Embedded Debugging of C/C++ Plugins and Extension Modules

Jing Cao

Department of Computer Science
University of Chicago
Chicago, Illinois 60637
jcao@cs.uchicago.edu

David M. Beazley

Department of Computer Science
University of Chicago
Chicago, Illinois 60637
beazley@cs.uchicago.edu

Abstract

Today, C and C++ are frequently used to write programs that operate as dynamic link libraries, plugins, and extensions of large application frameworks and high level programming languages. An unfortunate side-effect of this programming style is that it makes debugging considerably more difficult. Traditional debuggers don't work as well with extension modules because of the complexity of execution environment. Moreover, the information they provide may be incomplete. To address this limitation, we have been exploring an alternative approach to debugging plugin modules in which a small C/C++ debugger is embedded into the application framework itself. This embedded debugger requires no modifications to the application or plugin modules. Moreover, it is able to catch many common C/C++ programming errors—turning those errors into useful diagnostic error messages. We originally developed this approach for debugging scripting language extension modules, but the approach has since proven to be more widely applicable. In this paper, we describe the embedded debugging approach, and how it can be applied to complicated applications involving plugins, scripting languages, and multiple programming languages. We provide examples of using an embedded debugger with a variety of applications including Python, PHP, and Java, as well as with the popular Apache websever.

1 Introduction

Over the last decade or so, a shift has occurred in the way that a lot of programmers utilize systems programming languages such as C and C++. Specifically, rather than writing large standalone applications in C or C++, a lot of systems programming work has shifted towards creating dynamic link libraries, extension modules and plugins. For instance, a lot of programmers are now using a mix of C/C++ and high-level scripting languages such as Python, Perl, Ruby, and Tcl [2, 3, 5, 6]. In this case, low-level C code is often packaged into some kind of extension module that is just loaded into the language interpreter. Popular languages such as Java also provide a mechanism for accessing native code written in C/C++ [20]. Similarly, many large applications often include some kind of plug-in capability. For instance, internet browsers provide a plugin interface that is used to support different types of content (e.g., Macromedia Flash, Adobe PDF, etc.). A web server such as Apache also provides a plugin interface to support different content generation and management schemes [21]. In fact, significant applications in nearly any application domain tend to have some kind of extension and plugin mechanism.

In this environment of plugins and modules, the way in which C/C++ is utilized is much different than the way in which these languages are traditionally used (and taught). In particular, there is no longer any concept of writing a standalone application. Moreover, this type of coding really isn't the same as writing a programming library. Instead, you are writing code that is merely an optional extension of a larger application framework. The code is really nothing more than some extra functionality that gets dynamically loaded into the application as needed.

To many, the development of plugins and extensions might seem like a task that would be rarely performed. Yet, in certain application domains, extension modules might be the primary means by which end-users interact with an application framework. For instance, in scientific and engineering computing, this programming style is common-place. In this case, a lot of coding is still done in C, C++, or Fortran for performance even though that code may be driven from a large application framework or high-level scripting language.

An interesting side effect of writing plugins and extensions is the difficulty of debugging the associated C/C++ code. Fatal C/C++ programming errors such as invalid memory access, alignment, failed assertions, and

math exceptions don't disappear when writing plugins—-they just become a lot more difficult to isolate. This is because when these errors occur, it usually causes the entire application framework to crash. Then, to figure out what happened, you have to debug the entire application and replicate the environment at the time of the crash. Depending on the application, this task may be difficult. Moreover, even when a traditional debugger is used, it may reveal incomplete information regarding the sequence of events that led to the failure.

In this paper, we examine the problem of debugging C/C++ plugin modules and describe a different approach for handling fatal programming errors such as segmentation faults. In particular, we describe a special purpose debugger that is itself packaged in the form of a plugin module and which can be used to locate problems in mixed-language software and systems that rely on plugins. Originally we developed this debugger to help debug scripting language extensions, but we think that this approach is more widely applicable. Thus, we describe how we have expanded the original work to support new languages and applications that rely on plugins. We then show some examples that illustrate this debugging approach.

