Aspect-oriented Asynchrony in Java

University of Maryland at College Park
Computer Science Honors Thesis

Michael Nelson
May 26, 2003

Advisor: Dr. Michael Hicks

Table Of Contents

Abstract

Building scalable software services based on multithreading is difficult and time consuming. Loom is a prototype policy language and toolkit aimed at simplifying the integration and tuning of thread-based asynchrony into existing Java programs. Similar to weavers in Aspect Oriented Programming, developers indicate the threading behavior that they desire, and Loom rewrites the Java source code of their program to implement that behavior. Loom allows developers to indicate methods that should be made asynchronous and by what mechanism (e.g. thread per invocation, thread pool, etc...). It is a step toward our ultimate goal of developing a fully featured language that would support sophisticated, high-level threading policies such as the Staged Event-Driven Architecture developed at UC Berkeley. A major refactoring would be required to incorporate such an architecture into an existing program.

1. Introduction

Building scalable software services is an important and challenging proposition usually relegated to software practitioners with many years of experience in developing high performance software. Even then, practitioners often have to rely on a significant amount of tuning and testing in order to make applications truly scalable. Loom is a prototype implementation of a declarative policy language aimed at simplifying the process of incorporating thread-based asynchrony into Java programs. The policy language is focused on issues related to asynchronous program such as what methods should be made asynchronous and what concurrency mechanism should be used to achieve that behavior. A policy file indicates the asynchronous behavior that the Loom weaver[1] should implement by way of direct modifications to a program's Java source code. A few rules in the language can very easily influence changes to hundreds or thousands of lines of Java code in a program that a developer would otherwise have to make by hand.

Besides simplifying the tuning process, a policy language like Loom has the potential to support sophisticated, high-level concurrency policies using a small set of rules. One such policy might be modeled after the Staged Event-Driven Architecture (SEDA) developed at UC Berkeley [6, 7]. A program using the SEDA model is broken into a series of stages separated by *incoming event queues*. A web server, for instance, might be broken up into the stages of: accept connection, receive request, read disk, write response[2]. Each stage performs a discrete task that can be run in parallel across multiple threads. A thread pool associated with each event queue dispatches work items to *event handlers*. The event handler for each stage is responsible for performing a basic task (e.g. reading from disk) and then passing the results of its work on to other stages. Optional *controllers* can be provided to monitor a stage's behavior and performance, adjusting its thread pool size, the order of items in the queue, etc... as necessary. The architects of SEDA have demonstrated that their event driven model yields better scalability and lower latency

[1] Loom's behavior is similar to what a weaver does in Aspect Oriented Programming (AOP).
[2] This is an abbreviated list of the stages for a prototype HTTP server discussed in the SEDA paper.

as load increases than more typical monolithic designs – e.g. thread per message or using an application-wide thread pool.

With SEDA, a developer can focus on writing a series of event handlers, leaving the concurrency issues primarily up to the SEDA runtime. Leveraging SEDA, however, means that programs need to be specifically designed to work within the SEDA framework. Loom's ultimate design goal has been to enable SEDA-like functionality in existing, unmodified programs by using selected methods as the boundaries between stages – let's call them *boundary methods*. Calls to boundary methods would be asynchronous. Whenever a boundary method was called, the method's parameters would be put into a queue associated with it and then return. The actual work of the method call would occur when a thread became available to do the method's work. Thread pools would be associated with each boundary method, just like thread pools are associated with each stage in SEDA. Because their results would not be immediately available, boundary methods would return futures as results. A future [2] is a proxy object that pretends to be the result that a method would have returned. Actually using a future will cause it to wait until the actual result object becomes available. The power of futures is their ability to be stored in data structures or passed to other functions without waiting for the underlying result data to become available. Because that data isn't always needed immediately, futures allow developers to increase the parallelism and flexibility of their multithread applications by not waiting unnecessarily. Developers wouldn't have to change a single line of code by hand to enable any of this behavior.

This purpose of this paper is to describe the implementation details of the current version of Loom, a step toward the goal of providing a tool that can help existing programs implement SEDA without being significantly refactored. Its primary feature is the ability to make arbitrary method calls asynchronous. Asynchronous methods are supported by two different threading models – thread per invocation or thread pool backed by a queue. Loom has been used with small test drivers as well as Jetty, an open source web server.

