

Volta: Developing Distributed Applications by Recompiling

Dragos Manolescu, Brian Beckman, and Benjamin Livshits, *Microsoft*

This tool suite recompiles nondistributed executables into functionally equivalent distributed form, inserting remoting and synchronization boilerplate code and facilitating post hoc instrumentation.

Contemporary programming languages and tool suites are designed for quick and easy construction of sequential, nondistributed applications. To write distributed applications, programmers must learn and use a large variety of lower-level libraries for cross-tier communication, data marshaling, synchronization, and security. The libraries' sole purpose is to support distributed execution of application logic that could just as well be executed sequentially. In fact, programmers often create prototype applications that run in simplified, streamlined, sequential environments

so that they can test and debug their code. Then they manually break up the prototype, inserting communication and synchronization code and distributing the pieces among multiple execution tiers.

We've created tools that, directed by declarative annotations such as `RunAt` and `Async`, insert boilerplate code and transform nondistributed executables into logically identical, asynchronous, distributed applications. The tools, released by Microsoft Live Labs as Volta, revolve around rewriting programs at the CIL (.NET Common Intermediate Language)¹ byte-code level. We call this approach *recompiling*.

Motivating example

Consider Word Worm (available at <http://labs.live.com/volta/samples/WordWorm.html>), a Volta application that does real-time word completion as the user types characters into a browser text box. Word Worm is a prototypical multitier Web application, with nontrivial application logic on both the client and server tiers.

We started this application on a single tier, with

all components written in C# and running on the client. We also used a data structure such as the prefix tree, which stores positional indices. This strategy let us implement and debug on a single tier the critical elements as well as cursors and sliding windows for the dictionary. We used a small mock dictionary for initial development and debugging. With the basic functionality in place, we transformed the application into a functionally equivalent distributed application, with the full dictionary residing only on the server. We effected the transformation through declarative annotations marking the distribution boundaries. Volta automatically inserted the necessary cross-tier communication, data marshaling, and synchronization as a recompilation step, and retargeted the client to JavaScript.

This example, although small, illustrates almost all of Volta's benefits. We can write applications in a familiar, comfortable environment with quick turnaround for debugging. When we're ready to deploy an application on multiple tiers, Volta automates much of the necessary replumbing and

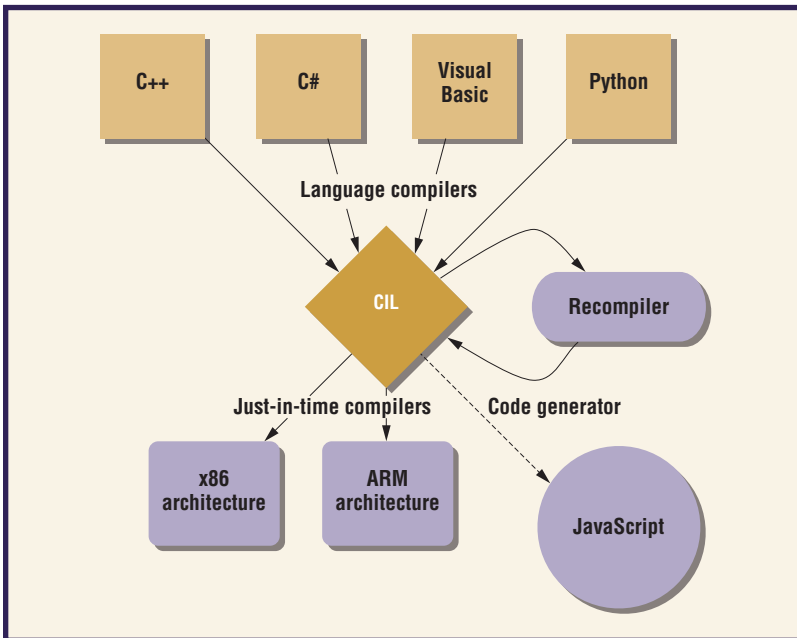


Figure 1. The CIL (.NET Common Intermediate Language) recompiler. Because the tool reads and writes CIL, it can take the CIL generated by any language compiler and recompile it into CIL with remoting, synchronization, and instrumentation code.

restructuring. Another way to look at this process is that Volta

- pushes distributed programming’s necessary but tedious, error-prone details into tooling,
- enables semantically neutral instrumentation, and
- extends refactoring and end-to-end profiling across runtimes.

