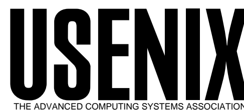
USENIX Association

Proceedings of the 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001



© 2001 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

An Embedded Error Recovery and Debugging Mechanism for Scripting Language Extensions

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

Abstract

In recent years, scripting languages such as Perl, Python, and Tcl have become popular development tools for the creation of sophisticated application software. One of the most useful features of these languages is their ability to easily interact with compiled languages such as C and C++. Although this mixed language approach has many benefits, one of the greatest drawbacks is the complexity of debugging that results from using interpreted and compiled code in the same application. In part, this is due to the fact that scripting language interpreters are unable to recover from catastrophic errors in compiled extension code. Moreover, traditional C/C++ debuggers do not provide a satisfactory degree of integration with interpreted languages. This paper describes an experimental system in which fatal extension errors such as segmentation faults, bus errors, and failed assertions are handled as scripting language exceptions. This system, which has been implemented as a general purpose shared library, requires no modifications to the target scripting language, introduces no performance penalty, and simplifies the debugging of mixed interpreted-compiled application software.

1 Introduction

Slightly more than ten years have passed since John Ousterhout introduced the Tcl scripting language at the 1990 USENIX technical conference [1]. Since then, scripting languages have been gaining in popularity as evidenced by the wide-spread use of systems such as Tcl, Perl, Python, Guile, PHP, and Ruby [1, 2, 3, 4, 5, 6].

In part, the success of modern scripting languages is due to their ability to be easily integrated with software written in compiled languages such as C, C++, and Fortran. In addition, a wide variety of wrapper generation tools can be used to automatically produce bindings between existing code and a variety of scripting language environments [7, 8, 9, 10, 11, 12, 13, 14, 15]. As a result, a large number of programmers are now using scripting languages to control complex C/C++ programs or as a tool for re-engineering legacy software. This approach is attractive because it allows programmers to benefit from the flexibility and rapid development of scripting while retaining the best features of compiled code such as high performance [16].

A critical aspect of scripting-compiled code integration is the way in which it departs from traditional C/C++ development and shell scripting. Rather than building stand-alone applications that run as separate processes, extension programming encourages a style of programming in which components are tightly integrated within an interpreter that is responsible for high-level control. Because of this, scripted software tends to rely heavily upon shared libraries, dynamic loading, scripts, and third-party extensions. In this sense, one might argue that the benefits of scripting are achieved at the expense of creating a more complicated development environment.

A consequence of this complexity is an increased degree of difficulty associated with debugging programs that utilize multiple languages, dynamically loadable modules, and a sophisticated runtime environment. To address this problem, this paper describes an experimental system known as WAD (Wrapped Application Debugger) in which an embedded error reporting and debugging mechanism is added to common scripting languages. This system converts catastrophic signals such as segmentation faults and failed assertions to exceptions that can be handled by the scripting language interpreter. In doing so, it provides more seamless integration between error handling in scripting language interpreters and compiled extensions.

2 The Debugging Problem

Normally, a programming error in a scripted application results in an exception that describes the problem and the context in which it occurred. For example, an error in a Python script might produce a traceback similar to the following:

```
% 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 2, in spam
    doh()
NameError: doh
```

In this case, a programmer might be able to apply a fix simply based on information in the traceback. Alternatively, if the problem is more complicated, a script-level debugger can be used to provide more information. In contrast, a failure in compiled extension code might produce the following result:

```
% python foo.py
Segmentation Fault (core dumped)
```

In this case, the user has no idea of what has happened other than it appears to be "very bad." Furthermore, script-level debuggers are unable to identify the problem since they also crash when the error occurs (they run in the same process as the interpreter). This means that the only way for a user to narrow the source of the problem within a script is through trial-and-error techniques such as inserting print statements, commenting out sections of scripts, or having a deep intuition of the underlying implementation. Obviously, none of these techniques are particularly elegant.

An alternative approach is to run the application under the control of a traditional debugger such as gdb [17]. Although this provides some information about the error, the debugger mostly provides detailed information about the internal implementation of the scripting language interpreter instead of the script-level code that was running at the time of the error. Needless to say, this information isn't very useful to most programmers. A related problem is that the structure of a scripted application tends to be much more complex than a traditional stand-alone program. As a result, a user may not have a good sense of how to actually attach an external debugger to their script. In addition, execution may occur within a complex run-time environment involving events, threads, and network connections. Because of this, it can be difficult for the user to reproduce and identify certain types of catastrophic errors if they depend on timing or unusual event sequences. Finally, this approach requires a programmer to have a C development environment installed on their machine. Unfortunately, this may not hold in practice. This is because scripting languages are often used to provide programmability to applications where end-users write scripts, but do not write low-level C code.

Even if a traditional debugger such as gdb were modified to provide better integration with scripting languages, it is not clear that this would be the most natural solution to the problem. For one, having to run a separate debugging process to debug extension code is unnatural when no such requirement exists for scripts. Moreover, even if such a debugger existed, an inexperienced user may not have the expertise or inclination to use it. Finally, obscure fatal errors may occur long after an application has been deployed. Unless the debugger is distributed along with the application in some manner, it will be extraordinary difficult to obtain useful diagnostics when such errors occur.

The current state of the art in extension debugging is to simply add as much error checking as possible to extension modules. This is never a bad thing to do, but in practice it's usually not enough to eliminate every possible problem. For one, scripting languages are sometimes used to control hundreds of thousands to millions of lines of compiled code. In this case, it is improbable that a programmer will foresee every conceivable error. In addition, scripting languages are often used to put new user interfaces on legacy software. In this case, scripting may introduce new modes of execution that cause a formerly "bug-free" application to fail in an unexpected manner. Finally, certain types of errors such as floating-point exceptions can be particularly difficult to eliminate because they might be generated algorithmically (e.g., as the result of instability in a numerical method). Therefore, even if a programmer has worked hard to eliminate crashes, there is usually a small probability that an application may fail under unusual circumstances.