2 The Debugging Problem

To illustrate the debugging problems that can arise with plugin modules, consider the problem of working with scripting language extension modules. If you're using a language like Python and an error occurs in a Python script, you typically get some kind of diagnostic traceback like this:

```
% python foo.py
Traceback (innermost last):
  File ''foo.py'', line 11 in ?
    foo()
  File ''foo.py'', line 8 in foo
    bar()
  File ''foo.py'', line 5 in bar
    spam()
  File ''foo.py'', line 5 in spam
    doh()
NameError: doh
```

Using the above information, a programmer can easily locate the error and fix it. On the other hand, if a Python program is accessing the features of an extension module written in C, an error that occurs in the extension module is handled in a much less graceful manner. Specifically, you might get an error like this:

```
% python foo.py
Segmetation Fault (Core dumped)
```

Needless to say, this is much less informative than before. In fact, it doesn't provide any information that could be used to locate the error. It simply tells the user that something very bad has occurred.

As a more extreme case, consider a situation where an extension module is running inside a Python interpreter which itself is running inside some other application such as the Apache webserver (e.g., using mod_python) [9]. In this case, the only indication of an error might be a message in the Apache error log file such as:

```
[Wed Aug 25 14:02:29 2004] [notice]
child pid 12957 exit signal
Segmentation fault (11)
```

In this case, users don't even get a failure html page on their browser. The only thing they know is there is something wrong with the whole application. Yet, no clues are given regarding the source of the error. It could come from a bug in the Apache webserver, from mod_python, from a Python extension module, or any number of other web-server plugins. Even if you had some idea that an error originated from a Python script, it may difficult to extract information from it. Moreover, if you somehow tried to use a script-level debugging tool, it too would crash when the error occurred. Thus, the only way to really get some idea of what has happened might be to use a trial-and-error approach involving print statements.

It is sometimes possible to use a traditional debugger such as gdb[14] to gather information about plugins, but the procedure for doing so is not as straightforward as one might imagine. First, you can't run a debugger on a plugin module in isolation. Instead, you have to run the debugger on the whole application. However, by doing that, you mostly obtain a lot of extraneous information about the internals of the application itself. For instance, the above example, you would obtain a lot of information regarding the internals of Apache and mod_python which can overwhelm the developer. Moreover, if a mix of different programming languages are being used, the information obtained is often incomplete. For instance, if a scripting language is being used, you won't be able to obtain any script-level information about where an error has occurred. Another problem with debugging is that replicating the conditions that led to a crash might be difficult. This is especially true if an application is complex (threads, multiple languages, multiple processes, interprocess communication, etc.). Finally, other peculiarities can make debugging problematic. For instance, to make gdb work nicely with Apache, you have to force Apache to run in a single process mode. However, doing this changes the execution environment in which the crash occurred.

3 Embedded Error Handling

To simplify the debugging of extension modules and plugins, we feel that two facets of debugging can be combined. First, by far the most common use of a traditional C debugger is simply to find out where an error has occured. Specifically, if a program crashes, you almost always type a debugger command like 'where' to get a stack trace. In a lot of cases, this information by itself is enough to isolate an error. Second, most applications that utilize plugins already have a mechanism for dealing with errors and reporting them to users. For example, a scripting language may provide an exception handling facility whereas a web server might print errors to a log file. Therefore, a different approach to plugin debugging might be to embed some kind of C/C++ debugging facility directly into the application itself and allow it to catch fatal errors, gather information normally obtained with a standalone debugger (such as a stack trace) and propagate that information back to the application as an ordinary error condition. The application, in turn, would report the error to the user in the same way that all other errors are reported.