We believe that Volta reduces the time needed for and cost of building distributed applications. Our hypothesis is based on feedback from colleagues and product groups using Volta, from developers on the Volta community forum (<http://tinyurl.com/3tdy3f>), and from personal experience building the online Volta samples. We’re working with partners to develop a disciplined usability study that will provide evidence. We hope that this article’s toy example will convey a sense of Volta’s benefits.

Distributed-application development

At the crux of distributed-system design lies the partitioning of functionality across tiers. The first criterion for partitioning is the location of resources, typically data on servers and presentation on clients. But partitioning must also account for the network’s adverse operational effects such as latency, low throughput, and packet loss. Although Volta handles many routine and tedious aspects of coding distributed applications, executing in a distributed environment introduces many concerns that tools can’t fully address.² For instance, local function

calls become remote procedure calls, and the operational semantics of failure and concurrency control are unavoidably different. However, we can try to retain much of the functional semantics: looking up words on a client dictionary is pretty much the same as looking them up on a server dictionary, except that we ignore surprising delays, dropped packets, out-of-order processing, duplicates, and so on. Likewise, formerly trusted execution now partially runs on an untrusted client.

Although partitioning should result from classic engineering trade-off analysis based on quantitative data, most tools and technologies force us to make an a priori partition before writing a single line of code—for example, clicking on the File menu, then New, then picking a client or a server project. When problems surface, we have a dilemma: should we stick with the current, now-known-broken design and kludge it into suitability? Or scrap it, start over, and eat the redevelopment cost? Sound engineering practice would have us simulating, prototyping, and measuring, and only then partitioning, partly on the basis of quantitative measurements.

Recompiling CIL

To get out of this predicament, we must make it easier to change software after it’s already “turning over”—that is, working in some demonstrable if vestigial way. More than a decade ago, object-oriented designers dealt with a similar dilemma. They came up with refactoring, a technique that facilitates change.³ Refactoring improves design by applying successive, behavior-preserving transformations. Nowadays, refactoring is so widespread that most development environments support it.

We’d like to bring two capabilities—refinement through successive transformations, and the ability to revisit design decisions often—to the development of distributed applications. In particular, we want our development tools to facilitate reassigning code to different tiers. Usually, refactoring implies absolute preservation of semantics. In our case, we preserve only the functional semantics; as we alluded to earlier, we know we can’t preserve operational semantics. We achieve this through transformations at the CIL bytecode level. However, these techniques aren’t restricted to CIL; we could apply them to Java bytecode, for example.

Figure 1 shows transformations from source code in several .NET languages to native code. Our recompiler operates in the middle, after the language compilers transform source code into CIL and before just-in-time compilers generate native code. The recompiler reads and writes CIL, and the recompilation entails CIL-to-CIL transformations.

CIL-level recompilation confers several benefits over other approaches, such as source-code transformation. First, the recompiler is language independent. Because all .NET programming languages compile into CIL, recompilation works the same way for programs written in any combination of .NET languages. This leverages the research and practical experience that went into the individual language compilers and tools. Our recompiler doesn't interfere with compiler analyses or optimization, and the developer enjoys all the benefits of source-level programming with integrated development environments (IDEs) such as Visual Studio.