3 Embedded Error Reporting

Rather than modifying an existing debugger to support scripting languages, an alternative approach is to add a more powerful error handling and reporting mechanism to the scripting language interpreter. We have implemented this approach in the form of an experimental system known as WAD. WAD is packaged as dynamically loadable shared library that can either be loaded as a scripting language extension module or linked to ex-

```
% python foo.py
Traceback (most recent call last):
  File "<stdin>", 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) in 'ceval.c',line 2650
#1 0xff083544 in _wrap_doh(self=0x0,args=0x1a1ccc) in 'foo_wrap.c',line 745
#0 0xfe7e0568 in doh(a=3,b=4,c=0x0) in 'foo.c',line 28

/u0/beazley/Projects/WAD/Python/foo.c, line 28

    int doh(int a, int b, int *c) {
 =>   *c = a + b;
      return *c;
    }
```

Figure 1: Cross language traceback generated by WAD for a segmentation fault in a Python extension

isting extension modules as a library. The core of the system is generic and requires no modifications to the scripting interpreter or existing extension modules. Furthermore, the system does not introduce a performance penalty as it does not rely upon program instrumentation or tracing.

WAD works by converting fatal signals such as SIGSEGV, SIGBUS, SIGFPE, and SIGABRT into scripting language exceptions that contain debugging information collected from the call-stack of compiled extension code. By handling errors in this manner, the scripting language interpreter is able to produce a cross-language stack trace that contains information from both the script code and extension code as shown for Python and Tcl/Tk in Figures 1 and 2. In this case, the user is given a very clear idea of what has happened without having to launch a separate debugger.

The advantage to this approach is that it provides more seamless integration between error handling in scripts and error handling in extensions. In addition, it eliminates the most common debugging step that a developer is likely to perform in the event of a fatal error–running a separate debugger on a core file and typing 'where' to get a stack trace. Finally, this allows end-users to provide extension writers with useful debugging information since they can supply a stack trace as opposed to a vague complaint that the program "crashed."

4 Scripting Language Internals

In order to provide embedded error recovery, it is critical to understand how scripting language interpreters interface with extension code. Despite the wide variety of scripting languages, essentially every implementation uses a similar technique for accessing foreign code.

Virtually all scripting languages provide an extension mechanism in the form of a foreign function interface in which compiled procedures can be called from the scripting language interpreter. This is accomplished by writing a collection of wrapper functions that conform to a specified calling convention. The primary purpose of the wrappers are to marshal arguments and return values between the two languages and to handle errors. For example, in Tcl, every wrapper function must conform to the following prototype:

```
int
wrap_foo(ClientData clientData,
         Tcl_Interp *interp,
         int objc,
         Tcl_Obj *CONST objv[])
{
    /* Convert arguments */
    ...
    /* Call a function */
```

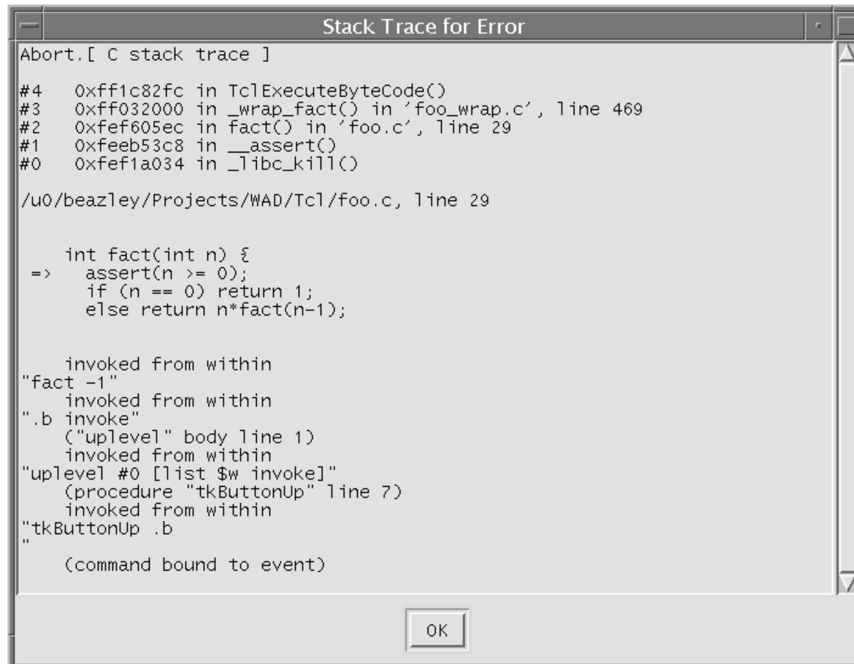


Figure 2: Dialog box with WAD generated traceback information for a failed assertion in a Tcl/Tk extension

```
    result = foo(args);
    /* Set result */
    ...
    if (success) {
        return TCL_OK;
    } else {
        return TCL_ERROR;
    }
}
```

Another common extension mechanism is an object/type interface that allows programmers to create new kinds of fundamental types or attach special properties to objects in the interpreter. For example, both Tcl and Python provide an API for creating new "built-in" objects that behave like numbers, strings, lists, etc. In most cases, this involves setting up tables of function pointers that define various properties of an object. For example, if you wanted to add complex numbers to an interpreter, you might fill in a special data structure with pointers to methods that implement various numerical operations like this:

```
NumberMethods ComplexMethods {
    complex_add,
    complex_sub,
    complex_mul,
    complex_div,
    ...
};
```

Once registered with the interpreter, the methods in this structure would be invoked by various interpreter operators such as $+$, $-$, $*$, and $/$.

