Hang Analysis: Fighting Responsiveness Bugs

Xi Wang[†] Zhenyu Guo[‡] Xuezheng Liu[‡] Zhilei Xu[†]
Haoxiang Lin[‡] Xiaoge Wang[†] Zheng Zhang[‡]

[†]Tsinghua University [‡]Microsoft Research Asia

Abstract

Soft hang is an action that was expected to respond instantly but instead drives an application into a coma. While the application usually responds eventually, users cannot issue other requests while waiting. Such hang problems are widespread in productivity tools such as desktop applications; similar issues arise in server programs as well. Hang problems arise because the software contains blocking or time-consuming operations in graphical user interface (GUI) and other time-critical call paths that should not.

This paper proposes HANGWIZ to find hang bugs in source code, which are difficult to eliminate before release by testing, as they often depend on a user's environment. HANGWIZ finds hang bugs by finding hang points: an invocation that is expected to complete quickly, such as a GUI action, but calls a blocking function. HANGWIZ collects hang patterns from runtime traces supplemented with expert knowledge, and feed these patterns into a static analysis framework that searches exhaustively for hang points that involve potential hang bugs.

Experiments with several large, real-world software packages (including a source control client, a graphics editor and a web server) show that there are several hang bugs in these applications, and that HANGWIZ is effective in finding them. The experiments also demonstrate that HANGWIZ is scalable and can analyze millions of lines of code. We further discuss related techniques and report our experience on fixing hang bugs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics; D.4.7 [*Operating System*]: Organization and Design; D.4.8 [*Operating System*]: Performance

General Terms Experimentation, Performance, Reliability

Keywords hang, responsive invocation, blocking invocation, interactive performance, responsiveness, program analysis

1. Introduction

Almost every user of modern computer software has had the annoying experience of the unresponsiveness problem known as *soft hang*. In such scenarios, a mouse click suddenly drives an application into a coma, and the operating system (OS) may declare the application to be "not responding". The application will *eventually* return to life. However, during the long wait, the user can neither cancel the operation nor close the application. The user may have to kill the application or even reboot the computer.

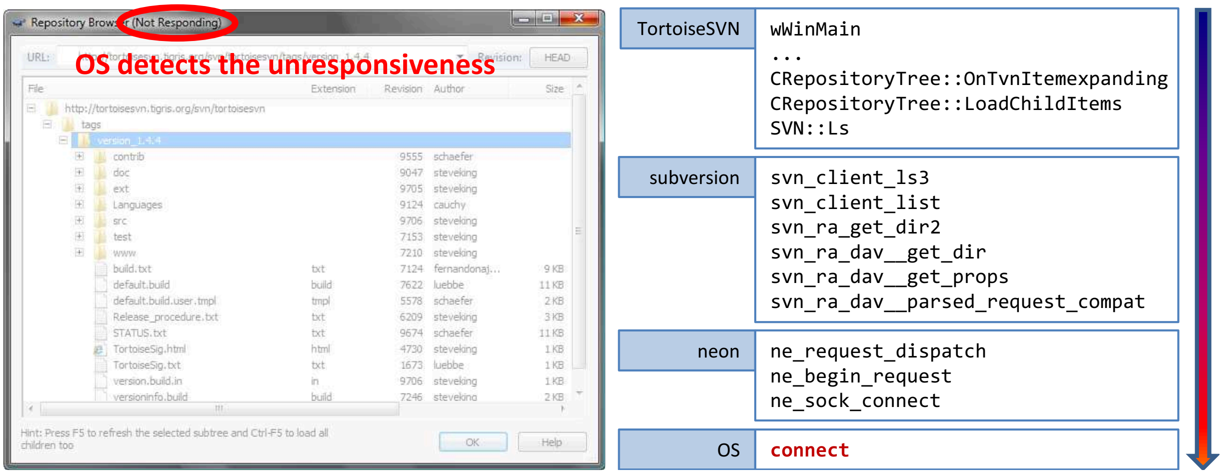
The problem of "not responding" is widespread. In our experience, hang has occurred in everyday productivity tools, such as office suites, web browsers and source control clients. The actions that trigger the hang issues are often the ones that users expect to return instantly, such as opening a file or connecting to a printer. As shown in Figure 1(a), TortoiseSVN repository browser becomes unresponsive when a user clicks to expand a directory node of a remote source repository.

The causes of "not responding" bugs may be complicated. An erroneous program that becomes unresponsive may contain deadlocks (Engler and Ashcraft 2003; Williams et al. 2005), infinite loops or many other correctness bugs that relate to the problem of termination (Cook et al. 2006, 2007). However, a large category of responsive problems, such as the *soft hang* that we will address in this paper, are *not* triggered by such correctness bugs. Consider the case of TortoiseSVN repository browser we just described. Figure 1(b) illustrates a snapshot of the call stack of the user interface (UI) thread, where the UI thread synchronously invokes a socket API function `connect` to establish a network connection to a remote source repository. Since the UI thread waits for the return of `connect`, further message processing such as the UI rendering task is blocked. This is the reason that it stopped responding. However, the UI thread will be unblocked eventually when the `connect` call timeouts or completes. This is not a correctness bug. Yet, it should most certainly be rewritten to be more user-friendly. The application should have performed network communications in a separate thread, popping up a dialog box to indicate the status and progress, and optionally allow the user to cancel the operation.

In addition to GUI applications, similar hang problems pervade in a broader range of systems, such as event-driven servers and chained filters. An event-driven server should avoid blocking on long-running operations; if there is such an operation in an event handler, subsequent operations such as accepting new connections and serving file contents cannot be processed until the long-running operation completes. Such defects lead to responsiveness problems and performance degradation (Pai et al. 1999; Elmeleegy et al. 2004). Similarly, in the case of chained filters, the end-to-end latency may also become significantly worse if an intermediate filter performs an operation that takes a long time. This is usually caused by third-party extensions or plug-ins that can slow down the system and cause responsiveness problems (Basu et al. 2007).

In this paper, we consider the particular category of responsiveness bugs that generally involve *no* correctness problems, namely *hang* bugs. In such abnormal cases, a program blocks on an operation that consumes a perceptible interval to finish, and unnecessarily prevents subsequent critical operations from being processed. This can lead to the loss of responsiveness and poor user experiences, or even degradation of performance. Hang bugs are becom-

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.
Copyright © 2008 ACM 978-1-60558-013-5/08/04. . . $5.00



Repository Browser (Not Responding)

OS detects the unresponsiveness

URL: http://tortoisesvn.tigris.org/svn/tortoisesvn/tags/version_1.4.4 Revision: HEAD

File	Extension	Revision	Author	Size
http://tortoisesvn.tigris.org/svn/tortoisesvn				
tags				
version_1.4.4				
contrib		9555	schaefer	
doc		9047	steveking	
ext		9705	steveking	
Languages		9124	cauchy	
src		9706	steveking	
test		7153	steveking	
www		7210	steveking	
build.txt	txt	7124	fernandonaj...	9 KB
default.build	build	7622	luebbe	11 KB
default.build.user.tmpl	tmpl	5578	schaefer	2 KB
Release_procedure.txt	txt	6209	steveking	3 KB
STATUS.txt	txt	9674	schaefer	11 KB
TortoiseSig.html	html	4730	steveking	1 KB
TortoiseSig.txt	txt	1673	luebbe	1 KB
version.build.in	in	9706	steveking	1 KB
versioninfo.build	build	7246	steveking	2 KB

Hint: Press F5 to refresh the selected subtree and Ctrl-F5 to load all children too

(a) TortoiseSVN repository browser (not responding).

(b) A snapshot of the UI thread's call stack.