There are many benefits to handling errors in this manner. First, by handling fatal plugin errors as normal errors, you might be able to obtain more detailed information from the application about the conditions that led to a crash. For instance, in Python, you would get information from the Python call stack and the C call stack like this:

```
% python foo.py
Traceback (most recent call last):
  File "", line 1, in ?
  File "foo.py", line 16, in ?
    foo()
  File "foo.py", line 13, in foo
    bar()
  File "foo.py", line 10, in bar
    spam()
  File "foo.py", line 7, in spam
    doh.doh(a,b,c)
SegFault: [ C stack trace ]

#2 0x00027774 in call_builtin(func
   =0x1c74f0, arg=0x1a1ccc,kw=0x0)
#1 0xff022f7c in _wrap_doh(0x0,
   0x1a1ccc, 0x160ef4,0x9c,0x56b44
   ,0x1aa3d8)
#0 0xfe7e0568 in doh(a=0x3,b=0x4
   ,c=0x0)
   in 'foo.c', line 28

/home/jcao/WAD/Python/foo.c, line 28
   int doh(int a, int b, int *c) {
=>   *c = a + b;
      return *c;
   }
```

Clearly this is much more informative than a general "Segmentation Fault" message. Another benefit of this approach is that it simplifies the task of obtaining debug information. Specifically, a developer would not need to run a separate debugging tool to obtain useful information from the application. In fact, the error message alone might be enough to give a developer a very clue about what is broken. Similarly, by reporting errors in this way, end-users can provide developers with useful information (e.g., simply paste the error message into an email message).

4 Challenges

There are many technical challenges to overcome in making an embedded debugging system work. First, it is not practical (or desirable) to modify or recompile the original application or all of the plugin modules. In other words, we don't want to have to modify an application to support debugging. Moreover, it should not be necessary to recompile the whole application in a special "debugging mode." It's important to be able to debug plugin modules within the normal application environment. Second, the way in which fatal errors are handled are handled. By default, most applications will just abort and dump core if a fatal C error occurs. This forces the whole application to quit. To change this, fatal errors have to be intercepted and the control flow of the program altered in order to generate an error. Making this work is especially tricky because the only way to accomplish this is to abort the normal control-flow of the offending plugin code and return control back to the enclosing application framework. Finally, very complicated environments may involve a mix of applications and programming languages—each of which may include it's own error handling facility. Thus, if an error occurs, you need to determine the most appropriate way to report the error.

5 Embedded Debugging with Scripting Languages

To explore the idea of embedded debugging, we first developed a system known as WAD (Wrapped Application Debugger). WAD was initially created for the purpose of debugging scripting language extension modules

and was described in detail in a paper that appeared at the 2001 USENIX Technical Conference [10]. It is not possible to include all of those details here, but we now provide a brief overview of how this system worked. Readers should consult the earlier paper for more detailed implementation information.

The primary goal of WAD is to turn fatal C/C++ errors such as segmentation faults and failed assertions into scripting language exceptions. To do this, WAD is packaged as a standalone shared library that can either be "imported" into a scripting language like any other extension module or linked to another extension module as a library. In either case, when WAD is loaded, it installs a special Unix signal handler to catch a variety of fatal errors such as SIGSEGV, SIGBUS, SIGABRT, SIGILL, and SIGFPE.

Upon reception of a fatal signal, the WAD signal handler takes control and attempts to collect a large amount of information about the execution environment. This information includes a process memory map, information about all loaded libraries, a stack trace, symbol tables, and debugging information—the same kind of information you would normally obtain with debugger. Collecting this information involves interacting with the UNIX /proc filesystem, decoding the contents of ELF object files, and collecting debug information stored in a common format known as "stabs" [16, 17, 18].

Once information has been collected, WAD attempts to report it back to the scripting language interpreter. The procedure for doing this is probably the most tricky and interesting part of the system. WAD starts by walking up the entire call-stack of the application and examining the symbol-table names of each function call. At each level, the name is compared to a list of "known" symbol names that have already been preprogrammed into WAD. The purpose of this scan is to find the boundary between the scripting language interpreter and user-defined extension code. Specifically, WAD is searching for the well-known procedures within the scripting language that are responsible for executing foreign functions. If a match is found during this search, WAD arranges to raise an exception in the scripting language. Previously collected debugging information is then packaged into a form that can be passed back to the interpreter. WAD then generates an error using the same mechanism that extension code would normally use to indicate an error. Finally, WAD aborts the execution of the extension code and forces a return back to the interpreter.