Most interpreters handle errors as a two-step process in which detailed error information is first registered with the interpreter and then a special error code is returned. For example, in Tcl, errors are handled by setting error information in the interpreter and returning a value of TCL_ERROR. Similarly in Python, errors are handled by calling a special function to raise an exception and returning NULL. In both cases, this triggers the interpreter's error handler—possibly resulting in a stack trace of the running script. In some cases, an interpreter might handle errors using a form of the C `longjmp` function. For example, Perl provides a special function `die` that jumps back to the interpreter with a fatal error [11].

The precise implementation details of these mechanisms aren't so important for our discussion. The critical point is that scripting languages always access extension code though a well-defined interface that precisely defines how arguments are to be passed, values are to be returned, and errors are to be handled.

5 Scripting Languages and Signals

Under normal circumstances, errors in extension code are handled through the error-handling API provided by the scripting language interpreter. For example, if an

invalid function parameter is passed, a program can simply set an error message and return to the interpreter. Similarly, automatic wrapper generators such as SWIG can produce code to convert C++ exceptions and other C-related error handling schemes to scripting language errors [18]. On the other hand, segmentation faults, failed assertions, and similar problems produce signals that cause the interpreter to abort execution.

Most scripting languages provide limited support for Unix signal handling [19]. However, this support is not sufficiently advanced to recover from fatal signals produced by extension code. Unlike signals generated for asynchronous events such as I/O, execution can *not* be resumed at the point of a fatal signal. Therefore, even if such a signal could be caught and handled by a script, there isn't much that it can do except to print a diagnostic message and abort before the signal handler returns. In addition, some interpreters block signal delivery while executing extension code–opting to handle signals at a time when it is more convenient. In this case, a signal such as SIGSEGV would simply cause the whole application to freeze since there is no way for execution to continue to a point where the signal could be delivered. Thus, scripting languages tend to either ignore the problem or label it as a "limitation."

6 Overview of WAD

WAD installs a signal handler for SIGSEGV, SIGBUS, SIGABRT, SIGILL, and SIGFPE using the `sigaction` function [19]. Furthermore, it uses a special option (SA_SIGINFO) of signal handling that passes process context information to the signal handler when a signal occurs. Since none of these signals are normally used in the implementation of the scripting interpreter or by user scripts, this does not usually override any previous signal handling. Afterwards, when one of these signals occurs, a two-phase recovery process executes. First, information is collected about the execution context including a full stack-trace, symbol table entries, and debugging information. Then, the current stream of execution is aborted and an error is returned to the interpreter. This process is illustrated in Figure 3.

The collection of context and debugging information involves the following steps:

- The program counter and stack pointer are obtained from context information passed to the signal handler.
- The virtual memory map of the process is obtained from /proc and used to associate virtual memory addresses with executable files, shared libraries, and dynamically loaded extension modules [20].
- The call stack is unwound to collect traceback information. At each step of the stack traceback, symbol table and debugging information is gathered and stored in a generic data structure for later use in the recovery process. This data is obtained by memory-mapping the object files associated with the process and extracting symbol table and debugging information.

Once debugging information has been collected, the signal handler enters an error-recovery phase that attempts to raise a scripting exception and return to a suitable location in the interpreter. To do this, the following steps are performed:

- The stack trace is examined to see if there are any locations in the interpreter to which control can be returned.
- If a suitable return location is found, the CPU context is modified in a manner that makes the signal handler return to the interpreter with an error. This return process is assisted by a small trampoline function (partially written in assembly language) that arranges a proper return to the interpreter after the signal handler returns.

Of the two phases, the first is the most straightforward to implement because it involves standard Unix API functions and common file formats such as ELF and stabs [21, 22]. On the other hand, the recovery phase in which control is returned to the interpreter is of greater interest. Therefore, it is now described in greater detail.

7 Returning to the Interpreter

To return to the interpreter, WAD maintains a table of symbolic names that correspond to locations within the interpreter responsible for invoking wrapper functions and object/type methods. For example, Table 1 shows a partial list of return locations used in the Python implementation. When an error occurs, the call stack is scanned for the first occurrence of any symbol in this table. If a match is found, control is returned to that location by emulating the return of a wrapper function with the error code from the table. If no match is found, the error handler simply prints a stack trace to standard output and aborts.

When a symbolic match is found, WAD invokes a special user-defined handler function that is written for a specific scripting language. The primary role of this handler is to take debugging information gathered from the call stack and generate an appropriate scripting language error. One peculiar problem of this step is that the

