How to Use the Vortex Compiler and Environment

The Cecil Group

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

Vortex 3.0
December 9, 1998

Abstract

This document describes how to use the Vortex compilation environment. We begin with a brief overview of the compilation model. We then detail the steps involved in compiling an application, and describe how to use the debugger and the Cecil evaluator. We describe how to set up separately compiled libraries, use non-Cecil front ends, and then finish with some installation and configuration directions.

1 Introduction

The Vortex compiler's whole-program analyses introduce complex intermodule dependencies, rendering the standard header file and `make`-based approach to incremental compilation untenable. Instead, Vortex tracks intermodule dependencies in a program database, enabling it to determine which compiled files must be recompiled after a programming change. The current Vortex compiler keeps this database in memory while running and supports operations to dump the program database to disk and later read the program database back from disk. The file containing the program's database and other information about the state of the compiler is called the program's *checkpoint*.

Given whole-program analysis and optimization, each program's compiled code (including libraries) is tuned to that particular application and consequently it cannot be shared with other applications (with the exception of sharable, separately compiled libraries; see section 5). In general, each application must be compiled into its own directory. We call this directory the application's *gen* directory (short for "generated code"). Vortex places all generated files (except for files that are part of separately compiled libraries) including the final executable and the program's checkpoint in the gen directory.

Vortex compilation occurs in three phases. In Phase Zero, a Vortex front end translates non-Cecil programs into the Vortex RTL intermediate language. In Phase One, Vortex translates a RTL or Cecil source program into either C++ or assembly code. In Phase Two, a C++ compiler or an assembler translates the output of Vortex into object files and an executable. Currently the SPARC is the only architecture with an assembly code back-end.

For an example of a compiler session, see Section 2.2.1.

2 Building Your Program

This section describes how to use Vortex to build an application in stand-alone mode. As discussed in section 5, Vortex also supports compiling applications against a shared pre-compiled library (the Cecil stdlib for example).

Normally, a wrapper script configured once per site sets necessary environment variables before invoking the underlying Vortex executable. See section 8 for details on setting up this wrapper script.

2.1 Gen directories

It is easiest to run the compiler from the directory containing your source files. During the first compile, Vortex creates a subdirectory of the current directory called `gen` which becomes the gen directory for your application. The program's executable is created in this directory. Its name is the name of the application's main file with the suffix removed. For example, if your program is called `mypoint.cecil`, the executable will be `gen/mypoint`.

If you have several applications, just put the sources for each application in their own directory, so each application has its own gen directory. (Section 2.4.3 explains how to adjust the gen directory.)

2.2 Running the compiler

2.2.1 Typical compiler session

Vortex normally runs interactively, reading and executing user commands. The notation `Vortex> some_command` indicates that `some_command` (followed by pressing the return key) is typed at the compiler prompt. Compiler commands appear in **`boldface`**, and `%` is used as a sample Unix shell prompt.

To compile a new program (say, `myprog.cecil`) run a make command which specifies the name of your program: `Vortex>` **`make`** `myprog.cecil`. The **`make`** command parses the file named `myprog.cecil` and any files transitively included by this file, constructs global data structures describing the program (e.g., the object inheritance graph and the table of defined methods), produces Phase-1"compiled" (C++ or assembly) code for the application in the gen directory (which is created if necessary), and then invokes a Phase-2 make* to compile the intermediate files into `.o` files and link the `.o` files into a target executable, `gen/myprog`. The generated executable can be run either from a UNIX prompt or by typing **`run`** at the Vortex prompt.

Once a program has been compiled, further invocations of the **`make`** command (the program name is optional after the first time) incrementally recompile only files that changed (or depend on a changed file).

The **`save`** command writes the program checkpoint to a file named *program*`.db`, where *program* is the name of the application being compiled. The checkpoint contains a snapshot of the application's source code, the interprocedural information computed by the compiler, and recompilation dependencies, as well as the current settings of the Vortex compiler's options. At this point, it is safe to quit Vortex by typing the **`quit`** command. (**`really_quit`** skips asking for confirmation.)

To reload a program's checkpoint file, use the **`load`** *program*`.db` command. The compiler's state is restored to where it was at the point of the previous **`save`**, permitting more incremental development to

* Vortex assumes that `gmake` is available to run Phase-2 compiles. Section 8 describes how to change this assumption.

proceed. If you do not load a checkpoint file, the first **make** command will perform a full non-incremental compilation.

2.2.2 Start-up commands

Upon startup, Vortex searches for a file named `.vortexrc` first in the current directory, then in the user's home directory. Each line is executed as if it were typed at the Vortex prompt. If the line contains the `#` symbol, the part of the line between the first `#` and the end of the line is ignored. For example, if you want Vortex to load the program's checkpoint upon startup, you can create a `.vortexrc` containing the following lines:

```
# put this .vortexrc in the directory where you start the compiler
load myprog.db
```

After `.vortexrc` is processed (if found), any command-line arguments are processed, in order. Each command-line argument is executed as if it were typed at the `Vortex>` prompt. As a special case, if the command line starts with `-F`, no startup file is read; if with `-f` *filename*, file *filename* is used for the startup file's name.