2. Loom

2.1 Loom Policy Language

The Loom policy language currently simplifies the task of making specific methods in Java classes asynchronous. Anyone familiar with the basics of multithreaded programming and thread pools should be able to learn and experiment with the policy language in a short period of time. Using a simple set of rules, a developer can declaratively specify methods that should be made asynchronous and by what mechanism – e.g. one thread per invocation, via a thread pool with 6 to 10 threads, etc... Policy rules are specified in a policy file that is processed (or "weaved") along with one-or-more Java source files using the Loom weaver program. The weaving process involves modifying the Java source files as appropriate to enact the policy, so you must have the source files of any class whose methods you want to make asynchronous. This process is described in more detail in Section 3. Asynchronous methods can be public, private, or static and have any number of parameters.

The results of asynchronous method calls are substituted by futures, automatically generated proxies whose type and behavior resembles the original result. By using futures, consumers of asynchronous methods don't have to wait for a method's result to be generated, which would make such methods pointless. Whenever a method on a future is called, it checks to see if the result to the asynchronous method call has finally become available. If so, the future's method call is passed along to the actual result object. If not, the future waits until the actual result becomes available.

A future's resemblance to the original result object is not perfect. Only public methods are accessible via futures in Loom; it is not possible to access private or static methods, or use fields. Each time a future method is called, it has to do a synchronized check to see if the actual

result has been sent to the Future. The overhead for using futures is fairly small[3]. The process of generating futures requires the Java source code to class of the asynchronous method's return value. This precludes the use of built-in types and primitives as return types from asynchronous methods. Since futures are proxies and do not derive from the original result class, they are not type equivalent. Loom currently includes a partial workaround for this issue – the future and the original result class are setup and modified, respectively, to implement an interface listing the method calls that are common to both. However, code where the original result class is used is not rewritten currently to indicate that it supports the common interface.

2.2 Policy Files

Loom policy files have two major section types – **global** and **class <classname>**. Any settings that need to be applied or initialized on a global basis are specified in the **global** section. The primary use of the global section is for initializing global *thread managers*, which will be explored shortly. One or more **class** sections specify the transformations that should be applied to specific Java class source files. When you want to make a method in a particular class asynchronous, you do so in the class section associated with that class's name. Class sections are also used to initialize per-class and per-instance thread managers.

[3] In tests on a 1.13Ghz Pentium, running Windows XP (Service Pack 1), Java Hotspot Client VM 1.4.0_01-b03, the difference between calling a Future and an actual object is approximately 200ms in 100 million iterations.

Here is an outline of a sample Loom policy file:

```
global {
    thread-manager "..." {
          ...
      }
}

class "myclass" {
      thread-manager "..." {
            ...
      }

      method "..." invoke-via "..." priority ...;
}
```

So, what are thread managers? Simply put, thread managers associate a custom threading policy with a named identifier. The two categories of threading policies supported by Loom are: 1) thread per method invocation, and 2) a thread pool backed by a queue. Let's say you want to have a thread pool called "foo" that starts out with 3 threads and has an upper bound of 10 threads during its lifetime. You would write a **thread-manager** subsection that looks something like the following:

```
thread-manager "foo"
{
      min-threads 3;
      max-threads 10;
}
```

That subsection would then be placed into the **global** section of the Loom policy file if you want the thread manager available to any asynchronous method in the program. If you want to limit that thread manager's use to methods in a specific class, you would put it in the appropriate **class** section. Class-specific thread managers can be made per-class or per-instance. In per-instance thread managers, a separate thread manager is created every time a new instance (object) of the class is created.

Methods in a class are associated with a particular thread manager using the **method** keyword in a **class** section:

```
class "myclass"
{
      [...]
      method "bar" invoke-via "foo" priority 1;
}
```

In this example, whenever *myclass*'s bar method is called, it will be executed on a thread from the *foo* thread manager. The **priority** attribute is not currently used.

2.3 Weaving Process

Loom processes the policy files and Java source files identified by command line parameters in 3 phases: parsing, code generation, and merge.