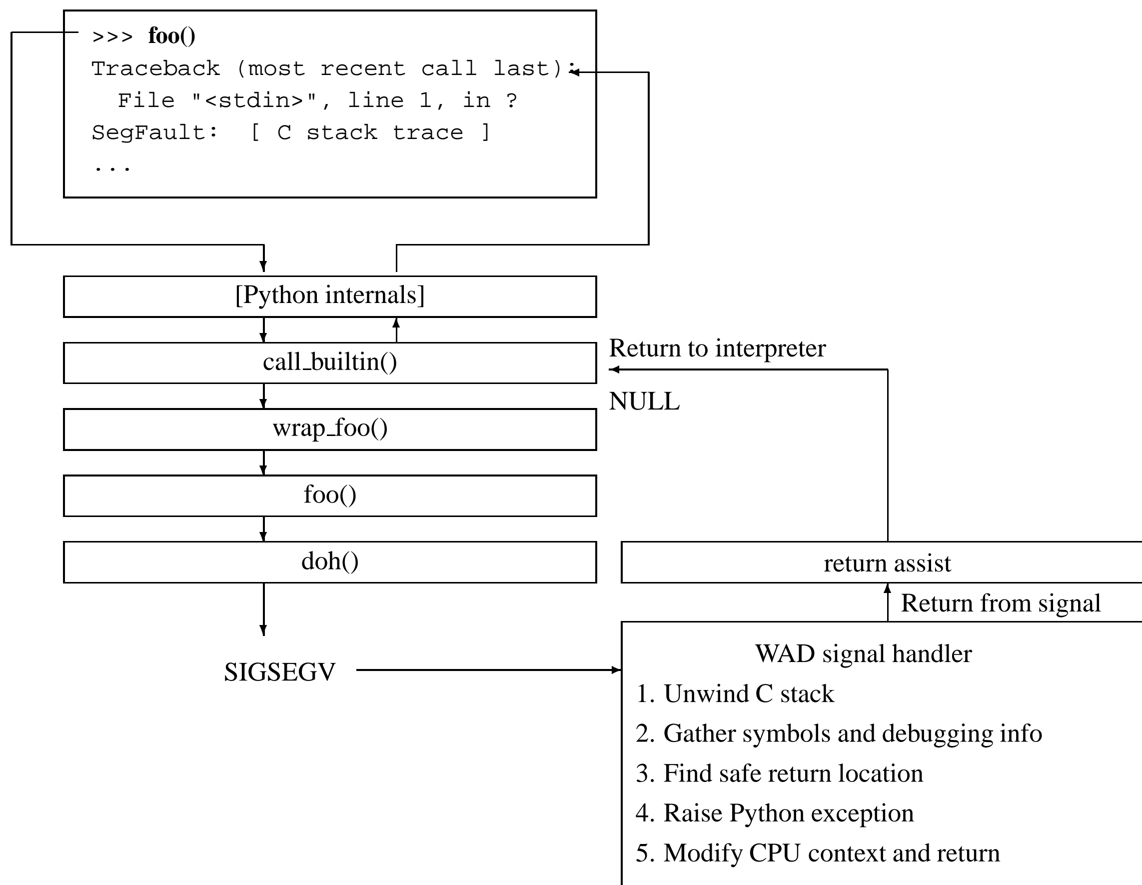


Figure 3: Control Flow of the Error Recovery Mechanism for Python

generation of an error may require the use of parameters passed to a wrapper function. For example, in the Tcl wrapper shown earlier, one of the arguments was an object of type "`Tcl_Interp *`". This object contains information specific to the state of the interpreter (and multiple interpreter objects may exist in a single application). Unfortunately, no reference to the interpreter object is available in the signal handler nor is a reference to interpreter guaranteed to exist in the context of a function that generated the error.

To work around this problem, WAD implements a feature known as argument stealing. When examining the call-stack, the signal handler has full access to all function arguments and local variables of each function on the stack. Therefore, if the handler knows that an error was generated while calling a wrapper function (as determined by looking at the symbol names), it can grab the interpreter object from the stack frame of the wrapper and use it to set an appropriate error code before returning to the interpreter. Currently, this is managed by allowing the signal handler to steal arguments from the caller using positional information. For example, to grab the `Tcl_Interp *` object from a Tcl wrapper function, code similar to the following is written:

```
Tcl_Interp *interp;
int         err;

interp = (Tcl_Interp *)
  wad_steal_outarg(
           stack,
           "TclExecuteByteCode",
           1,
           &err
  );
  ...
if (!err) {
  Tcl_SetResult(interp,errtype,...);
  Tcl_AddErrorInfo(interp,errdetails);
}
```

In this case, the Tcl interpreter argument passed to a wrapper function is stolen and used to generate an error. Also, the name `TclExecuteByteCode` refers to the calling function, not the wrapper function itself. At

Python symbol	Error return value
call_builtin	NULL
PyObject_Print	-1
PyObject_CallFunction	NULL
PyObject_CallMethod	NULL
PyObject_CallObject	NULL
PyObject_Cmp	-1
PyObject_DelAttrString	-1
PyObject_DelItem	-1
PyObject_GetAttrString	NULL

Table 1: A partial list of symbolic return locations in the Python interpreter

this time, argument stealing is only applicable to simple types such as integers and pointers. However, this appears to be adequate for generating scripting language errors.

8 Register Management

A final issue concerning the return mechanism has to do with the behavior of the non-local return to the interpreter. Roughly speaking, this emulates the C `longjmp` library call. However, this is done without the use of a matching `setjmp` in the interpreter.

The primary problem with aborting execution and returning to the interpreter in this manner is that most compilers use a register management technique known as callee-save [23]. In this case, it is the responsibility of the called function to save the state of the registers and to restore them before returning to the caller. By making a non-local jump, registers may be left in an inconsistent state due to the fact that they are not restored to their original values. The `longjmp` function in the C library avoids this problem by relying upon `setjmp` to save the registers. Unfortunately, WAD does not have this luxury. As a result, a return from the signal handler may produce a corrupted set of registers at the point of return in the interpreter.

The severity of this problem depends greatly on the architecture and compiler. For example, on the SPARC, register windows effectively solve the callee-save problem [24]. In this case, each stack frame has its own register window and the windows are flushed to the stack whenever a signal occurs. Therefore, the recovery mechanism can simply examine the stack and arrange to restore the registers to their proper values when control is returned. Furthermore, certain conventions of the SPARC ABI resolve several related issues. For example, floating point registers are caller-saved and the contents of the SPARC global registers are not guaranteed to be preserved across procedure calls (in fact, they are not even saved by `setjmp`).