For example, Vortex can compile a program in batch mode when invoked as follows:

```
% vortex "load myprog.db" make save really_quit
```

2.2.3 Compiler commands

This section describes the major Vortex compiler commands. The following notation is used:

- [*phrase*] -- *phrase* is optional
- *phrase1* | *phrase2* -- either *phrase1* or *phrase2*
- **...** -- the last parameter may be repeated

More commands are described in Sections 2.3, 2.4, and 2.5. This manual describes only the more common commands; a complete list of commands and their brief descriptions can be obtained by using the **help** command. Multiple commands separated by semicolons can be given on the same line.

- **help** [*helptopic*] -- display a list of compiler commands with brief descriptions (related to *helptopic* if specified; possible *helptopic*s are: `make`, `options`, `browse`, `typechecker`, `advanced`, and `startup`)
- **make** [*progname.ext*] -- set the current program to be *progname.ext* if specified, then compile the current program. The executable called *progname* is produced in the gen directory. The *.ext* extension may be omitted, in which case an input-language-specific extension (e.g. `.cecil` for Cecil programs, `.rtl` for Java programs, `.o` for C++ programs, and `.strtl` for Smalltalk programs) is appended.*

  There are a number of variations of the **make** command supporting parallel Phase Two compilation, Phase One or Phase Two only compilation, and/or optimized compilation, derived from the regular expression: [**p**]**make**[**1**|**2**][**o1**|**o2**] [*progname.ext*]

* If the filename doesn't already have the default extension, then the filename with the default extension appended is searched for first. If this filename isn't located in the current source search path (see section 2.4.3 about the `source_paths` option), then the filename as given by the user is searched for. For example, **lang** `Cecil;` **make** `foo.bar` first searches for a file named `foo.bar.cecil`, and if this fails then searches for a file named `foo.bar`.

- **p** -- Uses the `pm` script to perform the Phase Two compiles in parallel. (See section 8 for information on how to configure the list of machine names used by Vortex for parallel compiles.)
- [**1**|**2**] -- Only do Phase One or Phase Two compilation.
- [**o1**|**o2**] -- Recompile all file that have been compiled with a lower optimization level than that specified in command (see section 2.3 for a description of possible optimization levels and some examples of **make** commands).

- **info** -- report the status (up-to-date, out-of-date, need-to-parse, need-to-codegen) of all of the files making up the current program
- **save** [*filename*] -- save the compiler state to a checkpoint file *filename*, if given, or *curprog*`.db` otherwise, where *curprog.ext* is the name of the current program
- **load** *filename* -- load the checkpoint from file *filename*
- **quit**, `^D` -- exit Vortex. You will be prompted to make sure you really wanted to quit. Note that quitting the compiler does not save the compiler state. If you want to save the compiler state before quitting, you must use the **save** command and then quit. **really_quit** can be used to avoid the confirmation message.
- **really_quit** -- like **quit**, but does not require interactive user confirmation

A string typed at the Vortex prompt is interpreted as follows. If it starts with a word that matches one of the compiler commands, such as **make**, that command is executed. Otherwise, if the first word matches an option name, such as `optimize`, the value of the option is displayed or modified (section 2.4 describes options). Otherwise, the string is evaluated as a Cecil expression, statement, or declaration (section 4 describes the evaluator). If the evaluator does not recognize the string as a valid Cecil construct, it reports an error.

2.2.4 Typechecking

Vortex includes a typechecker for Cecil. (Programs in languages other than Cecil are typechecked by their corresponding front-ends.) The typechecker has to be invoked explicitly by the **typecheck** or **fulltypecheck** commands; it isn't run when a **make** command is issued. **fulltypecheck** rechecks the whole program for type errors, while **typecheck** only rechecks code in modified files. **typecheck** is faster for small program changes, but it can miss errors that arise in files that were not touched.

The current typechecker implementation does not perform any "implementation-side" typechecking (checking whether signatures are fully and unambiguously implemented), but does perform most other, "client-side" typechecking. It also includes a few known bugs and weaknesses, where it may report some type errors unnecessarily.

2.3 Controlling the compiler configuration

Vortex's behavior is controlled by configuration parameters and options; the primary distinction between them is that options are set and queried by a regular set of commands, while manipulation of configuration parameters is currently more ad hoc. This section describes configuration parameters and a few important options. Section 2.4 describes options and how to set or display them in more detail.

Vortex displays its configuration upon startup; you can also request it with the **config** command. The configuration is part of the compiler state saved in a checkpoint.

Current program. At each moment, Vortex works with a single program. If the program consists of one file, this file is the current program. If the program consists of multiple files, the current program is the file from which the others are included, possibly indirectly. The current program is set by any of the make commands, for example,

- **`make`** *progname.ext* -- set current program to *progname.ext* and compile it

No current program is assumed upon Vortex startup. See Section 2.2.3 for a detailed description of **`make`**.

