, M. *Dynamic Software Updating*. PhD Thesis. Dept. of CIS, U. Pennsylvania, Philadelphia, PA, 2001.

[11] Hj´almt´ysson, G. and Gray, R. Dynamic C++ classes. In *Proc. of the USENIX 1998 Annual Technical Conference* (Berkeley, CA, USA, June 15–19 1998), pp. 65–76, 1998.

[12] Kernighan, B. W. and Van Wyk, C. J. Timing Trials, or, the Trials of Timing: Experiments with Scripting and User-Interface Languages. *Software—Practice & Exp.* 28(8): 819–843, July 1998.

[13] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect Oriented Programming. *Proc. European Conf. on Object-Oriented Prog. (ECOOP '97)* (Jyväskylä, Finland, June 9–13), LNCS 1241, pp. 240–243, Springer-Verlag, 1997.

[14] Lam, J. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime, Demonstration at *Aspect Oriented Softw. Dev. (AOSD '02)* (Enschede, The Netherlands, April 22–26), 2002. http://www.iunknown.com/000092.html.

[15] Layer, D. K. and Richardson, C. Lisp systems in the 1990s. *Comm. of the ACM*, 34(9): 48–57, 1991.

[16] Lowy, J. Contexts in .NET: Decouple Components by Injecting Custom Services into Your Object's Interception Chain. *MSDN Magazine*, March 2003.

[17] Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J. F. Runtime support for type-safe dynamic Java classes. *Proc. Europ. Conf. on Object-Oriented Prog. (ECOOP '00)* (Cannes, France, June 12–16), 2000.

[18] Microsoft. The Profiler API of the Microsoft CLR. 2002.

[19] Microsoft. The Metadata Unmanaged API. 2002.

[20] Mikunov, A. Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. *MSDN Magazine,* Sept 2003.

[21] Miller, B. P., Christodorescu, M., Iverson, R., Kosar, T. Mirgorodskii, A., and Popovici, F. Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes. *Parallel Proc. Letters*, 11(2&3):267–280, 2001.

[22] Phelps, A. M. and Parks, D. M. Fun and Games with Multi-Language Development. *ACM Queue*, 1(10):46, 2004.

[23] Ousterhout, J. K. Tcl: An embeddable command language. *Proc. USENIX Winter Tech. Conf.* (Berkeley, CA, January 22–26), pp. 133–146, 1990.

[24] Ramsey, N. Embedding an Interpreted Language Using Higher-Order Functions and Types. *Proc. Workshop on Interpreters, Virtual Machines, and Emulators (IVME '03)* (San Diego, CA, June 12), 2003.

[25] Rigo, A. Representation-based Just-in-time Specialization and the Psyco prototype for Python. *Proc. Symp. on Partial Eval. and Prog. Manip. (PEPM '04)* (Verona, Italy, August 24–25), 2004.

[26] Schult, W. and Polze, A. Dynamic Aspect-Weaving with .NET. *Proc. Int. Symp. on Object-Oriented Real-Time Distrib. Comp. (ISORC '02)* (Crystal City, VA, April 29–May 1), 2002.

[27] Soules, C. A. N., Appavoo, J., Hui, K., Da Silva, D., Ganger, G. R., Krieger, O., Stumm, M., Wisniewski, R. W., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. System Support for Online Reconfiguration. *Proc. Usenix Tech. Conf.* (San Antonio, TX, June 9–14), 2003.

[28] Sridhar, N., Pike, S. M. and Weide, B. W. Dynamic Module Replacement in Distributed Protocols. *Proc. 23rd Int. Conf. on Distrib. Comp. Sys. (ICDCS 2003)* (Providence, RI, May 19–22), 2003.

[29] Staheli, D. G. Enhanced Debugging with Edit and Continue in Microsoft Visual C++ 6.0. *MSDN Library*, 1998.

[30] Vogels, W. Benchmarking the CLI for high performance computing. *IEE Proc.—Softw.*, 150(5): 266–274, October, 2003. http://cli-grande.sscli.net.

[31] 3 Leaf Solutions. Using the Edit and Continue Feature in C# 2.0. *MSDN Library*, 2004. http://msdn.microsoft.com/ library/en-us/dnvs05/html/edit_continue.asp.