On other platforms, the problem of register management becomes more interesting. In this case, a heuristic approach that examines the machine code for each function on the call stack can be used to determine where the registers might have been saved. This approach is used by gdb and other debuggers when they allow users to inspect register values within arbitrary stack frames [17]. Even though this sounds complicated to implement, the algorithm is greatly simplified by the fact that compilers typically generate code to store the callee-save registers immediately upon the entry to each function. In addition, this code is highly regular and easy to examine. For instance, on i386-Linux, the callee-save registers can be restored by simply examining the first few bytes of the machine code for each function on the call stack to figure out where values have been saved. The following code shows a typical sequence of machine instructions used to store callee-save registers on i386-Linux:

```
foo:
55          pushl %ebp
89 e5       mov  %esp, %ebp
83 a0       subl  $0xa0,%esp
56          pushl %esi
57          pushl %edi
...
```

As a fall-back, WAD could be configured to return control to a location previously specified with `setjmp`. Unfortunately, this either requires modifications to the interpreter or its extension modules. Although this kind of instrumentation could be facilitated by automatic wrapper code generators, it is not a preferred solution and is not discussed further.

9 Initialization

To simplify the debugging of extension modules, it is desirable to make the use of WAD as transparent as possible. Currently, there are two ways in which the system is used. First, WAD may be explicitly loaded as a scripting language extension module. For instance, in Python, a user can include the statement `import libwadpy` in a script to load the debugger. Alternatively, WAD can be enabled by linking it to an extension module as a shared library. For instance:

```
% ld -shared $(OBJS) -lwadpy
```

In this latter case, WAD initializes itself whenever the extension module is loaded. The same shared library

is used for both situations by making sure two types of initialization techniques are used. First, an empty initialization function is written to make WAD appear like a proper scripting language extension module (although it adds no functions to the interpreter). Second, the real initialization of the system is placed into the initialization section of the WAD shared library object file (the "init" section of ELF files). This code always executes when a library is loaded by the dynamic loader is commonly used to properly initialize C++ objects. Therefore, a fairly portable way to force code into the initialization section is to encapsulate the initialization in a C++ statically constructed object like this:

```
class InitWad {
   public:
      InitWad() { wad_init(); }
};
/* This forces InitWad() to execute
   on loading. */
static InitWad init;
```

The nice part about this technique is that it allows WAD to be enabled simply by linking or loading; no special initialization code needs to be added to an extension module to make it work. In addition, due to the way in which the loader resolves and initializes libraries, the initialization of WAD is guaranteed to execute before any of the code in the extension module to which it has been linked. The primary downside to this approach is that the WAD shared object file can not be linked directly to an interpreter. This is because WAD sometimes needs to call the interpreter to properly initialize its exception handling mechanism (for instance, in Python, four new types of exceptions are added to the interpreter). Clearly this type of initialization is impossible if WAD is linked directly to an interpreter as its initialization process would execute before before the main program of the interpreter started. However, if you wanted to permanently add WAD to an interpreter, the problem is easily corrected by first removing the C++ initializer from WAD and then replacing it with an explicit initialization call someplace within the interpreter's startup function.

10 Exception Objects

Before WAD returns control to the interpreter, it collects all of the stack-trace and debugging information it was able to obtain into a special exception object. This object represents the state of the call stack and includes things like symbolic names for each stack frame, the names, types, and values of function parameters and stack variables, as well as a complete copy of data on the stack. This information is represented in a generic manner that hides platform specific details related to the CPU, object file formats, debugging tables, and so forth.

Minimally, the exception data is used to print a stack trace as shown in Figure 1. However, if the interpreter is successfully able to regain control, the contents of the exception object can be freely examined after an error has occurred. For example, a Python script could catch a segmentation fault and print debugging information like this:

```
try:
   # Some buggy code
   ...
except SegFault,e:
   print 'Whoa!'
   # Get WAD exception object
   t = e.args[0]
   # Print location info
   print t.__FILE__
   print t.__LINE__
   print t.__NAME__
   print t.__SOURCE__
   ...
```

Inspection of the exception object also makes it possible to write post mortem script debuggers that merge the call stacks of the two languages and provide cross language diagnostics. Figure 4 shows an example of a simple mixed language debugging session using the WAD post-mortem debugger (wpm) after an extension error has occurred in a Python program. In the figure, the user is first presented with a multi-language stack trace. The information in this trace is obtained both from the WAD exception object and from the Python traceback generated when the exception was raised. Next, we see the user walking up the call stack using the 'u' command of the debugger. As this proceeds, there is a seamless transition from C to Python where the trace crosses between the two languages. An optional feature of the debugger (not shown) allows the debugger to walk up the entire C call-stack (in this case, the trace shows information about the implementation of the Python interpreter). More advanced features of the debugger allow the user to query values of function parameters, local variables, and stack frames (although some of this information may not be obtainable due to compiler optimizations and the difficulties of accurately recovering register values).

11 Implementation Details

Currently, WAD is implemented in ANSI C and small amount of assembly code to assist in the return to the interpreter. The current implementation supports

```
[ Error occurred ]
>>> from wpm import *
*** WAD Debugger ***
#5   [ Python ] in self.widget._report_exception() in ...
#4   [ Python ] in Button(self,text="Die", command=lambda x=self: ...
#3   [ Python ] in death_by_segmentation() in death.py, line 22
#2   [ Python ] in debug.seg_crash() in death.py, line 5
#1   0xfeee2780 in _wrap_seg_crash(self=0x0,args=0x18f114) in 'pydebug.c', line 512
#0   0xfeee1320 in seg_crash() in 'debug.c', line 20

      int *a = 0;
 =>   *a = 3;
      return 1;

>>> u
#1   0xfeee2780 in _wrap_seg_crash(self=0x0,args=0x18f114) in 'pydebug.c', line 512

        if(!PyArg_ParseTuple(args,":seg_crash")) return NULL;
 =>     result = (int )seg_crash();
        resultobj = PyInt_FromLong((long)result);

>>> u
#2   [ Python ] in debug.seg_crash() in death.py, line 5

    def death_by_segmentation():
 =>     debug.seg_crash()

>>> u
#3   [ Python ] in death_by_segmentation() in death.py, line 22

        if ty == 1:
 =>         death_by_segmentation()
        elif ty == 2:
>>>
```

Figure 4: Cross-language debugging session in Python where a user is walking a mixed language call stack.