Vortex can compile multiple languages. Cecil is the most stable input language, but Java, C++, Modula-3, and ParcPlace Smalltalk front-ends are also available, with an IBM Smalltalk front-end under development. To set the language being compiled, use the **`lang`** command:

- **`lang`** `(Cecil|Java|C++|M3|ParcPlaceSmalltalk|IBMSmalltalk)` -- change the current accepted language
- **`lang`** -- display the current accepted language

Regardless of the current language, the set of compiler commands and options remains the same. Only Cecil expressions are recognized by the evaluator (since they are evaluated in the context of the Vortex compiler's implementation, which is written in Cecil). Filename extensions have no influence on the current language, nor is any specific extension enforced, although each individual language has its own default extension.

Generated language. Vortex generates either C++ or assembly code. By default, it produces code appropriate for the architecture/OS combination on which it is being run.

- **`gen`** `(C|C_32|C_64|asm)` -- generate C++ or assembly code. The C_32 and C_64 flags can be used to control whether the generated C code assumes 32 or 64 bit pointers. (Assembly code generation is currently only supported on SPARC architectures for Cecil, Java, and Smalltalk.)
- **`gen`** `(sun4|solaris)` -- select the target OS for assembly codegen.
- **`gen`** -- display the current generated language and target OS (for assembly codegen).

The `gen` commands can be used to enable cross-compilation. For example, to use a `sun4` vortex executable to produce an executable that would run on an `alpha`, one would issue a **`gen C_64`** command, followed by a **`make1`** command at the `Vortex>` prompt. Phase two compilation could then be done on an `alpha` using either `make` or `pm`. `pm` takes an optional `-arch <archname>` command line argument that overrides its default assumption of compiling for the architecture/OS on which it is invoked.

Cecil standard library. The Cecil language includes no built-in data or control structures. When compiling a Cecil program, Vortex can automatically include the *standard library*, a selection of standard data and control structures (e.g. numbers, loops, arrays) plus the Cecil evaluator, which facilitates debugging the program. Alternately, you may select a subset of the standard library, or no library at all.

There are four levels of incorporating the Cecil standard library (described in more detail in a separate document), depending on the top library file that is included in the program (including a file causes other files referenced in it to be included, too). The levels are determined by compiler commands as follows:

- **`nostdlib`** -- no files are automatically included by Vortex, so no data and control structures are implicitly available.
- **`smallstdlib`** -- `prelude.small.cecil` is automatically included. It provides only the core standard data structures. Note: the smallstdlib is incomplete (in particular it excludes big_ints and

streams), thus a number of warning messages (which can be ignored) will be generated when compiling the smallstdlib's files.

- **noevalstdlib** -- `prelude.noeval.cecil` is included. It provides all the standard data structures, including those provided by smallstdlib, but excludes the Cecil evaluator.
- **stdlib** (default) -- `prelude.cecil` is included, providing the Cecil evaluator in addition to the standard data structures

These commands are processed in addition to any files included by the Cecil program. Choosing **nostdlib** and explicitly including `prelude.cecil` from one of the files comprising a program is equivalent to choosing **stdlib** when compiling that program (even if it contains no explicit includes).

(When compiling Java, the file `stdlib.rtl` is included as the default standard library; **nostdlib** can be used to disable this default library. Other languages have no standard libraries.)

Optimization level. When debugging code, it is convenient to compile it without optimizations both to reduce turnaround time and to make it easier to use the Cecil evaluator and debugger (discussed in sections 3 and 4). We typically make some source changes, do a series of incremental, non-optimizing compilations while debugging, and finally perform an optimizing compilation once things seem to work. Optimization speeds up applications by roughly an order of magnitude, so it generally is a good idea to optimize a program before running it on a large input set.

Which optimizations are performed depends on the value of the integer option `optimization_level`; larger values of `optimization_level` correspond to more aggressive combinations of optimizations. Its default value is 0 (perform no optimizations). It may be set using the following commands:

Commands	Value of optimization_level	Impact
`o0` or `no_optimize`	0	Perform no optimizations
`o1` or `optimize`	1	Only perform a few highly profitable optimizations (class analysis, splitting, class hierarchy analysis, class prediction, closure delaying, and inlining)
`o2` or `full_optimize`	2	Augment `o1` with useful, but smaller-impact optimizations such as CSE

Since separately compiled libraries (such as the precompiled standard library that comes with the distribution) are compiled with no optimization, you need to disable separate compilation (i.e., enable whole-program optimization) to achieve higher performance. This is controlled by the option `use_shared_libraries`.

- **set** `use_shared_libraries (true|false)` -- allow sharing of compiled library code (if true)/make a specialized version of all library code in the application's gen directory (if false; default value is true)

Be prepared for a long compilation time when you first set `use_shared_libraries` to `false`, since all the library files will need to be recompiled. You may want to switch to a smaller standard library before specializing library code.

In a large program, often only a few files are being debugged at a time. In such a case, it is reasonable to optimize the other files. This is achieved by first compiling everything with optimization (`Vortex>` **`set use_shared_libraries false; o2; make`**) and then turning optimization off (`Vortex>` **`o0`**). After these actions, every file that is modified will be recompiled without optimization (thus making debugging easier and reducing turnaround time). The command `Vortex>` **`makeo2`** will then recompile unoptimized files with optimization, bringing the current program up to full optimization. **`makeo1`** and **`makeo2`** set the `optimization_level` option for only the immediately succeeding compile. It is reasonable to keep `optimization_level` set to 0, but periodically use **`makeo1`** or **`makeo2`** to bring things up to full speed.