Figure 1: A simple hang case of TortoiseSVN 1.4 repository browser. When a user clicks in the tree view to expand a directory node of a remote source repository, the UI thread calls back to the event handler function `CRepositoryTree::OnTvnItemexpanding` in TortoiseSVN, which eventually invokes a blocking socket API function `connect`. Thus, its UI stops responding and the application hangs during network communication.

ing one of the most significant issues. A recent survey by the Microsoft Office team has indicated that user complaints about hang bugs are nearly as frequent as complaints about crashes.

Hang bugs are difficult to eliminate before release. In the hunt for the cause of hang, we lack effective tools. We are aware of the testing approach that catch abnormal states when an invocation takes longer than a pre-specified period. While useful, it has a number of drawbacks. First, tests in the lab do not cover all cases that trigger hang bugs due to complicated interactions between modules in several threads. More important, hang bugs are often sensitive to environments, e.g. special inputs and network conditions, which are hard to reproduce. Static analysis tools that can exhaust all call paths do not have the vulnerability of a limited testing environment. However, the lack of a formal model of hang bugs has so far hindered the use of static analysis.

To the best of our knowledge, this paper is the first to address the issue of soft hang in real-world applications. As we will demonstrate, instead of relying on testing to uncover hang bugs, we collect patterns as expert knowledge, which are fed into a model that our static analysis framework uses to find hang bugs. We believe that it has established the foundation of defining an ecosystem whereby we can gradually increase both accuracy and coverage.

1.1 Contributions

In this paper, we identify the problem of hang and discuss techniques to solve it. Specifically, we make the following contributions.

- We propose a general hang model to describe this particular category of responsiveness bugs. Based on the model, we collect patterns as the generalization of hang cases found in testing and client-side traces. Such patterns can also be specified by domain experts.
- We present an extensible framework to perform a precise and scalable static analysis on source code to find hang bugs based on collected patterns. Our approach has successfully analyzed software up to one million lines of code while having an acceptable false positive rate.
- We implement a hang analysis system HANGWIZ, and report our experience on analyzing various large, real-world software. We also discuss related techniques and design principles to solve hang problems.

The rest of the paper is organized as follows. Section 2 presents the hang model and Section 3 illustrates the architecture of our hang analysis system. Section 4 describes our hang analysis algorithm, while Section 5 explains implementation details. Section 6 evaluates our hang analysis system on various software. We report our experience in Section 7, and further discuss extensions in Section 8. We survey related work in Section 9, then conclude in Section 10.

2. Model

In this section, we present our hang model to describe operations that may cause hangs. We consider invocations, i.e. calls to functions, as the basic unit of operations, since the next level of operations such as memory access and computation statement are unlikely to cause hangs. Extensions are discussed in Section 8.

As discussed in Section 1, there are some time-critical invocations that should or are expected to complete in a timely fashion in some specific contexts (e.g. invocations in the UI thread). Such invocations are *responsive invocations*. Meanwhile, there are some time-consuming invocations that would block for a perceptible interval (e.g. `connect`). Such invocations are *blocking invocations*. If the call path of a responsive invocation contains a blocking invocation, this will produce a hang. In other words, a hang is uncovered if a responsive invocation (as expected by design) meets a blocking invocation (as manifested by the runtime system). Our model starts by defining responsive patterns and blocking patterns to capture responsive invocations and blocking invocations, respectively.

2.1 Responsive Invocations

Responsive invocations are domain-specific, such as invocations to message processing methods in GUI applications and those to event handlers in event-driven systems. The responsive property propagates down along the call graph: if an invocation to a function needs to be responsive, then all invocations inside the function to other functions should also be responsive.

As in Figure 1, wWinMain is the entry of the UI thread. So that the invocation to one of its callees, the GUI message processing method CRepositoryTree::OnTvnItemexpanding that is responsible to expand a directory view, should be responsive. Therefore, the invocations deep down the call graph to functions such as connect are also responsive in this thread. On the contrary, if connect is called in a separate thread other than the UI thread, the invocation is not responsive since its callers are not responsive.

We collect responsive patterns that capture these responsive invocations. Here is a common responsive pattern for GUI applications.

PATTERN 2.1. *An invocation to any function in the UI thread is responsive.*

Similarly, we can specify more responsive patterns for other systems to capture domain-specific responsive invocations, such as invocations to event handlers or filters.

2.2 Blocking Invocations

On the other hand, blocking invocations may synchronously consume much time to finish. They can be invocations to sleep and wait functions, networking and inter-process communications, interactions with intensive I/O or computations, etc.

We collect blocking patterns to capture blocking operations. Here is a blocking pattern for the socket API function connect in Figure 1.

PATTERN 2.2. *An invocation to* connect *is blocking.*

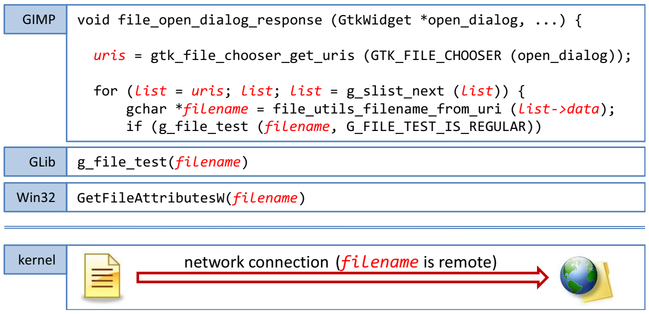
Note that in this work, we consider blocking APIs only and ignore other runtime performance problems caused by OS side effects such as paging/swapping. An API is blocking if there exists a worst case scenario that prevents the calling logic from making progress until timeout. In the above example, if the network environment is perfect, the operation will complete shortly and a user does not feel a lag. However, if the connection endpoint is over wide-area or is simply down, this is a blocking call.

Blocking patterns such as Pattern 2.2 are unconditional; they do not depend on calling contexts or function parameters, and always identify blocking invocations. However, some invocations may be blocking or not in different contexts, i.e. they are sensitive to properties of function parameters or other contexts. Consider an invocation to the Win32 API function GetFileAttributesW (or the POSIX API function stat) that retrieves file information. It may be blocking, if the file path is remote (e.g. \\server\directory\file); or it may be instant, if the file path refers to a local-disk file (e.g. application configuration file). The latter case should be pruned. A naive unconditional blocking pattern that states all invocations to GetFileAttributesW or stat are blocking will cause too many false alarms.

We introduce a more precise category of blocking patterns, namely *conditional* blocking patterns, which are attached with blocking conditions on parameters. For example, since a file path that reads from user inputs (e.g. a file-open dialog) may be remote, we can specify a conditional blocking pattern as follows.

PATTERN 2.3. *An invocation to* GetFileAttributesW *is blocking if the file path is may-remote.*

The pattern places a restriction on the parameter for the file path, so it captures fewer and more precise invocations compared to an unconditional counterpart. The may-remote property in Pattern 2.3 can also be applied to other blocking patterns of file operation API functions.



```
void file_open_dialog_response (GtkWidget *open_dialog, ...) {

  uris = gtk_file_chooser_get_uris (GTK_FILE_CHOOSER (open_dialog));

  for (list = uris; list; list = g_slist_next (list)) {
      gchar *filename = file_utils_filename_from_uri (list->data);
      if (g_file_test (filename, G_FILE_TEST_IS_REGULAR))
```

```
g_file_test(filename)
```

```
GetFileAttributesW(filename)
```

Figure 2: A more sophisticated hang case of GIMP when a file opens after a user inputs or selects a file in a file-open dialog. The code snippet usually does not cause hangs. However, if a user inputs a remote path filename that is passed to g_file_test, GetFileAttributesW and eventually processed by the kernel, the kernel will establish network connections, take some time to communicate over networks and prevent subsequent operations such as UI rendering; thus the application may hang.