Python and Tcl extensions on SPARC Solaris and i386-Linux. Each scripting language is currently supported by a separate shared library such as `libwadpy.so` and `libwadtcl.so`. In addition, a language neutral library `libwad.so` can be linked against non-scripted applications (in which case a stack trace is simply printed to standard error when a problem occurs). The entire implementation contains approximately 2000 semicolons. Most of this code pertains to the gathering of debugging information from object files. Only a small part of the code is specific to a particular scripting language (170 semicolons for Python and 50 semicolons for Tcl).

Although there are libraries such as the GNU Binary File Descriptor (BFD) library that can assist with the manipulation of object files, these are not used in the implementation [25]. These libraries tend to be quite large and are oriented more towards stand-alone tools such as debuggers, linkers, and loaders. In addition, the behavior of these libraries with respect to memory management would need to be carefully studied before they could be safely used in an embedded environment. Finally, given the small size of the prototype implementation, it didn't seem necessary to rely upon such a heavyweight solution.

A surprising feature of the implementation is that a significant amount of the code is language independent. This is achieved by placing all of the process introspection, data collection, and platform specific code within a centralized core. To provide a specific scripting language interface, a developer only needs to supply two things; a table containing symbolic function names where control can be returned (Table 1), and a handler function in the form of a callback. As input, this handler receives an exception object as described in an earlier section. From this, the handler can raise a scripting

language exception in whatever manner is most appropriate.

Significant portions of the core are also relatively straightforward to port between different Unix systems. For instance, code to read ELF object files and stabs debugging data is essentially identical for Linux and Solaris. In addition, the high-level control logic is unchanged between platforms. Platform specific differences primarily arise in the obvious places such as the examination of CPU registers, manipulation of the process context in the signal handler, reading virtual memory maps from /proc, and so forth. Additional changes would also need to be made on systems with different object file formats such as COFF and DWARF2. To extent that it is possible, these differences could be hidden by abstraction mechanisms (although the initial implementation of WAD is weak in this regard and would benefit from techniques used in more advanced debuggers such as gdb). Despite these porting issues, the primary requirement for WAD is a fully functional implementation of SVR4 signal handling that allows for modifications of the process context.

Due to the heavy dependence on Unix signal handling, process introspection, and object file formats, it is unlikely that WAD could be easily ported to non-Unix systems such as Windows. However, it may be possible to provide a similar capability using advanced features of Windows structured exception handling [26]. For instance, structured exception handlers can be used to catch hardware faults, they can receive process context information, and they can arrange to take corrective action much like the signal implementation described here.

12 Modification of Interpreters?

A logical question to ask about the implementation of WAD is whether or not it would make sense to modify existing interpreters to assist in the recovery process. For instance, instrumenting Python or Tcl with setjmp functions might simplify the implementation since it would eliminate issues related to register restoration and finding a suitable return location.

Although it may be possible to make these changes, there are several drawbacks to this approach. First, the number of required modifications may be quite large. For instance, there are well over 50 entry points to extension code within the implementation of Python. Second, an extension module may perform callbacks and evaluation of script code. This means that the call stack would cross back and forth between languages and that these modifications would have to be made in a way that allows arbitrary nesting of extension calls. Finally, instrumenting the code in this manner may introduce a performance impact–a clearly undesirable side effect considering the infrequent occurrence of fatal extension errors.

13 Discussion

The primary goal of embedded error recovery is to provide an alternative approach for debugging scripting language extensions. Although this approach has many benefits, there are a number drawbacks and issues that must be discussed.

First, like the C `longjmp` function, the error recovery mechanism does not cleanly unwind the call stack. For C++, this means that objects allocated on stack will not be finalized (destructors will not be invoked) and that memory allocated on the heap may be leaked. Similarly, this could result in open files, sockets, and other system resources. In a multi-threaded environment, deadlock may occur if a procedure holds a lock when an error occurs.

In certain cases, the use of signals in WAD may interact adversely with scripting language signal handling. Since scripting languages ordinarily do not catch signals such as SIGSEGV, SIGBUS, and SIGABRT, the use of WAD is unlikely to conflict with any existing signal handling. However, most scripting languages would not prevent a user from disabling the WAD error recovery mechanism by simply specifying a new handler for one or more of these signals. In addition, the use of certain extensions such as the Perl sigtrap module would completely disable WAD [2].

A more difficult signal handling problem arises when thread libraries are used. These libraries tend to override default signal handling behavior in a way that defines how signals are delivered to each thread [27]. In general, asynchronous signals can be delivered to any thread within a process. However, this does not appear to be a problem for WAD since hardware exceptions are delivered to a signal handler that runs within the same thread in which the error occurred. Unfortunately, even in this case, personal experience has shown that certain implementations of user thread libraries (particularly on older versions of Linux) do not reliably pass signal context information nor do they universally support advanced signal operations such as `sigaltstack`. Because of this, WAD may be incompatible with a crippled implementation of user threads on these platforms.