2.4 Compiler options

2.4.1 General commands

Compiler options control various aspects of compilation, such as optimization level, verboseness of messages, gen directory, etc. The following commands are used to display and modify options:

- **`options`** -- display the more interesting options
- **`options`** *topic* -- display options on a given topic (for example, `config`, `directories`, etc.)
- **`options`** `topics` -- display available option topics
- **`show`** *optname*`...` -- display option *optname* (multiple option names are allowed)
- **`set`** *optname* *optvalue* -- set option *optname* to value *optvalue*

For convenience, `Vortex>` *optname* is a shortcut for **`show`** *optname*; `Vortex>` *optname optvalue* is short for **`set`** *optname optvalue*.

Each option receives a default value upon start-up. The current option values are by default part of the compiler state saved in a checkpoint. For more options and option-related commands, see:

- **`help`** `options` -- display information about options

2.4.2 Kinds of options

Vortex has four classes of options:

Boolean options (e.g., `optimize`, `show_phases`). They can have `true` and `false` as values.

Integer options (e.g., `opt_verbosity_level`). Their values are integers.

String options (e.g., `gen_dir`). Their values are strings. When changing their values, the double quotes around the new string value (that would be required by the Cecil syntax) should be omitted. To set a string option to the empty string, use **`set`** *optname* (with no explicit option value, i.e., the empty string).

String list options (e.g., `source_paths`). Their values are lists of strings. When changing the value of a string list option, the new value is the list of words that are separated by spaces in the user's command. As with string options, the quotes around the strings should be omitted, and **`set`** *optname* sets an option to the empty list.

String and string list options allow **`prepend`** and **`append`** operations:

- `(`**`prepend`**`|`**`append`**`)` *str_opt_name* *value*`...` -- prepend/append value(s) to string or string list option *str_opt_name*

2.4.3 Frequently used options

Here we describe some of the frequently used options. A complete list of options is given by the command **`options`** `all`.

`source_paths` -- the set of directories where the source files are searched. With current implementation, once Vortex finds a file in some directory, it will always look for that file in that same directory during subsequent incremental recompilations.

`gen_dir` -- the gen directory for the application, absolute or relative to the directory where Vortex is run

`debug_support` -- determines availability of source-level debugging support in the compiled program. A program compiled without debugging support runs faster, but does not provide the features described in Section 3.

`show_directory_in_prompt` -- determines how many directory levels are displayed as part of the compiler prompt. This option allows one to distinguish more easily between Vortex processes compiling different programs.

`loud_compile` -- controls display of individual methods being compiled

`show_phases` -- controls display of individual optimization passes (with optimization-specific characters) performed for each method being compiled.

2.5 Browsing the program's inheritance hierarchy, methods, and fields

Several Vortex commands are available to examine the inheritance hierarchy, methods, and fields of the program currently being compiled. To use them, it is necessary first to run a make command (like **`make`** or **`graphs`**) on the program, or, instead, load a checkpoint that was saved after a make command had been run.

- **`parents`** *objectspec* -- display (immediate) parents of the object specified by *objectspec* (*objectspec* is the object name followed by the number of its type parameters, if non-zero, e.g., `int` or `list 1`)
- **`all_parents`** *objectspec* -- same, but include indirect ancestors
- **`children`** *objectspec* -- display (immediate) children of the object specified by *objectspec* (except for descendants of predicate objects)
- **`all_children`** *objectspec* -- same, but include indirect descendants
- **`methods`** *objectspec* -- display methods and fields where the specified object is the specializer of at least one formal
- **`all_methods`** *objectspec* -- same, but the specified object may be either the specializer or inherit from the specializer (the specializer `any` doesn't count)
- `[`**`all_`**`]`**`fields`** *objectspec* -- same, but display only fields
- **`methdefs`** *methodname* `[[`*nparams*`]` *nargs*`]` -- display all implementations of the message with the given name, number of type parameters and number of formal arguments (0s by default)
- **`matchmeths`** *name* -- display all message names that include string *name*

Two options affect the selection of displayed objects, methods, or fields

- `browse_predicate_objects` - when false, don't include predicate objects

- `browse_private_methods` - when false, don't include private methods or fields

Typing **`help`** `browse` will print a summary of these commands.

2.6 Gathering and applying profile information

Vortex can exploit dynamic profile information to generate highly optimized code, ranging from 50% to 500% faster than without profile data. In order to reap the benefits of this optimization, profile information must be gathered and then exploited. Gathering profile information involves the following steps:

- Build an instrumented executable. This requires a full phase two re-compilation. It does not require a phase one(Vortex) re-compilation. You can build an instrumented executable by typing `make pic` or `pm -pic` at the Unix shell prompt in your gen directory.* If the program is named `foo.cecil` then the instrumented executable is named `foo.pic`. An instrumented executable is typically 10-50% slower than a normal executable.
- Run the instrumented executable to gather profile information. The most important aspect of this step is running the program on a representative input set. The program should be started with the `--picstats` command line option.† Profile data is dumped to standard output when the program terminates. Generally, you want to redirect the output of the program into a file. For example, you could gather profile information for the Vortex compiler using the following command:

  ```
  % compiler.pic --picstats "optimize; make towers.tst" > c.pic
  ```