To make the last step work, WAD utilizes a little-known feature of UNIX signal handling that allows a signal handler to modify the process context in which the signal occurred. Specifically, it is possible to rewrite the CPU registers, program counter, and stack pointer within a signal handler and have those changes take effect once the signal handler has returned. Using this, WAD rewrites the process context in a way that forces a return back to the scripting language interpreter. The scripting interpreter itself has no idea that a fatal C/C++ error has occurred—-it merely thinks that extension code has raised an exception and returned. Because of this, when the interpreter regains control, it handles the raised exception in the normal way. This includes unwinding the scripting call stack, generating a traceback, and alerting the user.

Figure 1 illustrates the outcome of using WAD for a fatal error in a C extension to Java. The key feature of this example is that not only do you get a C stack track with debugging information, you *also* get a stack trace generated by the Java virtual machine (which is shown at the bottom of the figure). The reason this appears is that WAD has aborted the execution of the native method and raised an exception with the JVM rather than aborting the whole application and dumping core.

6 Embedded Debugging with Multiple Languages and Applications

The first version of WAD supported three different scripting languages, Python, Tcl, and Perl. For the most part, the system can be easily expanded to support other languages. For instance, we recently expanded WAD to support Java and PHP. This is because most of the implementation is independent of the scripting language itself—only a small amount of code is used to raise an error in the target scripting language. However, one limitation of the implementation was that only one language could be used at once. In other words, a separate debugger was provided for each language. Unfortunately, if multiple languages were used in the same application, things started to break since multiple debugging modules would interfere with each other. The other limitation is that we really only focused on compiled extensions to high-level programming languages. However, there are a variety of other applications where a facility for debugging plugins might be useful. Therefore, we were interested in seeing how this debugging technique might be generalized and used in other settings.

To illustrate the use of WAD in a slightly different setting, consider an application that relies on plugin modules such as the Apache webserver [8]. Apache allows modules written in C/C++ to be dynamically loaded into the webserver. These modules are typically used to implement different types of dynamic content such as server side includes and server scripting. They can also be used to implement security mechanisms, user tracking with cookies, and more. The only time that a module

```
Exception in thread "main" . Uncaught exception of type java.lang.Class.
Segmentation fault.

[ C stack trace ]

#9   0x4030791c in jni_invoke_static__FP7JNIEnv_P9JavaValueP8_jobject11JNICallTy
peP10_jmethodIDP18JNI_ArgumentPusherP6Thread()
#8   0x402ff6a6 in call__9JavaCallsP9JavaValueG12methodHandleP17JavaCallArgument
sP6Thread()
#7   0x403b33ad in os_exception_wrapper__2osPFP9JavaValueP12methodHandleP17JavaC
allArgumentsP6Thread_vP9JavaValueP12methodHandleP17JavaCallArgumentsP6Thread()
#6   0x402ff454 in call_helper__9JavaCallsP9JavaValueP12methodHandleP17JavaCallA
rgumentsP6Thread()
#5   0x4267a1c4 in ?()
#4   0x4267cdef in ?()
#3   0x4267cdef in ?()
#2   0x42682cb2 in ?()
#1   0x4c8a165f in Java_exampleJNI_java_1seg_1crash(jenv=0x805b878,jcls=0xbfffd6
e0) in 'example_wrap.c', line 86
#0   0x4c8a13e4 in java_seg_crash() in 'example.c', line 27

/home/jcao/classes/research/WAD/java_try/example.c, line 27

    int java_seg_crash() {
      int *a = 0;
 =>   *a = 3;
      return 1;
    }

FATAL ERROR in native method: runtime error
        at exampleJNI.java_seg_crash(Native Method)
        at example.java_seg_crash(example.java:12)
        at try1.main(try1.java:9)
```

Figure 1: Traceback information produced by WAD for a segmentation fault in a Java extension

would execute in Apache is when a remote client connected to an appropriate page or document type. If a fatal error occurs in one of these modules, Apache normally prints a small message to its error log, and terminates the child process. When this happens, the remote client will merely get an empty web-page or some kind of connection error indicating the server closed down the connection. However, with WAD we can do something a little different. Instead of crashing, WAD can obtain information about the client connection. Then, on behalf of the failed plugin module, diagnostic debugging information can be routed both to the Apache error log and to the outgoing network connection. Thus, a failure in server plugin code will produce a web page such as shown in Figure 2.

