Performance Evaluation of Vortex-compiled Applications

The Cecil Group

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350
cecil@cs.washington.edu

Abstract

This document outlines how to obtain maximal runtime performance from applications compiled with the Vortex compiler. It first describes how we typically use Vortex to produce programs with reasonable runtime performance during day to day application development. Then it discusses several issues that arise when benchmarking Cecil programs for maximal performance and when using the Vortex compiler infrastructure to study the impact of different optimization techniques. This document is intended as a supplement to the Vortex user manual, and assumes some familiarity with it.

1 Maximizing Normal Application Performance

Vortex's default initial compilation mode is non-optimizing compilation (`o0`); to enable optimization the user must set the compiler option* `optimization_level` (by convention, larger values denote more aggressive combinations of optimizations). Typically, this is done by issuing the `o1` (sets `optimization_level` to 1) or `o2` (sets `optimization_level` to 2) commands at the **`Vortex>`** prompt. During application development, we will typically have most of the application's source files already compiled with optimization while those files that are actively being edited/debugged will be compiled without optimization to minimize turnaround time and maximize debuggability. Periodically (over lunch for example) we utilize the `pmakeo2` Vortex command to recompile with optimization all files that are currently unoptimized.

There are a number of compiler options that control what optimizations Vortex performs during optimizing compilation. They default to settings that we have found most appropriate in our daily use of Vortex. Thus, the typical user can simply choose a built-in combination of optimizations (`o0`, `o1`, or `o2`) without needing to fine tune other optimization-related compiler options.

Two (often important) optimizations require additional effort by the user, however. Profile-guided class prediction can be quite effective for some applications, but requires that the user provide profile-derived class distributions to guide the optimization, We first describe how to gather these distributions from programs and then describe how to make them available to Vortex for exploitation.

Unfortunately, gathering profile data is a somewhat tedious process. Suppose we wanted to gather profile data for `my_program.cecil`, which has already been compiled by Vortex using either C or assembly code-generation. The first step is to build an instrumented executable from the Vortex-generated files by typing `make pic` at the Unix prompt (if you have the `pm` utility for spawning parallel C compiles, then you can give `mc` the `-pic` flag). This will produce an executable named `my_program.pic`. To gather the profile data, run the instrumented executable with the additional command line argument `--picstats`

* See the Vortex user manual for a description of the various compiler options and how to set them. A list of all the compiler options, their current values, and a brief description of each option can be obtained by typing `options all` at the `Vortex>` prompt.

on a representative input. The profile data is printed to stdout when the program terminates normal execution, and you need to capture the profile data into a file for later processing, so we typically redirect the program output to a pipe or file. For example,

```
(Unix%) my_program.pic --picstats [other arguments] > my_program.data
```

The raw profile data must be processed before it can be utilized by Vortex, and a script called `call-chain.perl` has been provided to do this. For example,

```
(Unix%) call-chain.perl < my_program.data > my_program.nCCP
```

will format the profile data gathered in the previous step into a profile file called `my_program.nCCP` that can be utilized by Vortex. Finally, we read the profile data into Vortex by saying:

```
Vortex> load_profile "my_program.nCCP"
```

Once profile data has been read into the compiler, it will become part of the persistent program database and will be utilized during all subsequent optimizing compilations unless explicitly flushed.

For the best results, one should iterate this process a couple of times (gather profiles, use them to build an optimized executable, gather new profiles from the optimized executable, and so on), because the call chain context associated with the profile data increases with iteration, thus making the data more useful for optimization. After a few iterations, there should be no more changes in the profile data; a `diff` of the `my_program.nCCP` generated files should indicate when the best profile data has been achieved. Our experience has been that profiles derived from optimized executables are much more effective than those from non-optimized executables, and that several repeated iterations can increase performance by 10-20%, depending on the application.

Specialization is another optimization that relies on the presence of profile data and must be invoked explicitly. After loading profile data in to Vortex, typing `graphs "my_program.cecil"; specialize` at the `Vortex>` prompt will invoke profile-guided method specialization. This optimization increases performance by around 10-15%, again depending on the application. Unfortunately, a specialized application is not suitable for profiling itself, so save specialization for last, once profile iteration has been completed. (In the future, we will try to make specialization better integrated into the rest of the compiler infrastructure.)

Currently specialization and static class prediction are only implemented for Cecil and Java programs.

2 Benchmarking Applications

In its default compilation mode, when compiling Cecil applications Vortex generates code that supports fairly good source level debugging. However, we have not yet spent much time optimizing our debugging support and the simple scheme currently used incurs a fairly hefty runtime cost, often on the order of 30% or 40% in optimized code. Therefore, when benchmarking Cecil applications we disable debugging support by setting the compiler options `debug_support` and `interrupt_checking` to false (e.g., `Vortex> no_debug_support; no_interrupt_checking`). Note that code compiled with and without debugging support cannot be mixed; toggling the `debug_support` option will automatically invalidate all compiled code.

Another thing to be aware of is that, to reduce the costs of gathering profiling data, Vortex does not fully instrument calls that have been statically-bound purely by means of some static analysis (for example, intraprocedural class analysis, class hierarchy analysis or static class prediction). In most cases this does not matter, but if different levels of static analysis are going to be used (for example to measure the effectiveness of various flavors of static class analysis), then it is critical that the profile data to be used in the experiments be gathered from executables compiled with the same level of static analysis. For example, to measure the impact of class hierarchy analysis, one might want to compare the following four combinations of optimizations:

- intra: intraprocedural class analysis
- intra+profile: intraprocedural class analysis + profile-guided class prediction
- intra+CHA: intraprocedural class analysis + class hierarchy analysis
- intra+CHA+profile: intraprocedural class analysis + class hierarchy analysis + profile-guided class prediction.

To do this requires two sets of profile information: a profile of the intra version of the program to build the intra+profile version and a profile of the intra+CHA version to build the intra+CHA+profile version. Future versions of Vortex may support an additional, more expensive level of profiling that instruments all statically-bound calls, but this is not currently implemented. Similarly, as mentioned above, Vortex does not perform enough instrumentation of specialized programs, and the missing information can degrade the quality of the profile data substantially.

If you are going to be doing a non-trivial amount of benchmarking and/or performance evaluation using Vortex, you should invest some time and become familiar with the config family of scripts that we have developed to support these tasks (found in `$VORTEX_HOME/bin/shell`). The file `configs.perl` is included in the rest of the scripts; it defines configuration information and handles command line arguments. `buildConfig` uses Vortex to build executable, `runProgs` measures the resulting exectuables, and `showData` displays experimental results.