In fact, an unconditional blocking pattern can be thought of as a special blocking pattern, predicated by a Boolean evaluation that will always turn out to be true. There is a wide category of conditional blocking patterns. Table 1 lists some examples. Sleep(0) is a popular trick to perform a yield and it does not block. TransmitFile performs asynchronous I/O operations and does not block, *if* the lpOverlapped parameter is non-null. We use conditional blocking patterns to prune the cases that do not meet their blocking conditions to reduce false alarms.

2.3 Hang Bugs

To discover hang bugs, we analyze source code, using responsive and blocking patterns to compute responsive and blocking invocations, respectively. Invocations in their intersection are potential hang bugs. In other words, a hang bug is an invocation that is both responsive and blocking.

For a simple example, as in Figure 1, assume that we have Pattern 2.1 that an invocation to any function in the UI thread is responsive, and Pattern 2.2 that an invocation to connect is blocking. Since the invocation to connect is in the UI thread, it is responsive; while the invocation to connect is also blocking. This is a hang.

Figure 2 gives a more sophisticated hang case. Again, assume that we have Pattern 2.1 that an invocation to any function in the UI thread is responsive, and Pattern 2.3 that an invocation to GetFileAttributesW is blocking *if* the file path is may-remote. As the string uris returned by gtk_file_chooser_get_uris is from an input dialog outside the program, it is may-remote. The may-remote property can be propagated to filename via list, such that filename is also may-remote. Since the invocation to file_open_dialog_response is in the UI thread, the invocation to GetFileAttributesW is responsive; while the invocation to file_open_dialog_response with a may-remote filename is blocking. Thus it is a hang bug.

Section 4 will give a detailed description of the analysis algorithm to compute responsive and blocking invocations for finding hang bugs.

3. Architecture

Figure 3 illustrates the overall architecture of our hang analysis system HANGWIZ, which consists of the following components.

property	example	brief description	blocking condition

may-remote	`GetFileAttributesW(`
	`lpFileName)`
		get file information	`lpFileName` is remote

may-nonzero	`Sleep(`
	`dwMilliseconds)`
		sleep for an interval	`dwMilliseconds` is nonzero

may-null	`TransmitFile(`
	`...,lpOverlapped,...)`
		send file data over a socket asynchronously, or synchronously if `lpOverlapped` is null	`lpOverlapped` is null

Table 1: Examples of conditional blocking patterns. `GetFileAttributesW`, `Sleep` and `TransmitFile` are Win32 API counterparts to POSIX API `stat`, `sleep` and `sendfile`, respectively.

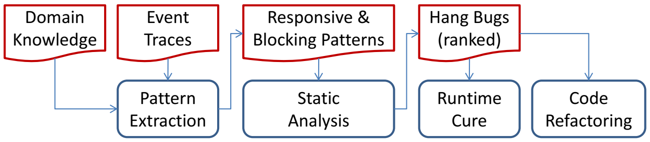


Figure 3: Architecture of the hang analysis system.

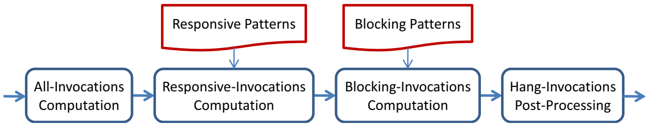

Responsive Patterns
Blocking Patterns
All-Invocations Computation
Responsive-Invocations Computation
Blocking-Invocations Computation
Hang-Invocations Post-Processing

Figure 4: Stages of the hang analysis algorithm.

- *Pattern extraction.* This step collects crucial patterns for the later stage to identify responsive and blocking invocations. These patterns can be specified by developers or testers using their expert knowledge based on event traces of running software. These patterns are independent of specific applications and are applicable to analyzing other software systems.
- *Static analysis.* This is the heart of our hang analysis system and also the focus of this paper. We first extract the set of all invocations from source code via a compiler plug-in. With the inputs of the responsive patterns and blocking patterns, we reduce the set of all invocations into responsive invocations and blocking invocations, respectively, and compute the intersection. The final result is a set of approximation of hang bugs.
- *Cure and refactoring.* The output of the previous stage is a hang bug report. Generally speaking, hang bugs should be fixed by code refactoring. From our experience of identifying hang bugs in a number of large, real-world applications, we offer summary of typical hang root causes and provide suggestions for fixes. For legacy applications, code refactoring may not always be possible. Thus, we have investigated techniques to cure hangs at runtime if a blocking invocation can be safely canceled.

4. Algorithm

The general idea of the algorithm is to start from all invocations of a program, and to compute responsive invocations and blocking invocations based on their corresponding patterns. Their intersection, the set of hang invocations, is the approximation to hang bugs.

Figure 4 illustrates the stages of the analysis algorithm to successively prune invocations that are out of interest, including:

1. all-invocations computation,
2. responsive-invocations computation,
3. blocking-invocations computation, and
4. hang-invocations post-processing.

4.1 Program Analysis as Database Queries

Our analysis leverages database techniques. Consider a simple example, to compute the set of functions that the `main` function can reach. Let $\mathbb{F}$ be the set of all functions. A $call$ relation (caller, callee) can be viewed as a database relation in the form $\mathbb{F} \times \mathbb{F}$. For two functions f and g, f calls g if and only if $(f, g) \in call$.

First, a compiler plug-in extracts the $call$ relation from source code: at each call site, it emits the corresponding pair for (caller, callee). After constructing the $call$ relation, to compute the functions that `main` can reach turns out to be a *recursive* query over the $call$ relation.

$$Reach = \{f|(\mathtt{main}, f) \in call\} \cup \{f|\exists g \in Reach : (g, f) \in call\}$$

Note that the definition of the query $Reach$ depends on itself, so it is recursive.

Such queries can be expressed in a logic programming language. A logic database can solve the queries over corresponding database relations and produce new resulting relations. Specifically, our analysis is expressed in the Datalog query language (Ullman 1989). Datalog enables recursive queries and can be evaluated in polynomial time. It is efficient for large databases and popular for program analysis (Reps 1994). Datalog evaluation and optimization techniques are beyond the scope of this paper; please refer to (Ullman 1989; Whaley and Lam 2004) for details.

Thus, performing analysis to a program is converted to the practice of constructing database relations and applying queries over them. This is how we will describe our algorithm in the rest of this section. To avoid introducing the syntax details of the Datalog query language, we use the set notation as above in our analysis. We use the terms *set* and *relation* interchangeably. Note that the set notation is *not* formalization but database queries over relations.

4.2 All-Invocations Computation

First of all, we compute a call graph for all invocations as the basis for subsequent computations. As discussed in Section 2.2, a function parameter may have a property that depends on specific contexts, e.g. a file path can only be may-remote on specific call paths. An invocation with such a parameter, e.g. `GetFileAttributesW` with a file path, may be blocking or not on different calling paths. To precisely analyze these cases, we should compute the properties on individual call paths. In other words, we need a context-sensitive graph.

We adopt a cloning-based context-sensitive analysis (Whaley and Lam 2004) to construct a context-sensitive call graph. In this method, we first compute an initial context-insensitive call graph, as shown in the left graph in Figure 5. Then, strongly connected components (i.e. direct or indirect recursive calls) are reduced into one node, and a clone is made for each new context. This effectively

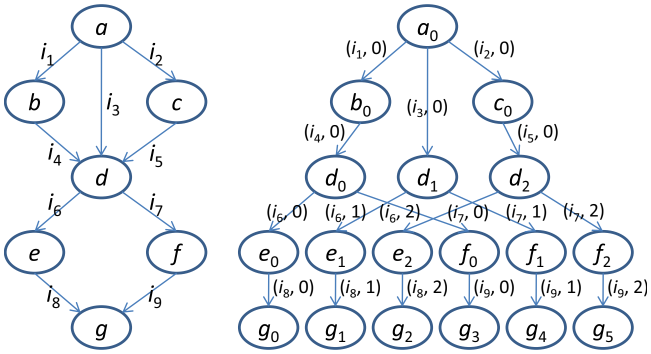


Figure 5: Example of cloning-based construction of a context-sensitive call graph. The left graph is a context-insensitive call graph. The right graph is a context-sensitive call graph with all call paths expanded.

transforms a context-insensitive call graph to an expanded one that is context-sensitive, as shown in the right graph in Figure 5. In a context-sensitive call graph, we can distinguish function parameter properties on each clone and perform a more precise analysis to reduce false alarms.