  A more sophisticated example is `${VORTEX_HOME}/Cecil/bin/shell/make_profile`, the script which gathers profile data for Vortex itself.
- Process the profile output into the proper form. Several perl scripts in `${VORTEX_HOME}/bin/shell` do this processing. The most useful form of profile data is n-CCP (call-chain profile data) which can be produced in the following manner (assuming `${VORTEX_HOME}/bin/shell` is in your path):

  ```
  % call-chain.perl < c.pic > c.nCCP
  ```

After profile data has been gathered, exploiting it is fairly easy. A file containing processed profile data can be read into Vortex by using the **`load_profile`** command, e.g., `Vortex>` **`load_profile`** `c.nCCP`. Once profile data has been read in, it is automatically exploited during optimizing compilations. Profile data is saved in the program checkpoint, so it can be utilized across many compilation sessions without having to be read in each time. It also is reasonably robust in the face of small program changes, so profile data from an older version of the program can continue to be used for modified versions of the program.

3 Using the Cecil Debugger

Most features of the Cecil debugger require the application being debugged to have been compiled with debugging support (the `debug_support` compiler option must have been set to true, which is its default setting). There are three main ways to enter the debugger. First, run-time errors like message-not-understood invoke the debugger in the context of the error. Second, typing ctrl-C interrupts a program compiled with debugging and invokes the debugger (typing ctrl-C kills a program compiled without debugging support). Third, the Cecil `breakpoint` function inserts an explicit debugger entry point. For example,

* To get these and other utility programs for manipulating profiling information into your path, you should source the `${VORTEX_HOME}/bin/shell/vortex.cshrc` file in your `~/.cshrc`.

† Command line arguments that start with `--` are interpreted by the Cecil runtime system and are not passed on to the user program. They may appear anywhere on the command line. Passing `--help` as a command line argument to a Vortex-compiled executable will cause it to list the recognized run-time system command line arguments.

```
if(x > 5, { breakpoint(); });
```

invokes the debugger if `x` is greater than 5 at this point in program execution. In the Vortex source code we often embed breakpoints guarded by compiler integer options into key routines; this allows us to simply set the integer option to a “high” enough level and easily enter the debugger at critical points when debugging that piece of the program. Alternatively, the `assert` and `error` functions can be used as defensive programming measures to invoke the debugger whenever unexpected situations arise.

When you first enter the debugger, you will see something like:

```
Current stack frame:
# 0 run (t: <anon/DeviceTaskRec/: 0x183f55>,
         work: <anon/Packet/: 0x15e859>), richards.cecil:153
debug>
```

The stack frame description tells you the method being executed when the debugger was invoked, and the values and types of its formal parameters and local variables. In the example above, we are in the `run` method which has two arguments and no local variables. Variable `work` contains an object which is an anonymous (not specifically named) descendant of `Packet`. If a message-not-understood error invoked the debugger, then the message that was not understood and its arguments appear above the stack frame.

3.1 display, display dyn, where

Typing **`display`** at the `debug>` prompt displays a view of the program stack, similar to typing `where` in gdb or dbx (**`where`** is available as a synonym for **`display`**). The default display option is to display the *lexical* chain, which is a subset of the *dynamic* chain. Since all control structures in Cecil are user-defined, you rarely want to see the complete (dynamic) chain of stack frames. The lexical chain is intended to hide most of the stack frames that are introduced by user-defined control structures. The lexical chain is built by starting at the top-most stack frame and tracing up the program’s stack. If the current stack frame is a closure stack frame, then its lexical parent is the next stack frame displayed. If the current stack frame is a method stack frame, then its dynamic parent is displayed next. We build the lexical chain by applying these rules to select which stack frames from the current program stack should be displayed. You can see the complete dynamic chain by typing **`display`** `dyn` at the `debug>` prompt.

For example,

```
debug> display
Lexical call stack:
#16 run_richards (count: 1), richards.cecil:397
#14 eval ({...}) in run_richards, richards.cecil:398
#10 eval ({...},
          i: 0) in run_richards, richards.cecil:399
# 9 richards (), richards.cecil:290
# 8 schedule (), richards.cecil:<unknown line>
# 4 eval ({...}) in schedule, richards.cecil:304
# 2 eval ({...}) in schedule, richards.cecil:309
# 1 runTask (tcb: <anon/DeviceTaskRec/: 0x183f55>), richards.cecil:126
# 0 run (t: <anon/DeviceTaskRec/: 0x183f55>,
         work: <anon/Packet/: 0x15e859>), richards.cecil:153
debug>
debug> display dyn
Dynamic call stack:
#16 run_richards (count: 1), richards.cecil:397
```

```
#15 time (closure: <closure 0x171011>), system.cecil:72
#14 eval ({...}) in run_richards, richards.cecil:398
#13 do (count: 1,
        c: <closure 0x171009>), small_int.cecil:<unknown line>
#12 loop (c: DelayedComputation), closure.cecil:351
#11 eval ({...}) in do, small_int.cecil:353
#10 eval ({...},
          i: 0) in run_richards, richards.cecil:399
# 9 richards (), richards.cecil:290
# 8 schedule (), richards.cecil:<unknown line>
# 7 while_true (cond: DelayedComputation,
                c: DelayedComputation), closure.cecil:<unknown line>
# 6 loop (c: DelayedComputation), closure.cecil:19
# 5 eval ({...}) in while_true, closure.cecil:<unknown line>
# 4 eval ({...}) in schedule, richards.cecil:304
# 3 if (_anon_0: false,
        tc: DelayedComputation,
        fc: DelayedComputation), boolean.cecil:<unknown line>
# 2 eval ({...}) in schedule, richards.cecil:309
# 1 runTask (tcb: <anon/DeviceTaskRec/: 0x183f55>), richards.cecil:126
# 0 run (t: <anon/DeviceTaskRec/: 0x183f55>,
         work: <anon/Packet/: 0x15e859>), richards.cecil:153
debug>
```

