# Introduction to the Plugin project

## Table of contents

## 1. About the Plugin project

The Plugin project provides a framework for implementing plugins in C. The plugin system is inspired by the plugin system which is the core of the Eclipse IDE but there are some significant differences due to the different languages of implementation.

One of the major goals of this project is to allow GCC to be an extensible compiler. Hence, there are plugins available for GCC and a GCC patch which adds plugin capabilities to that compiler.

The plugin system is designed to make it very easy to make an existing application extensible and, similarly, to make it easy to extend that application.

This project is part of the Milepost project.

**Warning:**

This project is in alpha state. This means that documentation and testing may not be very impressive. Also, the APIs are subject to change and installing is going to be more difficult that in a full release.

## 2. Why use a plugin system

A plugin system provides a way to add or modify behaviour of an application after it is built. Plugins let us keep our application free all of the ugly code that is required to let users tweak things. They also give our users the ability to increase the power of our application by themselves, probably in ways we never thought of.

### 2.1. An example - application extensibility the bad way

Let's consider a concrete example. Suppose we have a compiler that is able to do a loop unrolling optimisation. Now inside our compiler we would have a function somewhere that says, "You should unroll this loop by X times". It's our heuristic for loop unrolling.

```
int decideUnrollTimes( loop* lp ) {
        int times = /* the heuristic */;
        return times;
}
```

Well, a user might reasonably ask, "What loops have been unrolled, and by how much?" Ok, we could print all of this information to a file. We've got to write something which looks like:

```
int decideUnrollTimes( loop* lp ) {
```

```
        int times = /* the heuristic */;
        if( userWantsDumpInfo ) {
                print "Loop " + lp + " unrolled by " + times;
        }
        return times;
}
```

Not, perhaps terribly hard, but it's a bit irritating that it has to be there (And if the job were done properly, it would take substantially more code than that).

Well, now our user might say, "Ok, but I want to have control over which loops to unroll". We could add a #pragma somewhere in the mix but that is likely to be a fair bit of work. We could write some code to let them give a file with a list of loops and their unroll factors.

```
int decideUnrollTimes( loop* lp ) {
        int times = /* the heuristic */;
        if( userWantsControl ) {
                load and parse the file.
                times = the user's choice
        }
        if( userWantsDumpInfo ) {
                print "Loop " + lp + " unrolled by " + times;
        }
        return times;
}
```

But now we really would have quite a bit of code in there and it wouldn't have anything to do with loop unrolling, it would just be to allow the user to have some control.

What, though, if our user comes back and says they want more control than that? In some cases, they want to know the original value of the heuristic before making their decision, and well, they'd also like to check the weather in Peru since they think that will be relevant!

Now we would have to let them load some shared library that can take control. And we better think of how they can specify command line parameters to the library and while we're at it we ought to let them have multiple libraries, just in case.

```
int decideUnrollTimes( loop* lp ) {
        int times = /* the heuristic */;
        if( userHadSharedLibraries ) {
                for each( library in librariesForUnrolling ) {
                        lib = load( library )
                        f = findSymbol( lib, "controlUnrollTimes" );
                        times = f( lp, times );
                }
        }
        if( userWantsControl ) {
                load and parse the file.
                times = the user's choice
```

```
        }
        if( userWantsDumpInfo ) {
                print "Loop " + lp + " unrolled by " + times;
        }
        return times;
}
```

Ooof! Once we flesh out the pseudo code and add in all the error checking, we get quite a lot of gumph which has made our code look like a total mess! It's probably required lots of changes elsewhere, too, so that we know what files the user specifies, and how much control they want. It might hard to even find the original heuristic in there.

One more thing; if the original heuristic was very expensive to calculate and the user was just going to override it, we would have wasted our time. Consider putting in the possibility to *not* calculate the default up there and it becomes even more ugly.

## 2.2. An example - application extensibility the plugin way

The plugin system, however makes all of that simple and clean. All we have to is to call the *decideUnrollTimes* function through a function pointer instead.

```
int decideUnrollTimes_default( loop* lp ) {
        int times = /* the heuristic */;
        return times;
}
int ( *decideUnrollTimes )( loop* lp ) = decideUnrollTimes_default;
```

Then we write a little plugin specification in XML which tells the system that this function pointer can be replaced by plugins. That's it!

```
<?compiler version="1.0"?>
<plugin id="compiler.loop-unrolling">
        <library path=""/>
        <join-point id="compiler.loop-unrolling.decide" signature="int f(
loop* )">
                <call symbol="decideUnrollTimes"/>
        </join-point>
</plugin>
```

Well, nearly. We also provide a plugin to print the unrolled loops and one to read the user's overrides from a file. But neither of those plugins infect our application.

All of the loading of libraries, managing dependencies, passing command line arguments, deciding which bits to override will be taken care of for us. The user can easily add their own plugins to do things we never even thought of.

Also, the only cost we pay if the user decides not to change or record the behaviour of loop unrolling is the cost of calling from a function pointer, not directly.

The plugin system is simple, clean and efficient.

## 3. What are plugins

A plugin consists of a specification in XML and possibly some number of shared libraries. The XML file describes all of the information needed to load the plugin: it's identifier, what shared libraries it needs, what other plugins have to be loaded and what interactions it has with other plugins.

Plugins interact with each other and the application through *extension-points*. Each *extension-point* is described in the plugin specification and is typically backed a function in a shared library. The function knows what to do when other plugins extend the *extension-point*. To extend an *extension-point*, other plugins provide *extension*s, which are pieces of XML that the *extension-point* of interest understands.

Many plugins can be completely described by their XML file, without any shared libraries. They make use of the *extension-points* from other plugins which only need to read the XML in the extension to know what to do. Others need to provide a shared library to either create extension points or to add code based extensions.

The contents of an example plugin are shown above. It consists of an XML specification, in file *plugin.xml*, and a shared library, in *lib.so*. The plugin file tells the system where to find the library. It also says that the plugin provides an *extension-point* and an *extension* of another *extension-point* from a different plugin.

Plugins interact with each other, and the application. There can be lots of plugins, all working together. The system will manage plugin dependencies and ensure that only necessary plugins and extension-points are created.

Extension-points can be extended multiple times and plugins can even extend their own extension-points. The application can also provide extension points for plugins to use.

## 4. libplugin

The main system is delivered as a library which can be statically or dynamically linked to the application. It provides the core functionality of the system: loading plugins, managing dependencies, version management and so on.

The application initialises the plugin system, decides what capabilities plugins can use and then busies itself with it's ordinary tasks, some of which may have been extended by plugins.

It is possible, if you are developing an application from scratch, to have almost nothing but a tiny main which sets up the plugin system and offers no other functionality itself. Plugins would then provide extension points and extend each other to build the application.

Similarly, it is possible to take an existing application and make it 'plugin aware'. This entails setting up the plugin library and making a few small changes to indicate which core functionality can be extended. Plugins can then extend those core points or make their own extension points for others to extend.

## 5. Application agnostic plugins

Several plugins are provided which can be used in any application (if the application allows them).

## 6. GCC patch and plugins

The whole raison d'être for the plugin system was to make GCC extensible. This is necessary for the Milepost project since being able to replace GCC heuristics is a core requirement of that project. This also serves as a good test bed for the plugin system.

A patch to GCC is provided which initialises the plugin system. It also provides a few refactorings to allow some heuristics, etc. to be extended. Over time, we expect more such refactorings to be made.

The patch is currently as applied to the trunk of GCC.

There are also a number of plugins that turn the refactorings into extension points.