A even more subtle problem with threads is that the recovery process itself is not thread-safe (i.e., it is not possible to concurrently handle fatal errors occurring in different threads). For most scripting language extensions, this limitation does not apply due to strict run-time restrictions that interpreters currently place on thread support. For instance, even though Python supports

threaded programs, it places a global mutex-lock around the interpreter that makes it impossible for more than one thread to concurrently execute within the interpreter at once. A consequence of this restriction is that extension functions are not interruptible by thread-switching unless they explicitly release the interpreter lock. Currently, the behavior of WAD is undefined if extension code releases the lock and proceeds to generate a fault. In this case, the recovery process may either cause an exception to be raised in an entirely different thread or cause execution to violate the interpreter's mutual exclusion constraint on the interpreter.

In certain cases, errors may result in an unrecoverable crash. For example, if an application overwrites the heap, it may destroy critical data structures within the interpreter. Similarly, destruction of the call stack (via buffer overflow) makes it impossible for the recovery mechanism to create a stack-trace and return to the interpreter. More subtle memory management problems such as double-freeing of heap allocated memory can also cause a system to fail in a manner that bears little resemblance to actual source of the problem. Given that WAD lives in the same process as the faulting application and that such errors may occur, a common question to ask is to what extent does WAD complicate debugging when it doesn't work.

To handle potential problems in the implementation of WAD itself, great care is taken to avoid the use of library functions and functions that rely on heap allocation (malloc, free, etc.). For instance, to provide dynamic memory allocation, WAD implements its own memory allocator using mmap. In addition, signals are disabled immediately upon entry to the WAD signal handler. Should a fatal error occur inside WAD, the application will dump core and exit. Since the resulting core file contains the stack trace of both WAD and the faulting application, a traditional C debugger can be used to identify the problem as before. The only difference is that a few additional stack frames will appear on the traceback.

An application may also fail after the WAD signal handler has completed execution if memory or stack frames within the interpreter have been corrupted in a way that prevents proper exception handling. In this case, the application may fail in a manner that does not represent the original programming error. It might also cause the WAD signal handler to be immediately reinvoked with a different process state–causing it to report information about a different type of failure. To address these kinds of problems, WAD creates a tracefile `wadtrace` in the current working directory that contains information about each error that it has handled. If no recovery was possible, a programmer can look at this file to obtain all of the stack traces that were generated.

If an application is experiencing a very serious problem, WAD does not prevent a standard debugger from being attached to the process. This is because the debugger overrides the current signal handling so that it can catch fatal errors. As a result, even if WAD is loaded, fatal signals are simply redirected to the attached debugger. Such an approach also allows for more complex debugging tasks such as single-step execution, breakpoints, and watchpoints–none of which are easily added to WAD itself.

Finally, there are a number of issues that pertain to the interaction of the recovery mechanism with the interpreter. For instance, the recovery scheme is unable to return to procedures that might invoke wrapper functions with conflicting return codes. This problem manifests itself when the interpreter's virtual machine is built around a large `switch` statement from which different types of wrapper functions are called. For example, in Python, certain internal procedures call a mix of functions where both NULL and -1 are returned to indicate errors (depending on the function). In this case, there is no way to specify a proper error return value because there will be conflicting entries in the WAD return table (although you could compromise and return the error value for the most common case). The recovery process is also extremely inefficient due to its heavy reliance on `mmap`, file I/O, and linear search algorithms for finding symbols and debugging information. Therefore, WAD would unsuitable as a more general purpose extension related exception handler.

Despite these limitations, embedded error recovery is still a useful capability that can be applied to a wide variety of extension related errors. This is because errors such as failed assertions, bus errors, and floating point exceptions rarely result in a situation where the recovery process would be unable to run or the interpreter would crash. Furthermore, more serious errors such as segmentation faults are more likely to caused by an uninitialized pointer than a blatant destruction of the heap or stack.

14 Related Work

A huge body of literature is devoted to the topic of exception handling in various languages and systems. Furthermore, the topic remains one of active interest in the software community. For instance, IEEE Transactions on Software Engineering recently devoted two entire issues to current trends in exception handling [29, 30]. Unfortunately, very little of this work seems to be directly related to mixed compiled-interpreted exception handling, recovery from fatal signals, and problems pertaining to mixed-language debugging.

Perhaps the most directly relevant work is that of advanced programming environments for Common Lisp

[31]. Not only does CL have a foreign function interface, debuggers such as gdb have previously been modified to walk the Lisp stack [33, 34]. Furthermore, certain Lisp development environments have previously provided a high degree of integration between compiled code and the Lisp interpreter[32].

In certain cases, a scripting language module has been used to provide partial information for fatal signals. For example, the Perl `sigtrap` module can be used to produce a Perl stack trace when a problem occurs [2]. Unfortunately, this module does not provide any information from the C stack. Similarly, advanced software development environments such as Microsoft's Visual Studio can automatically launch a C/C++ debugger when an error occurs. Unfortunately, this doesn't provide any information about the script that was running.

In the area of programming languages, a number of efforts have been made to map signals to exceptions in the form of asynchronous exception handling [35, 37, 36]. Unfortunately, this work tends to concentrate on the problem of handling asynchronous signals related to I/O as opposed to synchronously generated signals caused by software faults.

With respect to debugging, little work appears to have been done in the area of mixed compiled-interpreted debugging. Although modern debuggers certainly try to provide advanced capabilities for debugging within a single language, they tend to ignore the boundary between languages. As previously mentioned, debuggers have occasionally been modified to support other languages such as Common Lisp [34]. However, little work appears to have been done in the context of modern scripting languages. One system of possible interest in the context of mixed compiled-interpreted debugging is the R^n system developed at Rice University in the mid-1980's [38]. This system, primarily developed for scientific computing, allowed control to transparently pass between compiled code and an interpreter. Furthermore, the system allowed dynamic patching of an executable in which compiled procedures could be replaced by an interpreted replacement. Although this system does not directly pertain to the problem of debugging of scripting language extensions, it is one of the few examples of a system in which compiled and interpreted code have been tightly integrated within a debugger.