2.3.1 Parsing Phase

The type information (classes and methods) and abstract syntax tree for each Java source file is stored for later use. Each policy file is parsed, validated against the stored type information (e.g. to verify class names, method names, etc...), and then stored.

2.3.2 Code Generation Phase

The code generation phase is responsible for generating the following pieces of Java source code: invocations, thread managers, stubs, and futures. They are presented on separate pages so that their example code can be seen clearly.

Invocations

An invocation class is an inner class created for each unique asynchronous method, and stored within the method's class. So, if there are 3 different asynchronous methods *a(), b(), c()* in class *Foo*, 3 different invocation classes will be stored inside of *Foo*. A specific instance of an invocation class stores the parameters for a specific call to an asynchronous method. If I make a call to *c()* with the parameters (1, 2), a new instance of the invocation class associated with method *c()* will be created and initialized with the parameters 1 and 2. Once initialized, instances are passed to a thread manager which will call the former's r*un()* method[4] when a method is scheduled to be executed. An example of an invocation is presented below.

```
class Foo
{
      // The actual method call implementation
      void Loom__Impl_test (int a, int b)
      {
            System.out.println ("This is Foo.test()! You passed
            in" + a + " " + b + "\n");
      }

      // The invocation class
      class Loom__Invocation_Foo_test implements Runnable
      {
            int a;
            int b;

            Loom__Invocation_Foo_test (
                  int a,
                  int b
            )
            {
                  this.a = a;
                  this.b = b;
            }

            public void run()
            {
                  Loom__Impl_test (a, b);
            }
      }

      // Stub for test
      void test()
      {
            // Will create an instance of the invocation and
            // send it to a thread manager
      }
}
```

[4] Invocations implement the Java *Runnable* interface.

Thread Managers

As mentioned earlier, thread managers are responsible for scheduling method execution according to some policy. The mechanism by which method calls are passed to thread managers is explained in the next section about stubs. Loom generates a small amount of code to initialize each thread manager based on its settings in the policy file. The bulk of the code to support thread managers is shared and located in Loom's runtime library.

Global thread managers are available through the *Loom__Global* singleton class. They are stored as fields within the class and use the name specified in the policy file. An example of a simple *Loom__Global* class is provided below. Here, tp1 and tp2 would have been specified in the **global** section of the Loom policy file.

```
class Loom__Global
{
    public static final Loom__Global instance = new
Loom__Global();
    LoomThreadPerMessage tp1;  // Thread manager
    LoomThreadPool tp2;        // Thread manager

    Loom__Global()
    {
        tp1 = new LoomThreadPerMessage();
        tp1.initialize();

        tp2 = new LoomThreadPool();
        tp2.minThreads = 4;
        tp2.maxThreads = 50;
        tp2.initialize();

    }
}
```

Class-specific thread managers are accessible via an inner class belonging to the class that owns them. An instance of that inner class is available to code as a field (unfortunately) named *manager*. An example is provided below.

```
class Test2
{
   Loom__Manager_Test2 manager = new Loom__Manager_Test2();

   // Test2 specific thread managers
   class Loom__Manager_Test2
   {
       LoomThreadPerMessage tpm;  // Thread manager
       LoomThreadPool tpo;        // Thread manager

       Loom__Manager_test2()
       {
           tpm = new LoomThreadPerMessage();
           tpm.initialize();

           tpo = new LoomThreadPool();
           tpo.minThreads = 1;
           tpo.maxThreads = 1;
           tpo.initialize();

       }
   }

   // Code would access thread managers via manager
}
```

Stubs

A stub is responsible for taking the parameters to an asynchronous method call, putting them into an invocation, and passing the invocation to a thread manager for processing using the latter's *invoke()* method call. Here is an example of the code transformation process that takes place when a method in a class is transformed from synchronous to asynchronous. Note that the original method name is prefixed with `Loom__Impl_` during the merge phase so that it doesn't conflict with the stub.