Volta provides a many-to-many mapping between the languages in which we write distributed code and the runtimes where the code executes.⁴ This contrasts with the approach adopted by the Google Web Toolkit (<http://code.google.com/webtoolkit>) and Script# (<http://projects.nikhilk.net/projects/scriptsharp.aspx>), which operate at the source level. Consequently, it supports all the features of high-level languages rather than a subset. Moreover, we don't need to revise and update the recompiler when the language we're using changes but only when the CIL specification changes, which happens much more infrequently. For example, CIL didn't change at all during the time that C# evolved from version 2.0 to 3.0.⁵

How Volta works

This section covers the details of Volta recompiler implementation. After introducing a simple example as the exploratory vehicle, we turn to tier-splitting refactoring, semantically neutral instrumentation, and retargeting.

Distributed Fibonacci

We begin with a synthetic application that we tailored to show that

- the assignment of specific bits of code to client and server dramatically affects performance and
- Volta helps measure this performance and re-partition the code.

Consider the simple application in Figure 2, in which the application code computes the Fibonacci number for a positive-integer argument. Two C# classes carry out the computation: `MainComputation` implements the expensive recursive algorithm, and `BaseComputation` validates the inputs. We start by calling the `Fibonacci` method on an instance of `MainComputation`.

Once we have a working application running on a single tier, we're ready to make a trial partition,

```
public class BaseComputation
{
    public int Fibonacci(int n)
    {
        if (n < 2)
            return BaseComputation.Validate(n);
        else
            return Fibonacci(BaseComputation.Validate(n - 1)) +
                Fibonacci(BaseComputation.Validate(n - 2));
    }
}

public class BaseComputation
{
    public static int Validate(int argument)
    {
        if (argument < 0)
            return -argument;
        else
            return argument;
    }
}
```

Figure 2. Fibonacci computation. The application code computes the Fibonacci number for a positive-integer argument.

```
[RunAt("Server")]
public class MainComputation
{
    public int Fibonacci(int n)
    {
        if (n < 2)
            return BaseComputation.Validate(n);
        else
            return Fibonacci(BaseComputation.Validate(n - 1)) +
                Fibonacci(BaseComputation.Validate(n - 2));
    }
}
```

Figure 3. Tier splitting through declarative annotation. To begin the partitioning, we first convert the original local function call into a remote invocation, then into an asynchronous remote invocation.

measure performance, and potentially change the partition assignments. We first convert the original local function call into a remote invocation, then into an asynchronous remote invocation. The first transformation affects the trial partition; the second transformation makes the application robust against unpredictable network latency.

To partition the application, we mark `MainComputation` with the `RunAt` custom attribute (see Figure 3). The C# compiler converts this attribute into .NET CIL metadata and saves it in the executable.⁶ The Volta recompiler in turn reads the executable's metadata and inserts boilerplate code for remote invocation and for marshaling parameters and return values. No other changes to the source code from Figure 2 are required.

```
// declaration of asynchronous method (no definition required)
[Async]
public extern void Fibonacci(int n, Action<int> continuation);

// call site update for asynchronous invocation
mc.Fibonacci(argument, res) => Console.WriteLine("Result = {0}", res);
```

Figure 4. Declaration and client-side modifications for asynchronous invocations. The declaration instructs the C# compiler to insert the appropriate metadata into the generated CIL, and the call site update reflects the signature of the automatically generated method.

```
public class MainComputation
{
    public delegate int FibonacciDelegate(int n);

    public void Fibonacci(int n, Action<int> continuation)
    {
        var mc = new MainComputation();
        var ad = new FibonacciDelegate(mc.Fibonacci);

        var rp = new ResultProcessor(continuation);
        var cb = new AsyncCallback(rp.ReceiveResult);

        var ar = ad.BeginInvoke(n, cb, new Object());
    }

    public class ResultProcessor
    {
        private Action<int> _storedContinuation;

        public ResultProcessor(Action<int> continuation)
        {
            _storedContinuation = continuation;
        }

        public void ReceiveResult (IAsyncResult ia)
        {
            var ar = (AsyncResult)ia;
            var ad = ar.AsyncDelegate;
            var fd = (FibonacciDelegate)(ad);
            var result = fd.EndInvoke(ia);
            _storedContinuation(result);
        }
    }
}
```

Figure 5. Invoking a synchronous method with asynchronous delegates. Without Volta, we'd have to manually implement asynchronous invocation using .NET asynchronous delegates.