Each stack frame is numbered, with #0 being the most recent (topmost) stack frame. Each stack frame shows the method invoked for that stack frame (methods named `eval` are often the bodies of closures), the names and values of the formal parameters of the method, the lexically enclosing method (in the case of closure eval methods), and the file name and line number where execution is suspended.

`display` *n1* displays stack frames starting from number *n1*. **`display`** *n1 n2* displays stack frames from *n1* to *n2*, inclusive. **`display`** `dyn` versions are also available.

`display` prints out a short description of each stack frame. To see a fuller description, including local variable values, use the **`show`** [*n*] command to display frame *n*:

```
debug> show 15
#15 time (closure: <closure 0x171011>), system.cecil:72
  Locals: start: 900,
          end: Uninitialized
debug>
```

`show` by itself shows the current frame; entering a blank line acts like **`show`**.

In the presence of optimization, some line numbers are not available, but all stack frames, even those inlined away through optimization, appear in the stack trace. Also, optimization can cause certain values to not be computed. In particular, often closures are inlined away; their values print out as `DelayedComputation` in the stack trace. Finally, some variables may not be initialized at the point the stack trace is printed, and these variables print out as `Uninitialized`.

3.2 up, down, goto

You can move around in the stack by using the **`up`**, **`down`**, and **`goto`** commands to change the current stack frame.

- **`up`** - Move up to the calling stack frame.

- **`uplex`** - Move up to the lexically enclosing stack frame, or the caller frame if already a method frame.
- **`down`** - Move down to the callee stack frame.
- **`downlex`** - Move down to the lexically enclosed callee stack frame.
- **`goto`** *n* - Goto stack frame *n*.

Moving to a particular frame implicitly **`show`**s the target frame.

3.3 restart, recover

The **`restart`** command restarts program execution from the beginning. It's useful when you find a bug, fix it by redefining the method (see section 4), and then want to test your bug fix. No re-initialization of global variables or named concrete objects takes place.

Some programs, such as the Vortex compiler, install explicit error handling wrappers around sections of code. For example, the compiler wraps execution of compilation commands in an error handler, so that message-not-understood errors during compilation return you to the `Vortex>` prompt, rather than abort the Vortex program. The **`recover`** command returns to the last point in the program where an error handler was installed. This is useful if you've identified a problem, installed a fix, and want to return to the error handling level rather than continuing program execution.

3.4 print

The **`print`** command prints, in a very low-level way, the object contained in any variable visible through lexical scoping from the current stack frame. For example,

```
debug> goto 0
# 0 run (t: <anon/DeviceTaskRec/: 0x183f55>,
/Packet/: 0x15e859>), richards.cecil:153
debug> print t
<anon/DeviceTaskRec/: 0x183f55>
  map: CecilMap 0x119978, id: 203 (anon/DeviceTaskRec/)
8 fields:
        packetPending@TaskState, offset 0: false
        taskWaiting@TaskState, offset 1: false
        taskHolding@TaskState, offset 2: false
        link@TaskControlBlock, offset 3: <anon/DeviceTaskRec/: 0x183f79>
        ident@TaskControlBlock, offset 4: 6
        priority@TaskControlBlock, offset 5: 5000
        input@TaskControlBlock, offset 6: 0
        pending@DeviceTaskRec, offset 7: 0
```

It is also possible to print the object addressed by any of the hex numbers printed out by a stack dump or a **`print`** command (other than the address of the internal CecilMap data structure). Note that if there is not an object that starts at the requested address, the debugger is likely to crash. This second usage of **`print`** can be used to trace through data structures. For example, we can look at the object stored in the `link` field of the `t` formal parameter:

```
debug> print 0x183f79
<anon/DeviceTaskRec/: 0x183f79>
  map: CecilMap 0x119978, id: 203 (anon/DeviceTaskRec/)
8 fields:
        packetPending@TaskState, offset 0: true
        taskWaiting@TaskState, offset 1: false
```

```
        taskHolding@TaskState, offset 2: true
        link@TaskControlBlock, offset 3: <anon/HandlerTaskRec/: 0x17b5f1>
        ident@TaskControlBlock, offset 4: 5
        priority@TaskControlBlock, offset 5: 4000
        input@TaskControlBlock, offset 6: <anon/Packet/: 0x15e8e9>
        pending@DeviceTaskRec, offset 7: <anon/Packet/: 0x15e8b9>

debug>
```

3.5 Breakpoints

Simple breakpoints can be set using the **`break`** command. **`break`** *name* sets a breakpoint on entry to any of the methods named *name*. (For those methods that are called through dynamic dispatching; see section 4 for more info.) **`break`** by itself lists the current set of breakpoints. Individual breakpoints can be disabled and enabled using the **`disable`** and **`enable`** commands. **`clear`** removes all breakpoints.