Before

```
class Foo {
      // Original function
      public void test (int a, int b) {
            System.out.println ("This is Foo.test()! You passed
            in" + a + " " + b + "\n");
      }
}
```

After

```
class Foo {
      // Original function, renamed
      public void Loom__Impl_test (int a, b) {
            System.out.println ("This is Foo.test()! You passed
            in" + a + " " + b + "\n");
      }

      // Newly created stub
      public void test (int a, int b) {
            Loom__Invocation_Foo_test_1 invocation;

            invocation = new Loom__Invocation_Foo_test_1 (
                  a,
                  b
            );

            thread_manager.invoke (invocation);
      }
}
```

Futures

Recall that if a method is made asynchronous and returns a result, the result is wrapped by a future that is dynamically generated by Loom. The future allows the asynchronous method call to return before the method's actual results become available. Being a proxy, it does not inherit from the actual result's class and would not, by default, be type equivalent with it. This would cause problems if, for example, there was code that tried to put the method's result into an array of the result class's type. To workaround this, an interface with all of the public methods from the result class is created. The generated future implements this interface and the result class is modified to indicate that it implements this interface. (Note: Loom does not currently modify locations in the program that use the original result class's type to indicate that they support this newly created interface.) An example of the transformation process is provided below.

Before

```
class Foo
{
      // Method that is going to be made asynchronous
      public result x3 (String q)
      {
            return new result();
      }
}

class result
{
      public int getA()
      {
            return 42;
      }
}
```

After

```
class Foo
{
      [...]

      // Actual implementation of x3
      public result Loom__Impl_x3 (String q)
      {
            return new result();
      }

      // Asynchronous stub for x3 that returns future
      public Loom_Ifc__Result x3 (String q)
      {
            // stub code
      }
}

// Common interface shared by result and future
interface Loom_Ifc__result
{
      int getA();
}

// Result class that was modified to implement the common itf
class result implements Loom_Ifc__result
{
      public int getA()
      {
            return 42;
      }
}

// The future version of result
class Loom_future__result implements future, Loom_Ifc__result
{
      // Support code

      public int getA()
      {
            ...
      }
}
```

In addition to the public methods that were implemented by the original result class, futures automatically dispatch *toString(), hashCode(), equals(), getObject(), and clone()* in an effort to be as compatible as possible. This compatibility is not foolproof however. Problems may arise if both the actual result object and the future are used actively within the program. This concurrent use would occur if an asynchronous method call stores its result in a data dictionary before returning it. Code that encounters both the future and/or the actual result and uses methods such as *wait()* and *notify()*, or tries to compare object references (a == b), might run into trouble.

Futures expose a *Loom_Future_set()* function that is used by thread managers to notify the future when the asynchronous method call's actual result has become available. All of the methods within a future use the synchronized function *Loom_Future__touch()* to check if the result is available in a thread-safe manner. An example of what a simple future object looks like is provided below.

```
class Loom_Future__result implements Future, Loom_Ifc__result
{
    private volatile Loom_Ifc__result __result = null;
    private volatile boolean finished = false;

    private synchronized Object Loom_Future__touch()
    {
        try {
            while (!finished) wait();
        } catch (InterruptedException e) {
            throw new Error("ERROR: Loom_Future__touch()
Interrupted");
        }
        return __result;
    }

    public synchronized void Loom_Future__set (Object obj)
    {
        if (finished)
            throw new Error ("ERROR: set() called twice");
        finished = true;
        __result = (Loom_Ifc__result) obj;
        notifyAll();
    }

    public String toString()
    {
        if (!finished) Loom_Future__touch();
        return __result.toString();
    }

    public int hashCode()
    {
        if (!finished) Loom_Future__touch();
```

```
        return __result.hashCode();
    }

    public boolean equals (Object obj)
    {
        if (!finished) Loom_Future__touch();
        return __result.equals (obj);
    }

    public Object getObject()
    {
        if (!finished) Loom_Future__touch();
        return __result;
    }

    public Object clone ()
    {
        if (!finished) Loom_Future__touch();
        return __result.clone ();
    }
    public int getA ()
    {
        if (!finished) Loom_Future__touch();
        return __result.getA ();
    }
}
```

2.3.3 Merge Phase

During the Merge phase, three types of files are generated:

- Any of the generated source code that modifies or enhances existing Java source files is parsed into an AST and grafted onto the original Java source file's AST. The combined AST is then outputted to a new file called L_<classname>.java (where <classname> is the name of the class that was modified). So, if methods in Foo (Foo.java) are made asynchronous, L_Foo.java will be generated.
- Loom__Global.java is created to store global thread managers.
- Any futures that are created are put into files named L_Future_<classname>.java (where <classname> is the name of the result class that was "futurized"). Files named L_Ifc_<classname>.java are also created and contain the common interface between each future and result class.