The fact that the error is produced on the client makes it extremely easy to debug the module—a developer can just look at the displayed page to get a stack trace.

The Apache example also serves as an introduction to a more complicated scenario involving multiple languages. In Apache, a variety of different extension mechanisms may be in use. For instance, the Apache server may have plugin modules for a variety of scripting languages including Python, PHP, and Perl. The purpose of these modules would allow for various kinds of server-side scripting and dynamic content generation. However, as all of these scripting languages also allow plugins of their own, you now have a situation where you might have Apache plugins, Python plugins, Perl plugins, and PHP plugins all coexisting in the same process. Needless to say, debugging C/C++ code in this setting becomes much more complex.

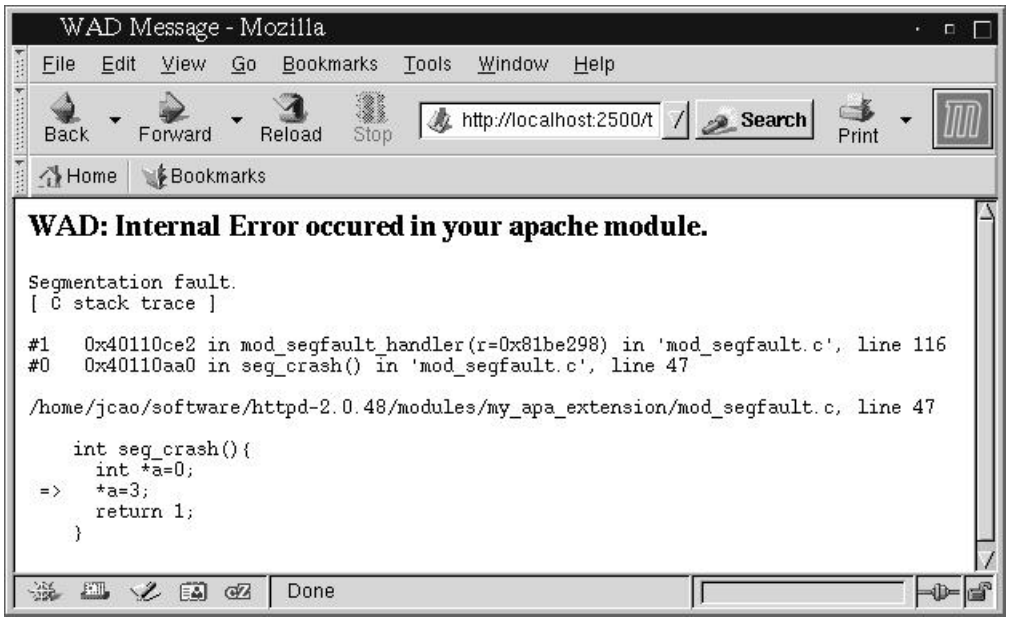

Figure 2: Displaying debugging information on a user's browser after an error in an Apache plugin module

To handle multiple languages and applications, WAD maintains a master symbol table of functions that are known to call out to extension code. This table contains a variety of entries like this:

Symbol name	Handler function
`PyCFunction_Call`	`python_handler`
`TclExecuteByteCode`	`tcl_handler`
`call_user_function_ex`	`php_handler`
`ap_run_handler`	`apache_handler`
...	

The contents of this table are application specific and require some detailed knowledge of how a particular application accesses extension code. However, it's not something that a user of WAD normally needs to worry about.