Volta's tier-splitting transformation is a refactoring in that it preserves the program's behavior.⁷ From the caller's viewpoint, it still looks like just the initial Fibonacci method. However, the recomplier transforms the single-tier application into an equivalent two-tier application by translating annotated local method calls into remote-service invocations.

Distributed-design best practices recommend

that all calls across the network be asynchronous. On the client, we declare a variant of the original Fibonacci function (see Figure 4), adding an additional argument of type `Action<int>` to receive the result. We then tag this variant with the `Async` attribute (the second line) and update the call site, supplying a lambda expression as an additional argument (the last line). On the basis of metadata on the method declaration, the Volta recomplier supplies the implementation of this new variant automatically.

The declaration of the asynchronous method is the only place where we make observable changes to the source code. Without Volta we would have to implement asynchronous invocation by hand, using .NET asynchronous delegates.⁸ Figure 5 shows the code we'd have to write.

The code for remoting and asynchronous invocation is too cumbersome to present here, easily overshadowing the original application code in accidental complexity⁹ and development cost. Tools can and should generate most of it, reducing the cognitive load on distributed-application developers.

Tier-splitting refactoring

We start building applications with the code running on one tier—typically the client tier, where we assume the typical .NET developer is most comfortable. All method invocations are local calls. Once we have a working application, we mark the classes that we want to execute on the server with the `RunAt` custom attribute (see Figure 3).

This attribute's presence in the CIL's metadata instructs the recomplier to apply tier-splitting refactoring, transforming the initial single-tier CIL into multitier CIL: a proxy-service pair. The code that calls the Fibonacci method on an instance of `MainComputation` doesn't change. Under the covers, the transformed Fibonacci method now calls a proxy. The proxy running on the client communicates with a service on the server, which executes the old body of the Fibonacci method. The recomplier moves the old body to the server, wrapping the original implementation within a service that responds to communications from the proxy on the client.

Figure 6a shows what the client code would look like if we had written it in C#. Of course, there's no C# source for the new client code; the recomplier operates entirely at the CIL level. The `Call` method of the `Proxy` class encapsulates communication over the network, including the complexities of `HttpRequest` on the server and `XmlHttpRequest` on the client. Likewise, the `GetInstance` method of the `Serializer` class returns a serializer suitable for marshalling arguments and results.

```

public int Fibonacci(int n)
{
    object[] objArray = new object[] { n };
    return Proxy.Call<int>(__Serializer.GetInstance(), __GetUri(), "MainComputation", 0, this, null, objArray);
}

```

(a)

```

protected override void ProcessMethod(Service<MainComputation>.Call call)
{
    switch (call.GetMethod())
    {
        case 0:
            call.Return<int>(call.GetInstance().Fibonacci(call.GetParameter<int>(0)));
            break;
            // additional dispatching
        default:
            throw new InvalidMethodException();
    }
}

```

(b)

Figure 6. Tier-split code reconstructed from generated CIL: (a) client and (b) server. The recompiler inserts code that invokes the service and dispatches the call to the initial code.

The new server code must include the body of the Fibonacci method from the old client code plus communication boilerplate. The recompiler exposes the initial application code by injecting a `Service` class into a copy of `MainComputation`—the one marked with `RunAt`. The service dispatches the invocations it receives from the proxy on the client to the Fibonacci method running on the server.

Figure 6b shows what the server code would look like if we had written it in C#. Of course, there's no C# source for the server code. The boilerplate code `ProcessMethod` takes an instance of the `Call` class. The `GetInstance` method of the `Call` class returns an instance of the transplanted `MainComputation`, code we wrote for the client. It contains the Fibonacci method (see Figure 2), which is our business logic. The boilerplate `Call` object also has a `GetParameter` method, which unmarshals the parameter values sent by the client. Finally, `Call` has a `Return` method, which marshals the computation's result.