2.4 Compiling a Weaved Project

Developers need to modify their makefiles/build scripts so that the generated L_*.java files and Loom__Global.java file are compiled. Java source files that are replaced by modified L_*.java versions must not be compiled since they would conflict with the modified versions. There is currently no mechanism that indicates which Java source files were modified during the weaving process.

3. Implementation

3.1 Tools and Libraries for Building Loom

The Loom weaver is platform independent and targets the Java 1.4 runtime environment. Policy files and Java source files are parsed using Sun Microsystems' JavaCC 2.1 lexer/parser [3] and Purdue University's Java Tree Builder (JTB) 1.2.1 [4]. JTB was designed to work hand in hand with JavaCC and is able to parse any JavaCC grammar file without modification. JTB greatly simplifies the code necessary to walk abstract syntax trees (ASTs) by allowing you pass in visitor classes (a la the Visitor design pattern [1]) whose overloaded *visit()* methods get called when an AST node of a type you are interested is encountered. For each grammar you compile with it, JTB generates a visitor class that will print an AST out in a form that looks almost identical to the original source. On the face of it, this would seem to make the job of source code modification simple – use visitor classes to modify the AST where desired and then print out the modified AST. Unfortunately, JTB's printer visitor classes store absolute column and row information for each token in the AST. If you want to modify ASTs and then render them, as Loom does, you have to adjust the row and column numbers on code that is "downwind" from a change. If you don't do this, the AST will not be rendered properly. Loom includes a workaround like that to cope with this problem.

The Java 1.4 grammar file used to parse Java source files was downloaded from the "JavaCC Grammar Repository" (http://www.cobase.cs.ucla.edu/pub/javacc/). The bulk of the grammar file for the Loom policy language was written from scratch, with lexer definitions borrowed from the Java grammar file for things such as string literals, integer literals, etc...

3.2 Loom Source Code

The weaver's source code is divided into four major packages: information database (*Loom.info*), Java source file parser (Loom.*java*), Policy manager and source file parser (*Loom.policy*), and the Loom runtime (Loom.*runtime*).

The *Loom.info* package maintains class and method information from the Java source files that Loom has parsed, information that is used by multiple components in the weaver.

The *Loom.java* package is responsible for parsing Java source files, registering their type information with the *Loom.info* package, and keeping track of their ASTs so that they can be modified when necessary. This source file parsing components of this package are conveniently implemented as a set of visitor classes which are invoked via JTB during the appropriate phases of the weaving process. The major classes in *Loom.java* are as follows:

- *JavaFileManager* is the primary class called by code outside of the *Loom.java* package when the need to parse and merge Java source files arises.
- *JavaCodeAnalysisVisitor* is responsible for parsing the class and method definitions and putting their information into the information database.
- *JavaClassMergerVisitor* takes the AST of policy generated class and integrates it into an existing class. This is used, for example, when incorporating asynchronous method stubs and thread manager code into existing source files.
- *JavaImplementsVisitor* is used to add a specific name to the list of interfaces that a class implements.
- *JavaRenameVisitor* renames the name of any methods which are going to become asynchronous.

The *Loom.policy* package is responsible for parsing Loom policy files, keeping track of policy information, and emitting the Java source code that supports futures, thread managers, and stubs. The major classes comprising this package are as follows:

- *PolicyManager* is the primary class used by code outside of the *Loom.policy* package to parse policy files, store and retrieve policy information, and generate all the Java source code necessary to support the rules in a policy.
- *GlobalPolicy* keeps track of all global thread managers and provides a method to emit the code associated with them to a Java source file.
- *PolicyProcessor* is a visitor class responsible for parsing the information in a policy file.
- *Futures* is responsible for emitting future classes and the common interfaces shared by futures and original result classes they are paired with.