However, the cloning-based approach would lead to a large number of calling contexts. The key to achieve scalability is to represent this exponential call graph as database relations compactly with an efficient data structure. Binary decision diagram (BDD) (Bryant 1986) is widely used for this purpose. It has been known to successfully analyze large programs up to 5×10^{23} paths in a context-sensitive call graph using a BDD-based Datalog database (Whaley 2007).

We compute a context-sensitive call graph for each thread using the cloning-based approach. For example, the root of the call graph of the main thread can be the `main` function for a C/C++ program. The call graph yields the set of all invocations $AllIvk$, in the form of tuples of (i, c):

- $i \in \mathbb{I}$ is a call site where a function is called. The caller is the function that contains the call site, denoted as $\mathcal{H}(i)$; while the callee is the function being called, denoted as $\mathcal{F}(i)$.
- $c \in \mathbb{C}$ is a calling context. If there are n different call paths to a function from the root, their calling contexts will be expanded from 0 for $n - 1$.

Thus, each edge in a context-sensitive call graph represents an invocation (i, c). As in Figure 5, the call site i_6 from d to e is expanded as $(i_6, 0), (i_6, 1)$ and $(i_6, 2)$ in a context-sensitive call graph. For each invocation (i, c), we can identify a unique call path that reaches it from the root.

All subsequent computations are based on (i, c) tuples from all invocations $AllIvk$.

4.3 Responsive-Invocations Computation

This stage computes the subset of all invocations that are time-critical, i.e. responsive invocations. Each responsive invocation has one of the following attributes:

1. It is an invocation captured by a responsive pattern, which is domain-specific.
2. It is an invocation that can be reached from a known responsive invocation along a call path.

The initial set of responsive invocations $ResponsiveIvk_0$ is captured by a group of responsive patterns $ResponsiveRule_k$ from all invocations $AllIvk$. Then we perform reachability analysis to compute the transitive closure $ResponsiveIvk$, in two steps as follows.

$$\begin{aligned}
&R_k = ResonsiveRule_k(AllIvk)\\
&ResponsiveIvk_0 = \bigcup R_k\\
&ResponsiveIvk = \{(i, c) | (i, c) \in AllIvk\\
&\quad \wedge \exists (i', c') \in ResponsiveIvk_0 : (i', c') \longrightarrow^* (i, c)\}
\end{aligned}$$

First, responsive patterns are specified as queries over (i, c) tuples. For example, Pattern 2.1 that an invocation to any function in the UI thread is responsive can be specified as the following query.

$$R_1 = \{(i, c) | (i, c) \in AllIvk \wedge \mathcal{H}(i) = \texttt{main}\}$$

Here we assume that `main` is the entry function of the UI thread, and all invocations in `main` are the initial responsive invocations.

Then we can compute the set of all responsive invocations $ResponsiveIvk$ as the transitive closure of $ResponsiveIvk_0$, the initial set of responsive invocations captured by responsive patterns such as R_1.

Let $(i', c') \longrightarrow^* (i, c)$ denote that invocation (i, c) is reachable from invocation (i', c'), i.e. there is a call path from (i', c') to (i, c) on a context-sensitive call graph. We perform a reachability analysis to find all responsive invocations $ResponsiveIvk$ that $ResponsiveIvk_0$ can reach. Since the responsive property can be propagated down along call paths, if (i', c') is responsive and $(i', c') \longrightarrow^* (i, c)$, (i, c) is also responsive. We iterate to construct $ResponsiveIvk$ until reaching a fixed point.

4.4 Blocking-Invocations Computation

This stage computes $BlockingIvk$, the union of blocking invocations, in which each invocation is captured by some blocking pattern $BlockingRule_k$.

$$\begin{aligned}
&B_k = BlockingRule_k(AllIvk)\\
&BlockingIvk = \bigcup B_k
\end{aligned}$$

A blocking pattern is specified in the form of queries over (i, c) tuples as two parts: a call to a specific function, and an optional blocking condition on function parameters. Following are two examples of blocking patterns: Pattern 2.2 that an invocation to `connect` is blocking, and Pattern 2.3 that an invocation to `GetFileAttributesW` is blocking if the file path is may-remote.

$$\begin{aligned}
&B_1 = \{(i, c) | (i, c) \in AllIvk \wedge \mathcal{F}(i) = \texttt{connect}\}\\
&B_2 = \{(i, c) | (i, c) \in AllIvk \wedge \mathcal{F}(i) = \texttt{GetFileAttributesW}\\
&\quad \wedge (\mathcal{P}(i, 1), c) \in may\text{-}remote\}
\end{aligned}$$

We can see that the blocking pattern for `connect` is quite intuitive, while a conditional blocking pattern such as Pattern 2.3 requires to specify additional restrictions on function parameters to capture blocking invocations precisely. Now we explain the latter one in details.

Let $\mathcal{P}$ be a map that $\mathcal{P}(i, n)$ returns the variable of the n-th function parameter at a given call site i. Consider an invocation to `GetFileAttributesW` at call site i. $\mathcal{P}(i, 1)$ returns the variable for the first parameter, i.e. the file path. If $(\mathcal{P}(i, 1), c)$ is in set $may\text{-}remote$, i.e. the file path parameter $\mathcal{P}(i, 1)$ in calling context c is may-remote, the invocation at call site i to `GetFileAttributesW` in calling context c will be put in $BlockingIvk$.

The question now is how to compute the set $may\text{-}remote$, a set of variables that may be remote file paths in a context-sensitive way. It is in the form $\mathbb{V} \times \mathbb{C}$, where $\mathbb{V}$ is the set of variables and $\mathbb{C}$ is the set of calling contexts. We compute $may\text{-}remote$ as follows.

In addition to checking constant strings for file paths, we annotate parameters or return values of certain functions as may-remote (e.g. a function that reads and returns a string from user inputs), and estimate an initial set of $may\text{-}remote$ variables. By following up operations such as assignments, we propagate the may-remote property from variables in $may\text{-}remote$ to other variables, and iteratively put them in $may\text{-}remote$, in a recursive manner. Take Figure 2 for an example. Assume that by annotation we know that the return value of `gtk_file_chooser_get_uris` is may-remote. Thus `uris` with its calling context is in the set $may\text{-}remote$, and our computation incrementally adds `list`, `list->data` and `filename` with their calling context into $may\text{-}remote$ via propagations. This process repeats until it converges.

In summary, conditional blocking patterns are evaluated in the following steps.

1. Compute variables that have certain properties starting from function annotations, e.g. the return value is read from network or file. This step is required only for computing proprieties such as may-remote; other properties such as may-null can be inferred automatically from source code.
2. Compute a set of variables that have the property via propagations, e.g. variable assignments and parameter passing.
3. Combined with call site information, evaluate blocking conditions of invocations by inspecting the property of corresponding parameters.

Similarly, we estimate sets in the form $\mathbb{V} \times \mathbb{C}$ for other properties such as may-nonzero and may-null listed in Table 1, and evaluate corresponding blocking conditions. Most such properties can be directly inferred from code, so no annotations are required. Besides, if a blocking pattern requires conditions on multiple parameters, it can specify a conjunction of blocking conditions of the parameters.