`trace` prints a line for every dynamically dispatched message sent, and **`step`** continues execution until the next message is sent. Since messages are sent very frequently, these operations are of limited utility. It would be nice to support flavors of **`step`** that continue execution until a send returns (analogous to `next` in gdb), perhaps with variations that reenable stepping when a nested closure is invoked (to support stepping into a control structure's closure argument).

There is no way to set a breakpoint at a particular line of a Cecil program. To do this, edit the method directly to include a `breakpoint()` call at the desired place, and either recompile and rerun (slow) or install the edited method directly using the evaluator (see below) (fast, but doesn't always work in the presence of optimization).

3.6 Evaluating expressions

If you are debugging a program that includes the Cecil evaluator (such as any program compiled with the full standard library), then any input that is not recognized as a debugger command is evaluated in the context of the current active stack frame. For example, if `x` is a list of `HandlerTaskRec`, you could find all the tasks of priority >2000 that are currently elements of `x` by doing the following:

```
debug> x.do(&(y){ if (y.priority > 2000, { y.print_line; }); })
```

If the result of evaluation is non-`void`, then the `print_string` message is sent to the result to compute a user-defined printable representation of the result.

Section 4 discusses the evaluator in more detail.

3.7 Exiting the debugger

There are two major ways to exit from the debugger. Typing **`quit`** or **`really_quit`** terminates program execution and returns you to the Unix shell prompt. Typing **`cont`** or ctrl-D exits the debugger and attempts to continue running the program. Note that if the debugger was entered due to a message-not-understood error and there is no Cecil error handler installed, then **`cont`** will have the same effect as **`really_quit`**.

3.8 Help

Typing **`help`** at the `debug>` prompt displays a short description of the available debugger commands.

4 Using the Cecil Evaluator

The Cecil evaluator functions like a Lisp read-eval-print loop. You can type Cecil code to be evaluated at the `Vortex>` and `debug>` prompts; there also is a stand-alone Cecil read-eval-print program named `cecil` that can be run. The Cecil code is evaluated and any results are displayed. New global variables, objects, and fields can be declared, and new methods can be added and existing methods can be replaced by entering method declarations to the evaluator. Expressions and declarations can span multiple lines, as long as open brackets, parentheses, or braces remain unmatched at line boundaries. For speed, the evaluator invokes precompiled code wherever possible, and interprets expressions otherwise.

Predicate object and precedence declarations are not currently supported by the evaluator, and typechecking is not performed (type and signature declarations are ignored). Top-level `let` declarations made in the evaluator create global variables available to the evaluator and interpreted methods (but not the pre-compiled code) in the same run of the program. (As with regular Cecil programs, module declarations are parsed but ignored.)

The evaluator is extremely useful for debugging programs. The ability to redefine methods is especially helpful when debugging, since a corrected method can simply be typed into the program and tested without having to wait for the program to be recompiled. However, there are some situations in which it does not work. Interpreted methods are invoked only through dynamically dispatched message sends. Therefore, if a message send is either inlined or statically bound (and thus does not go through the dynamic dispatching mechanism), then any new or replacing interpreted methods won't be invoked at that call site. Similarly, breakpoints are implemented only for methods called through dynamically dispatched call sites.

With some knowledge of the optimizations performed by the compiler, it is possible to predict whether a particular call site will be dynamically dispatched, and thus a call site at which an interpreted method could be invoked. However, it is likely that a casual user will not want to figure this out. A simple rule is that if a call site is located in a file which was compiled without optimization, then the call site will be dynamically dispatched and thus interpreted methods will perform as expected. Alternatively, if a file includes the `(**debug**)` pragma, then all methods in that file can be replaced by an interpreted method (i.e., all calls to the methods defined in the file will be forced to be dynamically dispatched). Individual methods can also be annotated `(**debug**)` as follows:

```
method testing(x@:int, y:string) (** debug **) { method_body }
```

Interpreted methods only persist for as long as the program is running. If you add a method to a program, and then exit the program and restart it, the new method will not be present. Interpreted methods are not saved in Vortex checkpoint files.

5 Libraries and Separate Compilation

For best performance of the generated code, Vortex can compile and optimize library code for the client application. However, this approach precludes sharing compiled code for libraries across applications. For simple applications, sometimes best performance is not required. To speed compilation and reduce disk space usage, Vortex can compile libraries in a special mode that makes the generated code sharable across applications. Because of inter-file dependencies created by optimizations, and to enable applications to add methods and fields and children to objects defined in separately compiled libraries, these libraries are compiled without optimization. The generated library code is put in a separate directory. By default, Vortex compiles libraries separately. For best performance, libraries can be specialized to the client application by disabling the `use_shared_libraries` option.

