The ADL Plugin Interface

Plugins

This document describes the ADL plugin interface, which allows users to add support for additional types of input and output files and system calls.

Author: Brian Kahne
Contact: brian.kahne@freescale.com

Table of Contents

1 Overview

By itself, ADL can handle several different types of input and output files. It can also handle a simple set of system calls using a memory-mapped interface known as a porthole. If a user wants to be able to process additional types of input files, create a new type of output file, or extend the system-call interface, he or she can create a *plugin* and have it installed at run-time. This is simply a shared object which contains the additional functionality and conforms to the ADL plugin interface.

2 The Interface

An ADL plugin must implement an entry point function which returns a pointer to a service-provider class. The entry point is declared as:

```
extern "C" Plugin *adl_plugin_entry()
```

The `Plugin` class is declared as:

```
struct Plugin {

  virtual ~Plugin() {};

  // Generalized initialization routine.  This is called when the plugin is
  // loaded and allows the plugin to perform custom actions, such as
  // registering custom loggers, etc.  The method has access to the options
  // database and may install new command-line options at this point.
  virtual void init(AnyOption &options) {};

  // Called after final option processing.  If the plugin should access the
```

```
  // command-line database at this point to read any needed option values.
  virtual void setup(AnyOption &options) {};

  // If 'type' is recognized (this will be the extension of an output file), then construct
  // the relevant writer type.  If not recognized, return a 0.
  virtual Writer *createWriter(IssNode &root,const std::string &type,const std::string &fn,const MemoryL

  // Same as above, except this is for constructing a stream writer, which
  // writes data to an output stream.
  virtual Writer *createStreamWriter(IssNode &root,const std::string &type,std::ostream &os,const Memory

  // If 'type' is recognized (this will be the extension of an input file), then construct
  // the relevant reader type.  If not recognized, return a 0.
  virtual Reader *createReader(IssNode &root,const std::string &type,const std::string &filename) { retu

  // If this plugin has a system-call handler, then create such an object.  If not, return 0.
  virtual SysCallHandler *createSysCallHandler() { return 0; };

  // If this plugin defines an external memory handler, then create such an
  // object.  This method should call setMemHandler on the root argument (or
  // its children if it's a system).
  virtual void createMemHandler(IssNode &root) {};
};
```

A user should publicly derive a class from `Plugin` and then implement the appropriate functions. The entry point function should then return a pointer to an instance of this class. Only a single instance is required, so the object may be defined as a static global within the shared object.

When an input or output file is being processed by the ADL framework, a file's type is derived from its name by looking at its file extension. This will be overridden if the user has specified a specific output format using the `--output-format` or `--of` option. The framework then calls the appropriate creation function for all installed plugins. If a plugin can process a file, then it should instantiate an appropriate object (a `Reader` or `Writer` derived object) and return a pointer to this object. Note that the base class for readers and writers is garbage collected, so a plugin does not need to worry about deleting these objects. The interface for a reader object is declared in `Reader.h` and for a writer in `Writer.h`.

The framework also queries each plugin to see whether it implements any system calls. If so, then the plugin should instantiate and return a `SysCallHandler` object when `createSysCallHandler` is called. The interface for system call handlers is declared in `SysCallHandler.h`.

If any setup operations need to be performed, these should be placed within the `init` function, which is called immediately after the plugin library has been loaded. The function receives a reference to an `AnyOption` object, which is the class that performs command-line option processing. The `init` function may add new options to the `AnyOption` class at this point. After all libraries have been loaded, the command-line is processed and then the `setup` function is called for each plugin library. At this point, the command-line option database may be queried.

For example, within the `init` function, a flag may be added:

```
void MyPlugin::init(AnyOption &options) {
  ...
  options.setFlag("foo","An example option");
  ...
}
```

The `setup` function may then query its value:

```
void MyPlugin::setup(AnyOption &optoins) {
  ...
  if (options.getFlag("foo",false)) {
    ...
  }
  ...
}
```

3 An Example

The following is a simple example of a plugin. The writer handles files of type `icnt`. Such a file contains a histogram of all instructions processed during a simulation.