4.5 Hang-Invocations Post-Processing

The intersection of responsive invocations $ResponsiveIvk$ and blocking invocations $BlockingIvk$ is hang invocations $HangIvk$, our approximation to hang bugs. Each hang invocation is in the form (i, c), a call site i in calling context c, so we can identify a hang call site i with a call path that can reach from a root function, e.g. `main`.

The results generate a hang bug report after a few more post-processing steps. First, we merge hang invocations that are on the same call path, i.e. the same hang bug that is captured on more than one invocation by different patterns. We either manually inspect a call path that leads to a hang bug or feed a call path into more expensive analysis and testing tools (Ball and Rajamani 2002; Henzinger et al. 2002; Das et al. 2002; Xie and Aiken 2005; Godefroid et al. 2008) to check its feasibility.

In addition, we rank hang bugs with the following significance factors.

- *Hot call site.* If a call site causes hangs in several calling contexts, it is hot. Intuitively, if the hang bug at the hot call site gets fixed, more hang invocations get cured.
- *Hot call path.* Modern software may have integrated built-in diagnosis mechanisms to monitor user actions. Some UI actions are triggered more often than others. This can be perceived as feedback that gives weights to responsive invocations. Such weight distribution assigns priority on which hang bugs to fix first.

Modern software is composed by many modules and libraries. A complete application often links its code with many external packages. A developer's main interest is to fix bugs in the application's code. We define *hang gate*, the first *external* function on a hang call path. Fixing a hang gate limits the developer's code refactoring effort within its own logic. Interestingly, as we will illustrate through our experiments, hang gates are usually hot call sites as well. This is in fact quite natural since hang gates are interfaces and (often) entry points to the lower-level functions provided by external libraries. As in Figure 1, `svn_client_ls3` is a hang gate for the case in TortoiseSVN, while `connect` at the bottom is where the hang would occur, in one of its external libraries.

4.6 Discussion

As we have discussed in Section 4.1, the analysis algorithm is expressed as queries in a Datalog program.

- Responsive-invocations computation is a recursive query over call edges in a call graph. Responsive patterns provide entries of critical functions; our analysis computes invocations that are reachable from the entries and prunes those that are out of interest.
- Blocking-invocations computation consists of recursive queries over dataflow edges (e.g. assignments, parameter passing). For blocking patterns conditioned on properties of parameters (e.g. may-remote), our analysis propagates properties along dataflow edges.

A Datalog implementation will apply optimizations to the queries automatically, and evaluate them efficiently to produce responsive and blocking invocations. The intersection of these two sets are hang bugs.

We employ an interprocedural context-sensitive analysis. As we will explain in the evaluation section, a completely context-sensitive analysis is neither practical nor required for large, real-world applications. The applications that we have examined contain millions of lines of code. This poses serious challenges in terms of memory usage, even with an efficient logic database implementation. Motivated by the observation that a developer cares far less about external libraries than main application logic, we adopt a hybrid approach. Essentially, we perform context-sensitive analysis on the software internal code, and context-insensitive analysis on external libraries that are deep down the call paths. The trade-off is that sometimes we will see more false positives, in the sense that properties such as may-remote are not propagated beyond the hang gates.

In addition to false positives raised by the above practical constraints, we may flag false positives for other reasons. Our algorithm is flow-insensitive, and it conservatively computes call graphs and propagate properties for blocking conditions. As such, there can be no feasible paths that lead to a reported hang invocation. Besides, if blocking patterns are too conservative, i.e. missing precise blocking conditions, we may also report false hang bugs.

Similar to previous static analysis work, we may miss some hang bugs in practice, i.e. false negatives. They happen in the following cases.

- Our collections of responsive pattern and blocking pattern are incomplete. A "sound" alternative is to specify non-responsive patterns and non-blocking patterns instead, and to prune those invocations captured that obviously do not cause hangs. However, this alternative is infeasible and nonintuitive, since neither experts nor tools would be able to collect such patterns. We believe that our approach allows incremental addition of pattern collection and refinement once the system is tested in the real world, and thus it will become more complete over time.
- Evaluating blocking conditions for some conditional blocking patterns relies on computing sets like *may-remote*. This requires annotations on function parameters or return values of proper-

		executables		LOC			
	version	main	plug-ins	internal	external	total	brief description
TortoiseSVN	1.4.5	5	1	254K	918K	1.1M	network client
GIMP	2.4.0	1	167	864K	198K	1.0M	graphics editor
lighttpd	1.5.0	1	42	62K	311K	0.3M	web server

Table 2: Benchmarks. The "executables" columns list the numbers of main programs and plug-ins. The "LOC" column lists lines of internal code, external dependent code and those in total.

	analysis	reported	false	hangs	
	time	invocations	alarms	invocations	gates
TortoiseSVN	17m30s	229	77	152	26
GIMP	1h29m57s	69	35	34	10
lighttpd	3m19s	33	9	24	10
Total	–	331	121	210	46

Table 3: Experimental results.

ties like may-remote. If the annotations are incomplete, we may miss corresponding hang bugs.

- For a practical tool that needs to deal with large, complex software package, our implementation has to make a few compromises. For instance, we track function pointers interprocedurally with a best-effort approach. To avoid too many false alarms, we also ignore any unresolved call sites instead of assuming that they will be calls to any functions. These compromises will lead additional missing hang bugs.
- A traditionally difficult problem of practical static analysis tools is to capture implicit invocations, such as callbacks from operating systems, dynamic loading techniques (e.g. shared libraries and reflections). It requires additional efforts to capture these implicit invocations for a more complete call graph, e.g. to annotate on functions or to perform reflection analysis (Livshits et al. 2005). Thus, we may compute an incomplete call graph on unresolved call sites and miss some invocations that involve hang bugs.
- For open programs such as plug-ins, the callee code is missing. Consequently, the call graph is also incomplete in such cases. We can compute a more complete call graph for open programs using domain-specific expert knowledge.

5. Implementation

We use the Phoenix compiler framework (pho) to analyze programs and dump their intermediate representations (IR). As Phoenix enables us to analyze C/C++ sources and C#/.NET bytecode, we develop a back-end plug-in to extract database relations based on the Phoenix IR, including variable assignments, memory accesses, invocations, parameter passing, and other instructions.

In our current implementation, responsive patterns are specified by experts. Blocking patterns are collected from two sources: surveys by product teams provide frequent blocking patterns on Windows; we also inspect suspicious runtime traces that cause hang problems on several popular desktop applications such as Microsoft Office, and manually extract blocking patterns from them. Blocking conditions for conditional blocking patterns are further refined by experts based on API reference documentations.

Responsive patterns, blocking patterns, and our analysis algorithm described in Section 4 are all expressed as Datalog queries. We solve them over database relations extracted from source code using the bddbddb database (Lam et al. 2005), an effective Datalog implementation based on binary decision diagrams that has been proven to scale to large programs.

As discussed in Section 4.6, our analysis algorithm requires annotations to compute set $may\text{-}remote$ for variables that may be remote file paths, and to capture implicit callbacks from operating systems. Besides manual effort, we also obtain such annotations from Windows header files, since most of them are well annotated using the Standard Annotation Language (SAL) (Hackett et al. 2006).

6. Evaluation

We apply our hang analysis system HangWiz to a number of large applications summarized in Table 2. These applications include both main programs and plug-ins. Our experiments are conducted on a Windows Server 2003 x64, with Intel Xeon 2.0 GHz CPU and 32 GB memory. As the Datalog implementation bddbddb we used is implemented in Java, we specify the maximum heap size by command-line parameters to reduce garbage collection time for a better performance.