When a fatal error occurs, WAD walks up the call stack and compares each stack frame to see if its symbolic name matches an entry in this list. If a match is found, the fatal error is handled by a language-specific handler function which is responsible for interacting with the enclosing application and signalling an error. This scheme allows an error to be handled in the most appropriate manner.

Figure 3 shows sample output of a fatal error occurring within a PHP extension module running inside Apache. In this case, the error is handled by PHP itself and you get information from the C call stack *in addition to* some information about the PHP script that was running at the time. Again, the key point is that you're getting much more information about what has caused a failure. You obtain a C stack track, you obtain PHP script information, and you get output on the web-page.

A peculiar problem that occurs with supporting multiple target languages is discovering which languages are in use and available. Normally, WAD might need to interact with all of the available languages in order to raise errors and abort execution of offending code. This implies that WAD needs to link to API functions in each language. Yet, at the same time, there is no guarantee that all of the languages are going to be installed or available in a given application. Nor is the prospect of relying upon many different debugging modules particularly appealing. To deal with this, we currently rely upon lazy-binding of functions. When WAD is loaded into an environment and an error occurs, WAD uses features of the dynamic loader to discover information about what applications and languages are available. In addition, WAD can defer the linking of special API functions until they are actually needed. With this approach, a single debugging module can be used to debug plugin modules in a variety of difficult programming languages and applications–even when mixed together.

7 Discussion

WAD provides an alternative approach to problem of debugging plugins and extensions. By embedding a debugger into the application, there is no need to modify

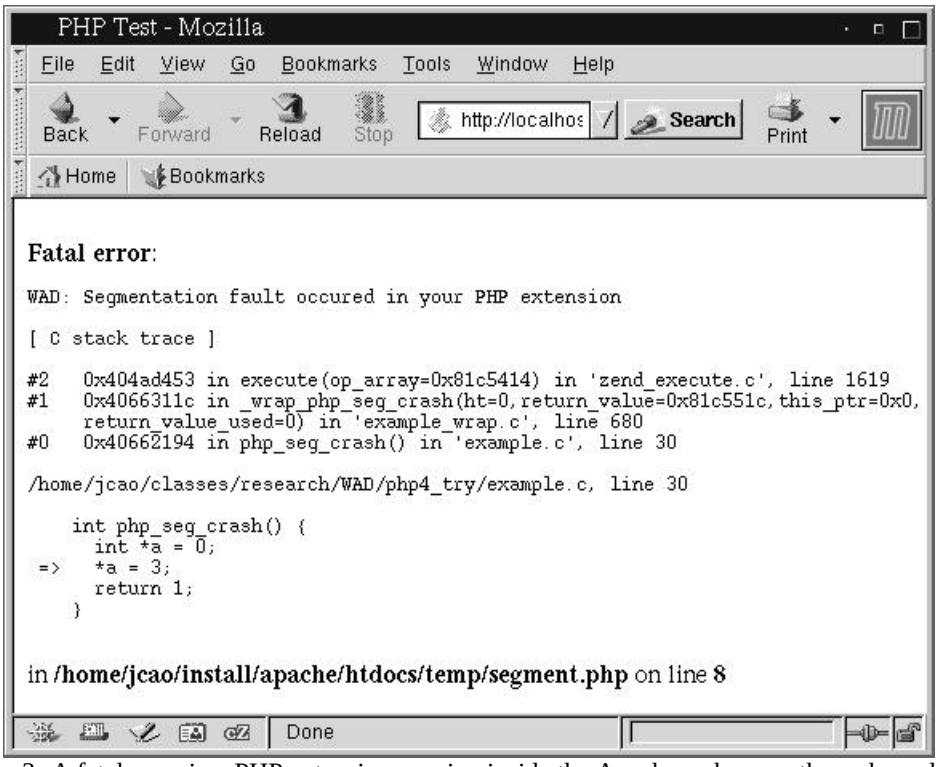

Figure 3: A fatal error in a PHP extension running inside the Apache webserver through mod_php

existing code or reconfigure the application. Moreover, when errors occur, users get immediate feedback about what has happened. In addition to getting a C stack, WAD is often able to produce additional information such as where in a script an error occurred. WAD also makes it easier for developers to get information from end users. For instance, if failure occurs, a user can simply paste the stack trace into an email and send it to a developer. This works even if a user doesn't have a C development environment installed on their machine.