More recently, a couple of efforts have emerged to that seem to address certain issues related to mixed-mode debugging of interpreted and compiled code. PyDebug is a recently developed system that focuses on problems related to the management of breakpoints in Python extension code [39]. It may also be possible to perform mixed-mode debugging of Java and native methods using features of the Java Platform Debugger Architecture (JPDA) [40]. Mixed-mode debugging support for Java may also be supported in advanced debugging systems such as ICAT [41]. However, none of these systems appear to have taken the approach of converting hardware faults into Java errors or exceptions.

15 Future Directions

As of this writing, WAD is only an experimental prototype. Because of this, there are certainly a wide variety of incremental improvements that could be made to support additional platforms and scripting languages. In addition, there are a variety of improvements that could be made to provide better integration with threads and C++. One could also investigate heuristic schemes such as backward stack tracing that might be able to recover partial debugging information from corrupted call stacks [28].

A more interesting extension of this work would be to see how the exception handling approach of WAD could be incorporated with the integrated development environments and script-level debugging systems that have already been developed. For instance, it would be interesting to see if a graphical debugging front-end such as DDD could be modified to handle mixed-language stack traces within the context of a script-level debugger [42].

It may also be possible to extend the approach taken by WAD to other types of extensible systems. For instance, if one were developing a new server module for the Apache web-server, it might be possible to redirect fatal module errors back to the server in a way that produces a webpage with a stack trace [43]. The exception handling approach may also have applicability to situations where compiled code is used to build software components that are used as part of a large distributed system.

16 Conclusions and Availability

This paper has presented a mechanism by which fatal errors such as segmentation faults and failed assertions can be handled as scripting language exceptions. This approach, which relies upon advanced features of Unix signal handling, allows fatal signals to be caught and transformed into errors from which interpreters can produce an informative cross-language stack trace. In doing so, it provides more seamless integration between scripting languages and compiled extensions. Furthermore, this has the potential to greatly simplify the frustrating task of debugging complicated mixed scripted-compiled software.

The prototype implementation of this system is available at :

`http://systems.cs.uchicago.edu/wad`.

Currently, WAD supports Python and Tcl on SPARC Solaris and i386-Linux systems. Work to support additional scripting languages and platforms is ongoing.

17 Acknowledgments

Richard Gabriel and Harlan Sexton provided interesting insights concerning debugging capabilities in Common Lisp. Stephen Hahn provided useful information concerning the low-level details of signal handling on Solaris. I would also like to thank the technical reviewers and Rob Miller for their useful comments.

References

[1] J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proceedings of the USENIX Association Winter Conference, 1990. p.133-146.

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

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

[4] Thomas Lord, *An Anatomy of Guile, The Interface to Tcl/Tk*, USENIX 3rd Annual Tcl/Tk Workshop 1995.

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

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

[7] 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.

[8] P. Thompson, *SIP*, `http://www.thekompany.com/projects/pykde`.

[9] P. F. Dubois, *Climate Data Analysis Software*, 8th International Python Conference, Arlington, VA., 2000.

[10] P. Peterson, J. Martins, and J. Alonso, *Fortran to Python Interface Generator with an application to Aerospace Engineering*, 9th International Python Conference, submitted, 2000.

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

[12] Wolfgang Heidrich and Philipp Slusallek, *Automatic Generation of Tcl Bindings for C and C++ Libraries.*, USENIX 3rd Tcl/Tk Workshop, 1995.

[13] K. Martin, *Automated Wrapping of a C++ Class Library into Tcl*, USENIX 4th Tcl/Tk Workshop, p. 141-148, 1996.

[14] C. Lee, *G-Wrap: A tool for exporting C libraries into Scheme Interpreters*, `http://www.cs.cmu.edu/~chrislee/Software/g-wrap`.

[15] G. Couch, C. Huang, and T. Ferrin, *Wrappy :A Python Wrapper Generator for C++ Classes*, O'Reilly Open Source Software Convention, 1999.

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

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

[18] D.M. Beazley and P.S. Lomdahl, *Feeding a Large-scale Physics Application to Python*, 6th International Python Conference, co-sponsored by USENIX, p. 21-28, 1997.

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

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

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

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

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

[24] D. Weaver and T. Germond, *SPARC Architecture Manual Version 9*, Prentice-Hall, 1993.

[25] S. Chamberlain. *libbfd: The Binary File Descriptor Library*. Cygnus Support, bfd version 3.0 edition, April 1991.

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

[27] F. Mueller, *A Library Implementation of POSIX Threads Under Unix*, USENIX Winter Technical Conference, San Diego, CA., p. 29-42, 1993.

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

[29] 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.

[30] 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.

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

[32] R. Gabriel, private correspondence.

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

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

[35] 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.

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

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

[38] A. Carle, D. Cooper, R. Hood, K. Kennedy, L. Torczon, S. Warren, *A Practical Environment for Scientific Programming.* IEEE Computer, Vol 20, No. 11, p. 75-89, 1987.

[39] P. Stoltz, *PyDebug, a New Application for Integrated Debugging of Python with C and Fortran Extensions*, O'Reilly Open Source Software Convention, San Diego, 2001 (to appear).

[40] Sun Microsystems, *Java Platform Debugger Architecture*, http://java.sun.com/products/jpda

[41] IBM, *ICAT Debugger*, http://techsupport.services.ibm.com/icat.

[42] A. Zeller, *Visual Debugging with DDD*, Dr. Dobb's Journal, March, 2001.

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