We will describe the responsive patterns used in the experiments individually. We use 102 blocking patterns (53 unconditional and 49 conditional) in these experiments. As we have discussed, they are general and independent of applications.

Our results are summarized in Table 3, including the analysis time, the number of reported hang invocations, false alarms, hang invocations and the corresponding hang gates. Overall, these applications are very large and contain millions of lines of code; we have succeeded in finding many hang bugs and the false positive rates are moderate.

For each experiment, we summarize top hang gates , including the number of call paths that reach a gate, the function names of hang gates, and the blocking patterns that capture the invocations at the bottom of call paths. We also report the call depth from a top invocation to a gate and the remaining call depth to a bottom invocation on a call graph (denoted as $\top/\bot$).

6.1 Network Client: TortoiseSVN

TortoiseSVN is a popular source control client software. It contains several GUI applications to synchronize between local working copies and their remote repository sources, and to support other file operations such as merge and diff. In addition, it provides a shell extension as a Microsoft COM component that plugs into Windows Explorer. As a typical network client, it is multi-threaded,

hit	gate	bottom	⊤/⊥
24	svn_client_open_ra_session	connect	3/7
17	svn_client_proplist2	connect	3/8
15	svn_client_cat2	connect	5/8

Table 4: Top TortoiseSVN hang gates.

and operates on files on local disks and communicates with remote servers.

TortoiseSVN depends on the portable Subversion library, which further relies on numerous other software packages, including Apache Portable Runtime and OpenSSL. These libraries contain more than 900K lines of code, and the complete application has more than 1.1 millions lines of code.

We use Pattern 2.1 (an invocation to any function in the UI thread is responsive) as the responsive pattern for the GUI applications. Windows GUI applications have a fairly standard structure, and we add implicit callbacks to UI message processing methods from the Windows operating system. For the shell extension, we mark invocations inside the public interface methods as responsive. We have found 152 hang invocations at 26 hang gates. Top hang gates are summarized in Table 4.

All the hang gates capture network invocations in TortoiseSVN UI threads. This is what we have shown in Figure 1: clicking to expand a repository tree would cause a hang. We follow the call paths and find another more serious problem: refreshing in the repository browser would make it become unresponsive for a much longer time, even up to several minutes.

There are 77 false alarms, most of which are due to invocations to GetFileAttributesW with a file path from outer environments. While we consider file paths from outer environments as may-remote, and according to Pattern 2.3 all invocations to GetFileAttributesW with a may-remote file path are blocking, we reported such invocations as blocking. Actually, putting sources at a network path may cause hangs. However, it rarely happens in practice; users usually keep a working copy on local disks instead. We manually re-classify these reported hang invocations as false alarms. All these false alarms were easily identified once we realized the semantics of the file path.

6.2 Editor: GIMP

GIMP is a popular graphics editor available on many platforms. It contains a main program and many plug-ins that enhance its functionalities. It is built based on the portable GTK+ toolkit and relies on a number of external libraries. We have analyzed GIMP and parts of GTK+ that provide system-related utilities, ignoring rendering-related external libraries that do not count. Unlike TortoiseSVN, GIMP's own logic is the majority, containing more than 800K lines of code; the complete application has about 1 million lines of code.

Since GIMP is a GUI application, we again use Pattern 2.1 as the responsive pattern, with seven additional rules as expert knowledge to capture callbacks to UI processing methods from GTK+. Because there are almost no networking operations in a desktop editor such as GIMP, we did not expect there would be hang bugs. Surprisingly, we have found 34 hang invocations at 10 hang gates. Top hang gates are summarized in Table 5.

GTK+ API function g_file_test_utf8 holds the first place, which further calls to Win32 API function GetFileAttributesW or POSIX API function stat. As in Pattern 2.3, it causes hangs in UI threads when a given file path is may-remote. In fact, this is a common problem for desktop editors: to input a network file path would cause hangs. Other two hang cases FcFontList and FcConfigAppFontAddDir may cause hangs when scanning directories and loading fonts, captured by the POSIX API function readdir that is used to read all entries in a directory in a loop. We consider an invocation to readdir as blocking, because its running time highly depends on the numbers of entries in a directory in outer environments, which is non-deterministic. This is why it takes rather a long time to start GIMP if a number of fonts are installed, which causes the splash window unresponsive.

hit	gate	bottom	⊤/⊥
28	g_file_test_utf8	GetFileAttributesW	4/1
1	FcFontList	readdir	3/8
1	FcConfigAppFontAddDir	readdir	2/8

Table 5: Top GIMP hang gates.

hit	gate	bottom	⊤/⊥
6	getaddrinfo	getaddrinfo	4/0
5	readdir	readdir	3/0
5	ldap_simple_bind_s	gethostbyname	2/9

Table 6: Top lighttpd hang gates.

There are 35 false alarms. Similar to those in TortoiseSVN, they are also mostly caused by invocations to GetFileAttributesW. Consider the following code snippet in GTK+.

```
g_home_dir = g_strdup (g_getenv ("HOME"));
if (!(g_path_is_absolute(g_home_dir) &&
  g_file_test(g_home_dir, ...)))
```

Since g_home_dir is read from an outer environment variable HOME, it is may-remote. Later it is passed into function g_file_test that subsequently calls GetFileAttributesW. It will hang if the environment sets HOME as a network path, which may happen in an intranet. However, as this is of low probability in practice, we classify them as false alarms. Again, once we inspected the code, it was easy to remove such false alarms.

6.3 Web Server: lighttpd

The lighttpd web server is fast and flexible to serve millions of pages per day. It is widely deployed to power many Web 2.0 sites, including MySpace.com and YouTube.com. To achieve high scalability and performance, it adopts a single-process, event-driven (SPED) architecture. Its execution requires neither context switching nor synchronization. However, a SPED web server has a major disadvantage: if the underlying operating systems or dependent libraries do not provide support for asynchronous operations, the synchronous substitutes will block the process and lead to loss of performance (Pai et al. 1999). In this experiment, we investigate such synchronous operations, i.e. blocking invocations used in lighttpd.

Like many other web servers, lighttpd is composed of dozens of module plug-ins that can be dynamically loaded. The complete application has more than 300K lines of code, including client libraries of OpenLDAP and MySQL.

Each module provides an entry function to register event handlers. In a SPED web server, all invocations there can be considered as responsive. Thus, we mark invocations in all handlers that are registered in the entry functions as responsive, since there are no explicit calls from the main program to plug-ins. We have found 24 hang invocations at 10 hang gates. Top hang gates are summarized in Table 6.

Among the hang gates, getaddrinfo is a POSIX API function that synchronously resolves a DNS hostname and IP addresses, so that it would cause hangs. The readdir function is

hit	pattern	brief description
43	GetFileAttributesW	Pattern 2.3
41	connect	Pattern 2.2
6	getaddrinfo	POSIX API function

Table 7: Statistics of top effective hang patterns.

similar to the cases in GIMP. Another source of hang bugs is blocking invocations to functions in external libraries, such as ldap_simple_bind_s in LDAP.

There are nine false alarms, due to false blocking connect invocations. While a bloated set of socket API functions on Windows (e.g. ioctlsocket, WSAAsyncSelect) will set a socket into non-blocking mode automatically, connect on a socket after these functions would not block. We conservatively consider all connect invocations as blocking, and thus we may report false alarms on connect. A better version would perform flow analysis to refine the blocking condition of connect and prune such invocations.

6.4 Experience

We have mentioned in Section 4.6 that we employ a hybrid approach that applies context-sensitive and context-insensitive analysis to an application's code and external libraries, respectively. Context-sensitive analysis is precise; and we did not choose the hybrid approach as an accident. Even with the efficient BDD-based Datalog implementation, our first attempt to apply context-sensitive analysis over the complete application has failed. In some cases, we ran out of memory on the 32 GB memory server. Though a BDD can represent relations compactly in general, its size is usually sensitive to BDD variable orders; some "bad" order would lead to an exponential size as well. However, finding an optimal order is an NP-complete problem (Bryant 1986).