On the other hand, this approach also has many limitations. The programming techniques used to implement WAD are complex and platform-dependent–involving highly specialized operating system features and assembly language. As a consequence, the system currently only works on Sparc Solaris and Intel Linux systems. Additional information about this can be found in our USENIX paper [10]. However, at best, the system can only be considered as a "proof of concept."

In addition, there are a variety of C programming errors that can't be handled in this manner. For instance, if a program overflows the stack, the call-stack becomes corrupted and it becomes impossible for WAD to properly recover. Similarly, certain kinds of memory errors can corrupt the application environment in a way that makes it impossible for WAD or the enclosing application framework to operate correctly—in which case the whole application just will crash as before. In these cases, WAD is of little use. However, it's worth nothing that a traditional debugger could still be used in these cases if desired.

Finally, a side effect of using a system like WAD is that an application continues to run even after a fatal error has occurred. For instance, if you get a segmentation fault in a Python extension, that generates an exception and the Python interpreter continues to operate. In principle, one could continue to use the interpreter for other tasks. However, WAD is not intended to serve as a generate purpose error-recovery mechanism. When execution is aborted, the C call stack is truncated and no attempt is made to cleanly reclaim resources. For languages like C++, this can leave all sorts of dangling pointers and memory leaks. In a multithreaded system, this might result in stale locks. Various system resources such as open files and network connections might not be closed properly. It's important to keep in mind that the primary goal of WAD is to merely extract useful information from the environment and present it to the user.

8 Related Work

Different debugging methods and debuggers have been developed on both C/C++ level and scripting level.

Much effort has been put on C/C++ level debugging problem. For instance new processors support JTAG pins by which debugger can get informations of internal processers. Other techniques like debugging moniter is also used by certain type of debuggers. RTOS uses a different approach known as dynamic loadable and linkable to work with OS level and application level bugs.

There are a large number of scripting debuggers exist. Most of them provides debugging facilities for one special scripting language. However, little information could be retrieved from C stack in case of extension error.

Unfortunately, little work has been done on the area of debugging in mixed-language environment. Modern debuggers either ingore the scripting interpreters by throwing the whole stack information to user or just provide scripting level stack information. Although some debuggers like gdb supports big application like apache webserver, it requires to modify the orniginal environment and no scripting level traceback information is available.

9 Future Directions

WAD currently supports a limited number of languages, applications and platforms. Thus, this is an area of obvious expansion. There are also a wide variety of subtle issues that could be explored. For instance, providing better integration with C++ (e.g., connecting C++ exceptions to application error handling mechanisms), working with threads, and working with software component frameworks.

It may also be possible to integrate WAD with some existing debugging tools in some way. For instance, in the case of scripting languages, it might be possible to connect WAD to existing script-level debuggers—allowing those debuggers to examine information from the C call stack in addition to providing script-level diagnostics.

Porting WAD to non-Unix systems could also be an interesting challenge. For instance, it might be interesting to see whether a similar system could be built using Windows structured exception handling [31].

10 Conclusions and Availability

In this paper we have presented an alternative approach to debugging plugin modules in which a debugger has been embedded directly into the application environment. This debugger is then able to catch fatal C/C++ errors, extract debugging information, and present it back to the application. This approach makes it easier for a developer to obtain more detailed information about how a complex application has failed. Moreover, it has the unique feature of being able to unravel programming problems that span multiple programming languages. Although our solution is merely a proof-of-concept, we feel that this approach is interesting and might have utility in a variety of applications domains—especially where developers spend a lot of time writing plugin modules. An embedded error handling mechanism such as this could also be of interest to language and application designers who want to enhance the interaction between the application and extension code.