So far, we have an easy way to make calls remote. The `Async` attribute (see Figure 3) declares our intent to invoke asynchronously the call we made remote. We must also change the Fibonacci signature to add a callback parameter, which is now available for static type-checking and IntelliSense. The recompiler supplies the implementation of the asynchronous delegates from Figure 5, so the declaration needs an `extern` modifier.

Volta doesn't make decisions about remoting boundaries; it only writes boilerplate code in a policy-agnostic manner. Also, Volta isn't in the same space as automated partitioning; if anything, the tool would facilitate such research.¹⁰

End-to-end instrumentation and profiling

Let's revisit Fibonacci and collect data about the partitioning between client and server. The developer sets a project option to enable end-to-end profiling. The option instructs the recompiler to instrument the tier-crossing boundaries. The instrumentation inserts time stamps and sends data to a logging service. The service—also generated by the recompiler—aggregates the data into trace files. Figure 7 illustrates user profiling, together with the remoting and instrumentation process.

The instrumented code can collect complete application traces, enabling us to compute statistics for latency and throughput and to perform application diagnostics. We have specialized tools for quickly interpreting the effects of partitioning. For example, Figure 8 displays the communication after tier-splitting `BaseComputation` in the Microsoft Service Trace Viewer.¹¹ The graph shows a chatty client-service interaction (which is why we tier-split `BaseComputation`). The time stamps let us quantify the processing time associated with each invocation, so that we can evaluate the effects of the initial, trial partitioning. Because partitioning the code was easy in the first place, changing the partitions on the basis of measurements is equally easy. In effect, we're promoting tried-and-true methods of quantitative optimization into the distributed realm.

Injecting instrumentation is a proven technique. The MIPS Pixie program lets programmers insert arbitrary instrumentation at multiple levels: instructions, basic blocks, functions, and so on.¹² Another instrumentation platform, AjaxScope,¹³ monitors the client-side behavior of Web 2.0 applications by

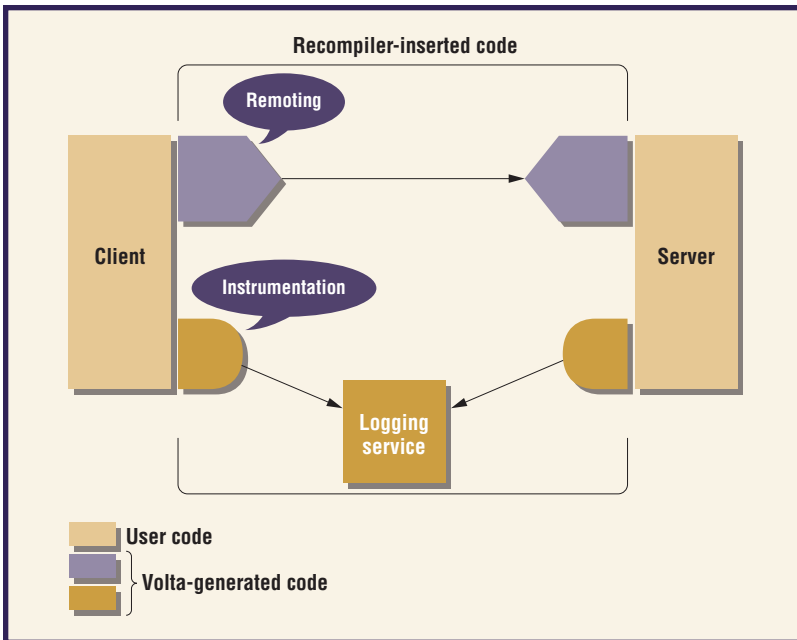


Figure 7. End-to-end profiling through recompiling. Data about the partitioning between client and server, including time stamps that help quantify each invocation's processing time, are aggregated into trace files that developers can use to quickly interpret partitioning's effect.

