DBToaster Runtime Problem Set 1

Yanif Ahmad {yanif@cs.cornell.edu}

Out date: September 22nd, 2009. Due date: September 30th, 2009

1 DBToaster Runtime Introduction

In this problem set, you will develop a single-threaded runtime for DBToaster that is capable of executing generic C++ code compiled and maintained by the LLVM framework. This assignment will help you gain familiarity with LLVM, as well as provide a basic execution and testing platform for future work on building a generic stream processing oriented runtime for executing, parallelizing and distributing black-box stream programs.

2 Software requirements

For this assignment, you will need to set up the following libraries on your system:

- LLVM 2.5: available from `http://llvm.org`
- Boost 1.4.0: available from `http://boost.org`. Note this is not a requirement but a recommended library to use for (file and network) I/O functionality.

3 LLVM Basics

We recommend you look over the following pieces of documentation to get an overview of the LLVM framework prior to developing the runtime. This includes both introductory LLVM documentation, as well as specific classes and methods from the LLVM API docs.

- LLVM introductory documentation:
  - LLVM overview: `http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf`
  - LLVM toolchain example: `http://llvm.org/docs/GettingStarted.html#tutorial`
  - LLVM common operations: `http://llvm.org/docs/ProgrammersManual.html#common`
- API docs: useful classes/functions: `http://llvm.org/doxygen/`
  - llvm::MemoryBuffer::getFile – reads a file into an in-memory representation for use in LLVM.
  - llvm::ParseBitCodeFile(MemoryBuffer*) – compiles a MemoryBuffer into LLVM bitcode.
  - llvm::Module class represents LLVM bitcode resulting from the compilation of a single .cpp file. Note that the Module class contains several iterators to let you determine symbols and globals present in the bitcode.
  - llvm::sys::DynamicLibrary::LoadLibraryPermanently() – loads a dynamic library (a `.so` file) for use by any module in the virtual machine.

  - llvm::Linker – enables linking of LLVM bitcode to other Modules and dynamically loadable libraries.
  - llvm::Function – LLVM representation of a bitcode function
  - llvm::FunctionType – function type signature representation
  - llvm::GenericValue – LLVM representation of a C-value (i.e. a basic type, array or struct).
  - llvm::ExecutionEngine – LLVM virtual machine engine for running functions.

4 Runtime Requirements

The runtime developed should be a single-threaded runtime, where only one program context/thread will run stream programs. Your runtime should be capable of compiling C++ source files into LLVM bitcode, and then execute the functions present in the resulting bitcode. In addition to source code files, your runtime should be able to load object files that have already been compiled to LLVM bitcode. The functions present in either the source code or precompiled bitcode will take different numbers of arguments, and your runtime should be able to supply arguments to your compiled functions from two sources: a file source and a network source. You may make the following assumptions about the source code file, and function argument input file.

- The source code may include both standard header files present as part of the STL, and other user-defined header files. Specifically the source code does not include third party header files. The path of these other user-defined header files will have to be included in the compilation path when producing LLVM bitcode. Similarly assume the source code links only to the standard C++ library (i.e. `libstdc++.so`).
- The C++ source and object code will have multiple functions to be executed. There will be no `main()` function present in the source or object code.
- The C++ code may have global variables or data structures whose contents should be accessible.

You may use LLVM to perform reflection to extract the list of functions, and data structures and their types. Your runtime should implement the following additional features for usability and testing purposes:

- Implement a file stream to drive execution of C++ code, where the file stream contains function arguments. Assume a different file for each function being invoked in the C++ source code.
- Implement a network stream to feed data into C++ code. Try to develop an abstraction (i.e. a common interface) for file and network streams, to re-use code for executing functions regardless of the type of stream.
- Implement a *viewer* class to write out the contents of the data structures to an output file after every invocation of a function. You should write out the class in a readable format, for example as Boost::Serialization text or XML format.
- Develop a simple test client to send data to your basic runtime and execute the object code that we supply, on the data file that we supply.

For the DBToaster core group, we would like you to demo and perform a brief code walkthrough of your solution in the next DBToaster meeting.

5 Extra Credit

For extra credit, implement a multi-threaded runtime that uses a thread pool (for example the Boost library includes an implementation of a thread pool) to execute the functions available in a module concurrently. You should make the number of threads available for concurrent processing a parameter to your implementation. Your report should include a graph showing processing throughput for an input file as you vary the number of threads. You should choose an appropriate range of thread counts to demonstrate i) threading overhead ii) processing speedup from using multiple cores if you have a multicore machine.

You may make the following assumptions about the functions being executed:

- Independent functions – the functions will not share data structures, enabling any function to be run concurrently with any other function without any need for complicated locking of data structures.