Source code and documentation for WAD can be obtained at:
`http://systems.cs.uchicago.edu/wad.`

11 Acknowledgments

This work has been performed in part under the generous support of the National Science Foundation (NSF Grant CCR-0237835).

References

[1] J. K. Ousterhout, *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer, Vol 31, No. 3, p. 23-30, 1998.

[2] M. Lutz, *Programming Python*, O'Reilly & Associates, 1996.

[3] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, 2nd. Ed. O'Reilly & Associates, 1996.

[4] T. Ratschiller and T. Gerken, *Web Application Development with PHP 4.0*, New Riders, 2000.

[5] D. Thomas, A. Hunt, *Programming Ruby*, Addison-Wesley, 2001.

[6] J. K. Ousterhout, *Tcl: An Embedable Command Language*, Proceedings of the USENIX Association Winter Conference, 1990.

[7] K. Arnold, J. Goslingm, *The Java Programming Language.*, Sun Microsystems.

[8] *Apache HTTP Server Project*, `http://httpd.apache.org/`

[9] *Mod_python project*, `http://www.modpython.org/`

[10] D.M. Beazley, *An Embedded Error Recovery and Debugging Mechanism for Scripting Language Extensions.*, Proceedings of the USENIX Technical Conference, June, 2001.

[11] John E. Grayson. *Python and Tkinter Programming.* Manning, 2000.

[12] D.M. Beazley, *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*, Proceedings of the 4th USENIX Tcl/Tk Workshop, p. 129-139, July 1996.

[13] S. Srinivasan, *Advanced Perl Programming*, O'Reilly & Associates, 1997.

[14] R. Stallman and R. Pesch, *Using GDB: A Guide to the GNU Source-Level Debugger*. Free Software Foundation and Cygnus Support, Cambridge, MA, 1991.

[15] W. Richard Stevens, *UNIX Network Programming: Interprocess Communication, Volume 2*. PTR Prentice-Hall, 1998.

[16] R. Faulkner and R. Gomes, *The Process File System and Process Model in UNIX System V*, USENIX Conference Proceedings, January 1991.

[17] J. R. Levine, *Linkers & Loaders.* Morgan Kaufmann Publishers, 2000.

[18] Free Software Foundation, *The "stabs" debugging format*. GNU info document.

[19] M.L. Scott. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

[20] S. Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison Wesley, 1999.

[21] Apache API. `http://httpd.apache.org/docs/misc/API.html`

[22] J. B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley & Sons, 1996.

[23] D.E. Perry, A. Romanovsky, and A. Tripathi, *Current Trends in Exception Handling-Part I*, IEEE Transactions on Software Engineering, Vol 26, No. 9, p. 817-819, 2000.

[24] D.E. Perry, A. Romanovsky, and A. Tripathi, *Current Trends in Exception Handling-Part II*, IEEE Transactions on Software Engineering, Vol 26, No. 10, p. 921-922, 2000.

[25] G.L. Steele Jr., *Common Lisp: The Language, Second Edition*, Digital Press, Bedford, MA. 1990.

[26] H. Sexton, *Foreign Functions and Common Lisp*, in Lisp Pointers, Vol 1, No. 5, 1988.

[27] W. Henessey, *WCL: Delivering Efficient Common Lisp Applications Under Unix*, ACM Conference on Lisp and Functional Languages, p. 260-269, 1992.

[28] P.A. Buhr and W.Y.R. Mok, *Advanced Exception Handling Mechanisms*, IEEE Transactions on Software Engineering, Vol. 26, No. 9, p. 820-836, 2000.

[29] S. Marlow, S. P. Jones, and A. Moran. *Asynchronous Exceptions in Haskell.* In 4th International Workshop on High-Level Concurrent Languages, September 2000.

[30] J. H. Reppy, *Asynchronous Signals in Standard ML*. Technical Report TR90-1144, Cornell University, Computer Science Department, 1990.

[31] M. Pietrak, *A Crash Course on the Depths of Win32 Structured Exception Handling*, Microsoft Systems Journal, January 1997.