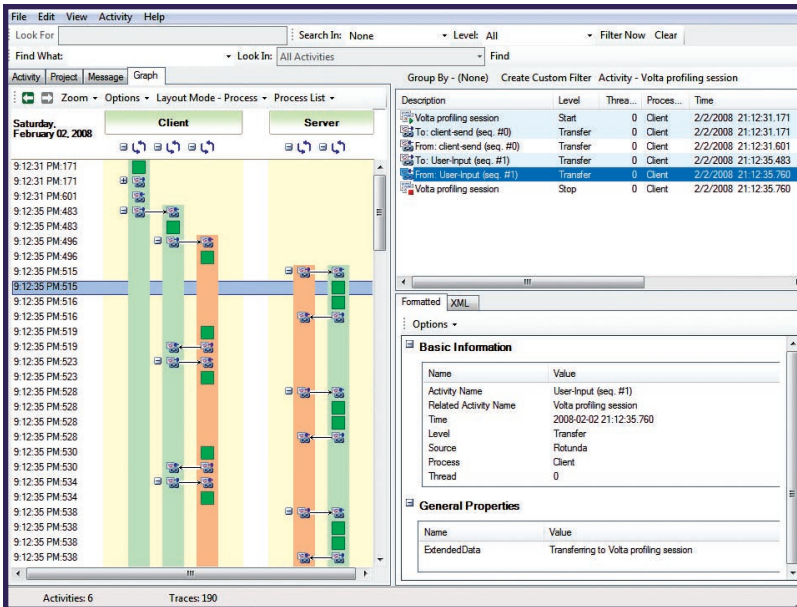


Figure 8. End-to-end profiling data displayed in the Microsoft Service Trace Viewer. The graphs let developers assess the effect of client-server partitioning.

injecting the instrumentation into client-side JavaScript. Just as in Figure 7, the instrumented JavaScript code periodically collates and sends log messages back to the AjaxScope proxy for analysis.

Rewriting changes the game in end-to-end profiling. The recompiler exposes a small surface area to the developer and pushes the details about collecting and aggregating instrumentation data into tooling. Putting end-to-end profiling literally one click away improves maintainability—we can easily

quantify the partitioning whenever functionality, location of resources, usage profiles, and other parameters change. In addition, through retargeting, we can instrument heterogeneous systems, where partitions execute in environments without a .NET runtime.

Retargeting

Ajax-style Web-based applications are one of the most popular forms of distributed applications. They involve a heterogeneous mix, in which the server side executes on a runtime environment such as CLR (Common Language Runtime) or JRE (Java Runtime Environment) and the client side executes in a JavaScript engine hosted by a browser.

We'd like to use Volta for code that runs in the browser. Using Volta to transform CIL into JavaScript was a natural decision, but it presents interesting challenges. For example, CIL code often uses the branch instruction `br`. JavaScript has no equivalent `goto` construct, so we simulate it with a trampoliner¹⁴ (a loop that iteratively invokes other functions) and a local program-counter variable (see Figure 9).

The full details of transforming CIL into JavaScript are beyond this article's scope. However, we can simulate advanced control-flow constructs that aren't natively supported by JavaScript, such as threads and coroutines.

Volta represents an innovative kind of software tooling. It lets developers refactor a sequential application into an equivalent distributed one with straightforward, declarative annotations placed in the source code. It also enables unobtrusive, post hoc instrumentation for quantitative evaluation of the partitioning. Finally, it extends the reach of .NET programming languages, libraries, and tools to cover the cloud. We achieve all three capabilities through recompilation of intermediate-language executables. Our approach has great promise for distributed-application development and beyond.

We've released a technology preview of the tools to the developer community. The release (<http://labs.live.com/volta>) includes several sample applications that we wrote in-house to verify our approach. The preview has attracted numerous early adopters, who used Volta to build various Web applications within a few days of the release.

In the long run, we expect tools such as Volta to dramatically reduce development costs and to increase application robustness by reducing the complexity of distributed-application development. By

bringing mature tools and familiar .NET programming languages into the realm of distributed applications, we hope to reduce developers' cognitive load and concept count.