The *Loom.runtime* package is used by code generated by the weaver and not by the weaver itself. It provides a common implementation of the thread manager types supported by Loom, per-method-invocation and thread pool, via the *LoomThreadPerMessage* and *LoomThreadPool* classes, respectively.

4. Applying Loom

The Loom weaver was applied in two different scenarios: (1) a series of simple test cases, and (2) to replace the built in thread pooling mechanism in Jetty version 4.2.9 [5], a popular open source web server.

In the first scenario, a series of classes and methods were written to evaluate and debug Loom's ability to handle scenarios that it might encounter in the real world. Methods with a mix of parameters, return values, and attributes (e.g. static, public, private) were made asynchronous. Combined with various types of threading policies, the generated code was validated by hand and executed to verify its correctness.

In the second scenario, the Jetty web server source code was downloaded and the sample configuration its author provides was setup on a Red Hat Linux 7.3 server. Jetty's architecture is modular and simple to understand, making it a good test bed on which to try to apply Loom. The primary goal of this scenario was to test Loom's ability to be smoothly incorporated in a moderately complex program that already contains multithreading support. Given Jetty's modular architecture this was easily achieved. Because thread pooling resizing at runtime was not fully developed in Loom, no performance comparison was made between the original and modified versions of Jetty.

Jetty's *ThreadedServer* class (in the package *org.mortbay.util*) contains the code responsible for dispatching incoming socket connections to the appropriate protocol handlers (e.g. HTTP) via a thread pool. One or more *Acceptor* objects are responsible for the dirty work of accepting the connections and then dispatching them to Jetty's thread pool. *Acceptor* is an inner class of *ThreadedServer*. When dispatching a connection, an *Acceptor* calls *ThreadServer*'s *run()* method with the Java *Socket* object as a parameter.

Once the basic layout of Jetty's source code was understood, introducing Loom's own threading code was easy. The first step was to figure out what method handled the actual processing of HTTP connections. In Jetty's case, that is the responsibility of the *handle()* method

in *ThreadedServer*. Next, the code within *Acceptor* that dispatches incoming socket connections to the thread pool was replaced with a direct call to *handle()*. Finally, a simple policy file was written that associated the method *handle()* with a thread manager that spawned a thread for each call to *handle()*.

5. Conclusion and Future Work

Loom has allowed us to investigate and debug the basic job of making methods in Java source files asynchronous. On simple projects this has been successful and gives hope that a primitive SEDA-like environment based on Loom is not too far away. The following is a list of some of the major issues that still need to be explored:

- Overall futures support needs to be significantly enhanced. Sites where an original result class is used need to be updated to match the common interface that it and its future share. Support for modifying class files needs to be implemented so that a wider range of classes can be turned into futures. The limitation that futures can only relay public method calls may, in some cases, cut against our desire for a tool that requires no hand modified of Java source code. Static code analysis was explored as a way of being able to safely determine when a future could be thrown away, but wasn't implemented due to this project time constraints. This work would help reduce the overhead that futures impose.
- Thread manager behavior needs to be fleshed out significantly. For example, thread pools currently do not resize themselves according to load. Nor is there any infrastructure in place that allows user-supplied code to monitor and modify the behavior of thread managers (like controllers do in the SEDA architecture).
- Finally, building source code that has been modified by Loom is cumbersome at best. There needs to be better integration with build tools such as Ant for Loom to be viable in medium to large projects.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] D. Lea. *Concurrent Programming In Java*. Addison-Wesley, 1999.

[3] JavaCC website. http://www.webgain.com/products/java_cc.

[4] Java Tree Builder website. http://www.cs.purdue.edu/jtb.

[5] Jetty Web Server website. http://jetty.mortbay.org/jetty.

[6] M. Welsh, D. Culler, E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. http://www.cs.berkeley.edu/~mdw/papers/seda-sosp01.pdf

[7] M. Welsh, D. Culler. Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services. http://www.eecs.harvard.edu/~mdw/papers/seda-hotos01.pdf