The implementation file is:

```
//
// Copyright (C) 2005 by Freescale Semiconductor Inc.  All rights reserved.
//
// You may distribute under the terms of the Artistic License, as specified in
// the COPYING file.
//

//
// This demonstrates a very simple, example plugin.  All it does it create a
// hash of what instructions were generated, then prints out this hash to a
// file.  The writer file extension type is 'icnt'.
//

#include <iomanip>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdexcept>

#include "helpers/Macros.h"
#include "helpers/BasicTypes.h"
#include "helpers/AnyOption.h"
#include "helpers/stringhash.h"

#include "iss/Writer.h"
#include "iss/Plugin.h"

using namespace std;
using namespace adl;

typedef map<string,unsigned> InstrHash;

// This maintains a histogram of instructions encountered and prints out the
// results at the end of a test to the specified file.
```

```
struct InstrCount : public Writer, public LoggingIface {

  // This just opens the file and makes sure that the operation succeeded.
  InstrCount(const std::string &filename,IssNode &root,const MemoryLog &memlog) :
    Writer(filename,root,memlog),
    _fwidth(0),
    _out(filename.c_str())
  {
    if (!_out) {
      RError("Unable to open file " << filename);
    }
  }

  virtual LoggingIface *register_core(IssCore &core)
  {
    return this;
  }

  // Required by the Writer class: Returns a string identifying the writer's
  // type.
  virtual const char *format_str() const { return "ICNT"; };

  // Add to the histogram and track instruction width so that we can nicely
  // format the output.
  virtual void logInstr(const uint32_t *opc,int num_bytes,const char* name, Disassembler,uint32_t)
  {
    _counts[name]++;
    _fwidth = max(_fwidth,strlen(name));
  }

  struct VComp {
    bool operator()(const InstrHash::value_type *x,const InstrHash::value_type *y)
    {
      return x->second > y->second;
    };
  };

  // This dumps the histogram to the output file.
  virtual void writeResults()
  {
    vector<const InstrHash::value_type *> values;
    ForEach(_counts,iter) {
      values.push_back(&(*iter));
    }
    sort(values.begin(),values.end(),VComp());

    _out << "\n\nInstruction Frequency (Ordered By Frequency):\n"
         << "=============================================\n\n";
    ForEach(values,iter) {
      _out << setw(_fwidth) << right << (*iter)->first << ":  " << (*iter)->second << '\n';
    }

    _counts.clear();
    _fwidth = 0;
  }

  size_t    _fwidth;
  InstrHash _counts;
  ofstream  _out;

};

// This class is the service provider.  For a request for a writer of type
// 'icnt', it returns an instruction-count object.  It also displays a small
```

```
// banner at load-timee so that we know that the plugin was installed.
struct CountPlugin : public Plugin {

  // Respond to a request for a writer.
  virtual Writer *createWriter(IssNode &root,const std::string &type,const std::string &fn,const MemoryL
  {
    if (type == "icnt") {
      return new InstrCount(fn,root,memlog);
    }
    return 0;
  };

  // Called when the plugin is installed.
  virtual void init(AnyOption &options)
  {
    cout << "Loaded the instruction-count plugin.\n";
    options.setFlag("foo","Example plugin flag.");
  };

  // Called after final optiona processing.
  virtual void setup(AnyOption &options,IssNode &root)
  {
    if (options.getFlag("foo",false)) {
      cout << "Option foo was set with core " << root.name() << "\n";
    }
  }

};

static CountPlugin count_plugin;

// Main entry point- returns a pointer to our service-provider object.
extern "C" Plugin *adl_plugin_entry()
{
  return &count_plugin;
}
```

The file is compiled using the command:

```
g++ `$adl/scripts/adl-config --cflags` CountInstrs.C -shared -o count-instrs.so
```

Given a model named `model`, the following command will read in a DAT file called `in4.dat` which contains the simulator initial state, simulate this model, then generate an output file named `in4.out.dat` containing a simulation trace and final state and also create an instruction histogram file named `in4.icnt`:

```
./mod1 in4.dat -o=in4.out.dat -o=in4.icnt --plugin=./count-instrs.so  -trace
```

An example output file is:

```
Instruction Frequency:
======================

    bc:  427
  lwzx:  8
 addis:  5
 mullw:  8
  stwx:  200
  halt:  1
    or:  13
```

```
 mtspr:  2
   stw:  136
   add:  230
  cmpi:  229
rlwinm:  12
   lwz:  98
  addi:  347
```

Generated on: 2018/02/28 15:14:26 MST.