Future Volta work will focus on security through construction, finer-grained tier splitting, and tier migration. Security by construction will add invariant checking code to both sides of tier-split applications. Finer-grained tier splitting will let developers annotate for tier-splitting methods or finer-grained constructs. Tier migration will allow parts of distributed applications to move dynamically. ☞

Acknowledgments

We thank our Volta team colleagues for doing the project's heavy lifting. The information in this article represents our personal views and doesn't necessarily represent the current view of our employer, Microsoft Corp.

References

1. *Standard ECMA-335, Common Language Infrastructure (CLI)*, Ecma Int'l, 2006, www.ecma-international.org/publications/standards/Ecma-335.htm.
2. J. Waldo et al., *A Note on Distributed Computing*, s.l., tech. report SMLI TR-94-29, Sun Microsystems Labs, 1994.
3. W.F. Opydyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Proc. 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications* (SOOPPA 90), ACM Press, 1990, pp. 145–160.
4. M.J. Foley, "Microsoft Architect Compares Volta and Google's GWT," blog, 7 Dec. 2007, <http://blogs.zdnet.com/microsoft/?p=1023>.
5. G. Bierman, E. Meijer, and M. Torgersen, "Lost in Translation: Formalizing Proposed Extensions to C#," *Proc. 22nd Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications* (OOPSLA 2007), ACM Press, 2007, pp. 479–498.
6. J. Bock, *CIL Programming: Under the Hood of .NET*, tech. report 978-1590590416, Apress, 2002.
7. W.F. Opydyke, "Refactoring Object-Oriented Frameworks," doctoral dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.
8. R. Grimes, ".NET Delegates: Making Asynchronous Method Calls in the .NET Environment," *MSDN Magazine*, Aug. 2001.
9. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995.
10. S. Chong et al., "Secure Web Applications via Automatic Partitioning," *Proc. 21st ACM Symp. Operating Systems Principles*, ACM Press, 2007, pp. 31–44.
11. "Service Trace Viewer Tool," *MSDN*, 2007, <http://msdn2.microsoft.com/en-us/library/ms732023.aspx>.
12. M.D. Smith, *Tracing with Pixie*, tech. report CSL-TR-91-497, Computer Systems Laboratory, Stanford Univ., 1991, pp. 1–29.
13. E. Kiciman and B. Livshits, "AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications," *Proc. 21st ACM Symp. Operating Systems Principles*, ACM Press, 2007, pp. 17–30.
14. S.E. Ganz, D. Friedman, and M. Wand, "Trampolined Style," *Proc. 4th ACM SIGPLAN Int'l Conf. Functional Programming*, 1999, pp. 18–22.

```
while (true) switch($next){
  case 0:
  {
    /** uninteresting JavaScript code removed */
    if (br1 || br1 === "") {
      $next = 27;
      continue;
    }
    /** uninteresting JavaScript code removed */
    $next = 42;
    continue;
  }
  /** other labels removed removed */
  case 60:
  {
    return loc_5;
  }
}
```

Figure 9. Implementing CIL's br in JavaScript, trampolined style. Branching to a location is equivalent to setting the \$next variable to the corresponding switch label.

About the Authors



Dragos Manolescu is a senior program manager on the Volta team at Microsoft Live Labs. His interests include software architecture, Web systems, and disruptive technologies. He received his PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at dragosm@microsoft.com.

Brian Beckman is a principal architect in Microsoft's Developer Division, where he works on optimization, security, transaction systems, programming languages, and physics for video games. Before that, he worked at Caltech's Jet Propulsion Laboratory on parallel and distributed operating systems and software engineering. Beckman received his PhD in astrophysical sciences from Princeton University. Contact him at brian.beckman@microsoft.com.



Benjamin Livshits is a researcher at Microsoft Research. His interests include compilers, static analysis, runtime analysis, application security, distributed systems, and Web application development. Livshits received his PhD in computer science from Stanford University. Contact him at livshits@microsoft.com.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.