As reported earlier, the hybrid approach does bring some false positives, but still at an acceptable rate. To improve the scalability for a precise and complete context-sensitive analysis, it requires extra efforts to find a better BDD variable order for BDD-based database queries, particularly for analyzing C/C++ code. This will be addressed in future work.

We inspect the most effective hang patterns, i.e. the top API functions that cause the most hangs in our experiments from Table 7, and find some interesting results.

- *Implicit blocking specifications of high-level library API.* Contemporary large software packages depend on a great number of external libraries. This is the case in *all* three applications we have examined. A subtle problem is that certain blocking functions such as connect can be hidden deep down the call paths *inside* external libraries. This makes the blocking specifications of high-level library API such as Subversion and MySQL *implicit*. These high-level APIs are entry points to external libraries. Unawareness of whether these APIs are blocking — in other words whether they are hang gates — puts the developer at the risk of bringing hangs into their applications. In our results, connect holds the one of the top killer places of hang causes, but the call paths all traverse to the external libraries through the hang gates.
- *Testing the existence of a path.* It is generally a gentle way to test whether a directory or a file exists before taking further operations. However, the corresponding API functions usually do not provide a timeout parameter for developers to specify a maximum wait span. Thus, calls to these functions may block in kernel and then timeout at rather a large value, often defined by drivers. This is an issue of legacy platform assumption from the days when computers were not connected and all storage I/O were local. Transparently supporting legacy storage I/O APIs that work over networked file path has its cost. Unawareness of such may-block APIs is another dominant source of hang. It is even worse to perform such testing while a user is interacting with an application. For instance in Eclipse JDT, when creating a project whose directory is remote mounted, it will cause hang every time a user presses a character when inputing the directory path.
- *Loading or saving files.* Many editors try to load and save files in UI threads. After all, this is the most convenient place to code in such logic. This is not an issue for small files, but will most certainly generate hangs when the format is complex to parse, or a file is large and stored on a slow device or a remote machine over networks. Besides, an editor often employs an extensible structure that enables plug-in modules to support more file formats. These modules may come from third parties, and expose the application to the similar level of responsiveness risks. Some recent editors such as the latest Microsoft Office do the job in a better way. They load files in a separate thread and pop up a dialog box indicating the status with a cancel button.

In our current implementation, most blocking patterns focus on OS-level API functions, such as the patterns listed in Table 1. If high-level API functions are similarly attached with blocking patterns, we will arrive at a more scalable solution since exploration can stop at these function calls. However, high-level library API functions may involve more complex blocking conditions. Consider the following Subversion API function.

```
svn_error_t *
svn_client_status(..., const char *path,
                  ..., svn_boolean_t update,
                  ..., svn_client_ctx_t *ctx,
                  ...)
```

It retrieves the status of a working copy file path; if update is true, it connects to the repository specified by ctx. It may block *if* the working copy path is remote, *or* update is true *and* ctx represents a remote repository. Such a blocking pattern can also be expressed in Datalog and solved by a logic database. However, it is not practical to enforce developers to specify the blocking condition for every API of their libraries. Thus, our future work will look into automatic specification synthesis for high-level API functions.

7. Dealing with Hangs

Once hangs are discovered, they must be fixed. Refactoring code is the most fundamental approach, and it calls for sound coding discipline as well as correct architecture. For legacy applications, refactoring may not always be possible, and we need to investigate techniques to safely cure hangs at runtime.

7.1 Code Refactoring

In general, it should be considered as a domain-specific *timing requirement* that a part of a program should be responsive. The blocking property of an operation, on the other hand, is a matter of *timing specification*. A hang bug occurs if the timing specification of an operation contradicts the timing requirement at a program point.

The general fix is to spawn a separate thread that takes the task, with a progress dialog box that also allows a user to cancel the operation. For simple patterns, this strategy is already adopted by developers. Some latest software such as Microsoft Office has adopted such a practice for some of its file operations.

It becomes extremely difficult for developers to understand the timing behavior of some invocations. This is where hang analysis tools such as the one we presented in this paper bring value. Such tools makes clear where the potential hangs can be, whether they are hang gates to external libraries or platform APIs that may hang depending on the calling context (e.g. `GetFileAttributesW` and `stat`). Once they are found, they should be generally fixed with the same strategy as outlined above.

However, there are a group of hang bugs that cannot be resolved by simply spawning separate threads. One category is problematic platform implementations. For example, the standard Windows print dialog thats pops up before printing may internally call the Win32 API function `EnumPrintProcessorDatatypesW` to connect to the default printer. The printer is usually locally attached or shared in an intranet. As such this API will return quickly. However, if the environment has changed, e.g. users reopen the print dialog in another workplace, or when there happens to be some problems with the printer or the intranet, this API will block, causing a hang. This problem is difficult to fix at the user level, since the user action that triggers the API *has* to be in the UI thread. One way to fix this problem is to let the print dialog include a cancel option. More fundamentally, however, it is the API that should be fixed.

The long debate about the design of high-performance, event-driven web servers (Pariag et al. 2007) is qualitatively similar. If the platform does not support an asynchronous API for the service that an incoming event handling relies on, *or* that the event handling must process this event in its entirety (i.e. cannot re-service it in a separate thread), the hang is unavoidable (e.g. the hang case of lighttpd).

7.2 Runtime Cure

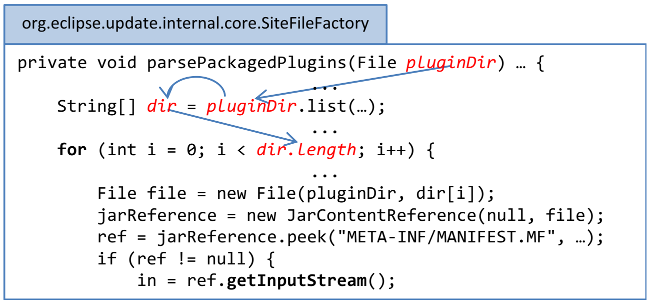
We have also developed a tool to cure hangs in legacy software at runtime without touching the source code. When an application hangs and a user presses a hot key, our tool will try to bring the application back to control.

The basic idea is to intercept functions that may block and to avoid falling into a coma. The tool requires no source code for binary instrumentation (Guo et al. 2008a). It takes API function prototypes as input and generates wrapper code automatically based on a given code template (Guo et al. 2008b). The wrapper code works as follows.

- For invocations to `wait` functions for a long timeout that most blocking functions may end up with, we break the timeout into shorter intervals (500 ms). If a user presses the cure hot key, the tool returns immediately with a timeout error code.
- For invocations to other functions that may block in kernel, i.e. they do not call `wait` functions in user space, we put them in a queue that are taken by threads in a separate pool. We periodically (500 ms) check the status, and it returns immediately with a preset error code if a user presses the cure hot key.

We applied our cure tool on several software systems and successfully recovered them from comas. A hang case occurs often in Microsoft Office when a user clicks on a faulty network path or an HTTP URL, such as in Word documents, PowerPoint slides or Outlook emails. Our tool is able to cancel all these blocking invocations in the UI threads and bring the applications back to respond. Furthermore, we can extract patterns in these cases for our analysis algorithm.

