libspe: A Dynamic System Performance Analysis Library

Joel Reardon

January 17, 2009

Abstract

This document consists of the `libspe` section of my thesis, along with more detailed information on the pragmatics of its use.

1 Introduction

This paper describes `libspe`: a dynamic system performance analysis library. It allows for static collecting of timing information and the ability to register any number of *observees*: a reference to some object that is being monitored. Each observee belongs to a *family*; for example each data structure type can be a family and observees in that family are instantiations of that type. Each family has a method that is invoked for observations, and so when an observation is made on that object, `libspe` invokes the corresponding method passing the observee. To allow for dynamic changes in the observation methods, we maintain a dictionary that maps families to their observation methods, and allow this to be easily modified at runtime.

The source code for `libspe`, written in C, is released as a free and open-source project under the BSD license.

2 Overview

2.1 Static Data Collection

A set of static data collection variables are defined at compile time. These correspond to regions of the source code under timing scrutiny, a particular line of code whose execution period is being computed, or any particular variable whose value the operator has decided to record. Each variable is collected under both a family name and a subindex. For example, each buffer has a unique pointer value in memory, and so buffer sizes were recorded under the buffer size family subindexed by the buffer's memory address. The subindex value of zero is used to store either generic, amalgamated, or nonindexed variables.

The memory allocated to store the results during execution is currently configured at compile time. Each variable can either be a linked list of unbounded size, or a fixed-size array. If a variable uses a fixed-size array, then observations are collected by accepting new measurements with a decreasing probability in order to collect a random set of data distributed uniformly on a stream of unknown length. Our Tor node used a fixed-sized array of 1024 observations per non-zero subindex and 4096 observations for the zero index.

2.2 Interaction Socket

When initialized, `libspe` is provided the port number for a local interaction socket. A thread is spawned that accepts localhost connections on that port. All runtime aspects of `libspe` are controlled through this socket. Foremost, static data collection can be toggled, and the cumulative distribution functions for all the static timing variables previously collected can be output to a file. Since observations on data structures are initially disabled, the interaction socket allows the listing of all the observees currently being managed and the enabling or disabling of observation for specific observees or entire families, along with the filenames used for results. Finally, the callback library used to collect data during an experiment can be changed with a command specifying the new callback library.

2.3 Observers

Registration of candidate data structures for observation is made at compile time. Each object registers itself with the `spe` instance when in use, and deregisters itself from the `spe` instance when it is no longer being used. Tor was modified to register objects in their constructors and deregister in their destructors. While generic observations are made in discrete intervals, `spe` can also be forced to make observations at particular places in the program (e.g., observing on a socket before writing or observing on a buffer before an insertion). Each observee is assumed to be in an *off* state when the program begins execution; `libspe` will not generate any data for observers that are *off*, even if they force an observeration. At run-time, through the interaction socket, the operator can turn observees *on* to generate data, either individually or by family. Each enabled observer is given an open file for outputting data, and an observation time period.

Static state is maintained between observations. Each observee has an associated state object that can be used in the observation method; the state is passed alongside the observee when making observations. When an observee is registered, a special invocation of the observer method indicating initialization is performed so that state data can be initialized. Similarly, when an observer is disabled, another special cleanup invocation is performed to signal that the observer must free allocated memory. Thread safety is assured by having each observee acquire a lock before before calling the observation method and release it after the observations method returns.

2.4 Dynamic Callbacks

Dynamic callbacks are used to collect data from data structures at runtime. Since the data the operator may wish to collect may change as the program executes and collected data is examined, `libspe` allows dynamically loaded libraries of observation routines to obviate restarting the instrumented program due to a change in experiment.

The set of callback families must be known prior to execution. Each observee that registers specifies its family (i.e. data structure type) upon registration. Each family is associated with a specific observation method that is invoked in the library. Initially, there is no library and so all families are associated with a `null` function. When the interaction socket loads a new library, each family's associated function symbol is loaded from the new library. Henceforth, all observations will now invoke this new method.

As an example, suppose the operator wants to report the size of a buffer over time. They compile a library with an observation function that takes a buffer as a parameter, and writes its size to a file. The operator connects to the interaction socket, informs `libspe` of its observation library, and enables observations on all buffers. Suppose after determining the sizes of every buffer, they find one buffer which they wish to explore in more detail. The operator then writes a new method to report the contents of the buffer, changes in sizes over time, the memory allocated for the buffer, etc. This new method is compiled into a new libray, and `libspe` is told (via the interaction socket) to use this new library for experimental callbacks henceforth. The operator then enables observations on the single buffer of interest, and `libspe` will use their new method to report more information.

2.5 System Interface

The system interface for `libspe` contains two components: the static API used during the instrumentation of the program, and the runtime interface used to control experimentation while executing.

The instrumented program initializes `libspe` with `initialize()`, which configures data set sizes and the local port used for the interaction socket. Data structures that are to be inspected are registered with the `register()` method. The functions `start_timing()` and `stop_timing()` are placed around code to respectively start and stop the timer. The elapsed time is computed when stopping the clock, and is stored locally using the `data_point()` function. To store a single piece of data, such as the current size of the buffer, one can call `data_point()` directly. Finally, `period()` is used in lieu of `start_timing()` and `stop_timing()` to measure the time elapsed before the program counter returns to the same line of code. It computes the difference in time between now and the time it stores locally. It replaces the stored time with the current time, and adds the computed difference using `data_point()`. Since there is no initially stored value, the first call is ignored but all subsequent calls compute the period properly. Like all the other methods, `period()` is indexed by both a family name and a subindex, allowing for expressive period calculations based on the current system logic.

At runtime, an operator can connect to `libspe` through the interaction socket to control its behaviour. The listening thread will spawn a new thread upon accepting a connection, and the new thread will respond to operator demands sequentially. As mentioned, these include changing the callback libraries, enabling observation, and outputting the static data that has been collected. Resources are shared between the `libspe` static program interface and its dynamic socket interface, and all data access and function calls are threadsafe.

The file main.c shows an example of an instrumented program.

3 Future Work

- Variable Interaction: Allow the operator to set a variable by name through the interaction socket to a particular value, and allow the compiled program to retrieve these values while its running. Toggles would allow the control of logic, so the operator could decide the outcome of a decision block. Right now the hackish way to accomplish this is to have a special toggles_t that contains the variables, and write an experimental callback that sets the values to the desired value, load the new library and enable observations for the toggles_t one time only.

- Interaction GUI: The simple prompt with four-letter commands was clearly just a way to get it done. Much better to have a more interactive GUI that shows all the observees and allows selection of libraries, enabling of observees, and viewing of the resulting data with ease.

- `spe_observer_enable`: Use the interest variable to discriminate against observees that are inactive. Currently its commented out, needs more of a configuration.

- Use a configuration file to load configuration options.