5.1 Defining a library

A library is a source file that contains a `library` pragma, plus all files included from this root library file that do not have their own `library` pragmas. The `library` pragma specifies the name of the library, e.g. `(**library("my-lib")**)`. The only restriction on a library name is that it should be a legal UNIX directory name. The Cecil standard library is currently broken down into a core `smallstdlib`, a larger `noevalstdlib`, and the complete `stdlib`.

5.2 Compiling libraries

By default, Vortex assumes separate compilation of libraries. In this mode, only non-library code is compiled into the gen directory (i.e., those files that are not in any library). Code for files in a library, say, `my-lib`, is compiled into the subdirectory *my-lib* of the directory determined by the string option **`libs_dir`**, e.g., `${VORTEX_HOME}/Cecil/lib`.

When compiling a program, Vortex automatically checks if the libraries the program includes are up-to-date, and if not, recompiles them. The first time a library is compiled, Vortex creates the appropriate directory for its compiled code.

5.3 Disabling separate compilation

To turn off separate compilation, set the `use_shared_libraries` option to `false`. When `use_shared_libraries` is false, Vortex acts as if no `library` pragmas were in the program, compiling and optimizing library files into the same directory as application files.

5.4 Implementation limitations

The current implementation of separate compilation in Vortex imposes two minor restrictions on how the library objects and methods can be extended by an application. In particular, applications compiled with `use_shared_libraries` set to `true` (the default value) may not extend a library by either:

- defining a global variable that is used, but not defined, in the library, or
- adding a predicate child to an object defined in the library.

However, since dependencies are not maintained for library code when compiling in library mode, violations of these restrictions will not be automatically caught by Vortex and may result in unexpected behavior. The only safe way to allow an application to extend a library in one of these two forbidden manners is to disable separate compilation for the application (by setting `use_shared_libraries` to `false`) and recompiling with **`make`**.

Vortex does not provide a way for a programmer to modify some of the library files while still using the rest of the precompiled libraries. If you want to have your own file supercede the file with the same name in the library, you need to disable separate compilation (see Section 5.3).

6 Utilities

The directories `${VORTEX_HOME}/bin` and `${VORTEX_HOME}/Cecil/bin` contain utilities that support Vortex application development. Some of these utilities can be invoked from within Vortex, which is responsible for ensuring that the environment is set up correctly. To use these programs at a Unix prompt, source `${VORTEX_HOME/bin/shell/vortex.cshrc` from your `~/.cshrc` to define the appropriate environment variables and add the appropriate directories to your path. Some of the most useful utilities are:

- Phase Two compilation tools: Phase two compilations can be completed in parallel across a number of machines. Parallel compiles are synchronized via lock files (`_foo.cecil.c.lock` is a lock file for `foo.cecil.c`) that are created and removed in the gen directory during phase 2 compilation. The scripts `pm`, `pmwait`, and `pzap` are used to drive parallel compilation. To immediately spawn parallel compiles, use `pm`; `pmwait` waits until the current phase 1 compilation completes (time stamp on `Makefile` in the gen directory changes) and then spawns the parallel compile jobs; `pzap` kills the spawned compile jobs.
- Profile-guided class prediction: The scripts `call-chain.perl`, `call-site.perl`, and `summary.perl` all invoke `pic-filter.perl` to format raw profile data into the form expected by Vortex (see section 2.6).
- An Emacs mode for editing Cecil programs appears in `${VORTEX_HOME}/Cecil/bin/shell/cecil-mode.el`.
- The `run-vortex` script invokes Vortex after modifying the environment. It enables “casual” Cecil programmers to run Vortex without having to modify their environments or explicitly source `vortex.cshrc`.

7 Non-Cecil Front Ends

To use Vortex on languages other than Cecil, see the `README` files for each of the desired languages:

- `${VORTEX_HOME}/Java/README` for compiling Java programs
- `${VORTEX_HOME}/C++/README` for compiling C++ programs
- `${VORTEX_HOME}/Smalltalk/parcplace/README` for compiling ParcPlace Smalltalk programs
- `${VORTEX_HOME}/Smalltalk/ibm/README` for compiling IBM Smalltalk programs
- `${VORTEX_HOME}/M3/README` for compiling Modula-3 programs

8 Installation

To install and configure Vortex at your site, follow the instructions on the Vortex release web page at:

`http://www.cs.washington.edu/research/projects/cecil/www/Release`

or in the INSTALL file availble by anonymous ftp from:

`hobbes.cs.washington.edu/pub/cecil/INSTALL`

The installation process creates a sharable, baseline tree for Vortex and any languages to be compiled by Vortex (e.g. Cecil). If only one person is going to use the Vortex installation, then it is probably simplest to place your source and gen directories right in the baseline tree. If more than one user will be writing Cecil programs or otherwise modifying the baseline tree, then they will need their own copies of any Cecil source code they plan to modify. If several people are going to be making substantial changes to Cecil compiler/stdlib code you’ll probably want to set up some kind of revision control system. We use CVS, available via ftp from `ftp://prep.ai.mit.edu/pub/gnu/`.