- The UI thread of Outlook blocks on a networking API function `WNetEnumResourceW` for a network path.
- The UI threads of Word and PowerPoint block on the Win32 API function `GetFileAttributesW` for a network path.
- The UI thread of PowerPoint blocks on a wait API function `WaitForMultipleObjectsEx` inside another API function `HlinkNavigate` for an HTTP URL.



```
org.eclipse.update.internal.core.SiteFileFactory

private void parsePackagedPlugins(File pluginDir) … {
                                 ...
    String[] dir = pluginDir.list(…);
                                 ...
    for (int i = 0; i < dir.length; i++) {
                                 ...
        File file = new File(pluginDir, dir[i]);
        jarReference = new JarContentReference(null, file);
        ref = jarReference.peek("META-INF/MANIFEST.MF", …);
        if (ref != null) {
            in = ref.getInputStream();
```

Figure 6: Example of long-running loops in Eclipse 3.3. The stop condition depends on `dir.Length` of the outer environment, and thus the loop may take a long time.

While useful, this tool is *not* a silver bullet solution. First, though modern operating systems such as Windows Vista may support cancelable I/O (Maffeo and Sliwowicz 2005), it would still be dangerous to cancel I/O operations in some cases, particularly if the application code is unaware of cancellation. In worst cases, it can result in inconsistent application states and even lead to crash. Besides, it cannot cancel operations that do not use synchronization objects, such as a wait by checking a flag variable in a loop rather than a wait for a signal. In these cases, code refactoring remains necessary to fix hang bugs.

8. Extensions

Our model has taken the granularity of invocation; this need not be. Furthermore, the concepts of responsive and blocking invocations can both be generalized. Below is an incomplete list of possible extensions.

Responsiveness is a design attribute, meaning certain code region/path should be completed quickly. Invocation is arguably the simplest instance when we talk about responsiveness. Another typical and useful instance is lock. If a lock is being held for a long period of time, lock contention can result in performance degradation. A simple rule is that there should be no long-running I/O in a lock region. Thus, lock region can be considered as a responsive pattern.

The blocking property, too, can refer to code regions more than just blocking invocations. The essence of blocking pattern is not that it will always be long-running, but that it some times does. The root of the problem is the *unpredictability* plus high *probability* of becoming long running. Hypothetically, a system written with many `connect` calls may not hang at all if it always connects to a loopback socket running on the same machine. Following this line of reasoning, there are more blocking patterns.

The blocking property may highly depend on *outer* environments, such as a may-remote file path we have discussed. Figure 6 illustrates a loop in Eclipse 3.3 when a user starts the update manager. The platform begins to examine each plug-in in succession. The loop depends on `dir.length`, which further depends on the number of files in an outer directory. If there are a large number of plug-ins installed in Eclipse, as is usually the case, the action causes a hang bug. Thus, the loop structure is potentially a blocking pattern, and it critically depends on the stopping conditions. Similarly, scanning a directory of files, computing a MD5 digest of a file by reading its data sequentially, all have highly structured patterns that we can generally lump into the blocking patterns.

Besides, the blocking property contextually depends on states that the blocking code is conditioned on. This brings up the subtle issue of `wait`. We consider it generally an unsound practice to wait in responsive invocations, since they introduce more dependencies among multiple components and increase the complexity of software. As a minimum, `wait` should have a timeout to prevent that crashing of the signaler from hanging the thread altogether.

The difficulty of `wait` in the case of hang analysis is that it depends not on any state at the local calling context, but on the execution of the signaler, which can be generalized into the "condition" of the blocking property of the `wait`. In its simplest form, if a wait is on termination of a separate thread, then we can mark the invocations in the separate thread as responsive. If, on the other hand, a wait is for a signal that is to be triggered in a separate thread, then we must trace signals passing in different threads and perform a flow analysis to compute happens-before relations between invocations.

The above are technically feasible to be performed semi-automatically. The most difficult part is when the wait is to be triggered by other processes. We have developed a dependency tracker that intercepts synchronization and scheduling functions in kernel to track dependencies between multiple threads and processes. For the hang case of clicking an HTTP URL in Powerpoint slides (see Section 7.2), we found that three processes were tangled in a complicated manner, which we still do not fully understand.

9. Related Work

We believe that our hang analysis framework is the first to establish a hang-focused model and apply it against large, real-world software. In the course of this research, we have borrowed ideas from a large body of existing work. We summarize them below according to several categories.

9.1 Responsiveness

This work is motivated by large number of hang cases that have plagued our daily research work. However, improving responsiveness has been a long standing research problems for many other critical software.

The pioneer work TIPME (Endo and Seltzer 2000) introduces a measurement infrastructure to detect response time of GUI systems such as Microsoft Windows and the X Window system. It focuses on interactive performance or responsiveness issues, and provides a basis for tracing abnormal cases and diagnosing responsiveness problems at runtime. We further generalize the model to event-driven servers and other time-critical applications, and develop several tools to deal with hang bugs.

One important category of performance-critical programs is drivers (Ball et al. 2006; Zhou et al. 2006; Anderson et al. 2007). In essence, our work shares a lot of similarities. However, drivers have a much smaller code base, and their interface is much more restricted. Scale and complexity are unique challenges that we must tackle, and to some extent we have tackled them via a scalable yet precise analysis framework .

Another type of applications that have received many attentions are web servers. Scalability and performance are both critical for web servers. Event-driven servers usually can achieve both goals. However, they also suffer from performance problems if there are blocking operations, as shown by one of the applications we have examined. There are ongoing discussions on high-performance web server architecture (Pai et al. 1999; Zeldovich et al. 2003; Elmeleegy et al. 2004; Wassermann and Su 2007), and we expect that their lessons are relevant to other software.

9.2 Correctness and Performance

A possible cause of responsiveness problems is due to correctness bugs that prevent a program from making progress. A typical instance of such correctness bugs is deadlock (Engler and Ashcraft 2003; Williams et al. 2005), which usually occurs in multi-threading systems. More generally, it is a fundamental halting problem to prove that a program terminates or to find corresponding bugs (Cook et al. 2006, 2007) that prevent it from doing so. Hang bugs are not correctness bugs, and the techniques to fight them need to be driven by a different model. Our hang analysis tool is complementary to previous work in the perspective of tackling correctness problems.

There are also a rich set of existing work on performance debugging. They aim at understanding root causes, and many of them rely on techniques to extracting patterns. There are numerous performance analysis tools (Aguilera et al. 2003; Chen et al. 2004; Barham et al. 2004; Chanda et al. 2007; Liu et al. 2008) for event tracing and performance profiling. In addition, machine learning techniques (Wang et al. 2004; Yuan et al. 2006; Basu et al. 2007) can be applied on traces for pattern extraction. These work are not directly related to our study, but the pattern extraction techniques and potentially some of the patterns extracted can be used in our hang analysis work.

9.3 Static Analysis

Many existing static analysis tools catch bugs such as resource leak and invalid use based on system-specific rules or patterns (Engler et al. 2001; Hallem et al. 2002; Larus et al. 2004). Our analysis system focuses on the problem of hang, but can also leverage these tools for a more complete diagnosis.

Our framework relies on a few techniques at the next level. Inconsistency inference via property propagation is used to compute the condition of blocking invocations (e.g. may-remote). This is similar to taint analysis that finds system vulnerabilities (Shankar et al. 2001; Martin et al. 2005) such as SQL injection. Our may-null analysis is related to null-deference analysis (Hovemeyer and Pugh 2007; Dillig et al. 2007).

We use annotations to inject expert knowledge into the hang analysis system. Annotation languages such as SAL (Hackett et al. 2006) and Deputy (Zhou et al. 2006) are effective to specify specifications and find system bugs. We use SAL for building monitoring and analysis tools. Our patterns for finding bugs can contribute to the annotation languages to specify additional specifications for functions, e.g. blocking conditions.

10. Conclusion

Soft hang is a widespread defect that plagues many software systems. This bug type is not correctness related, but harms user experience in desktop applications as well as degrades performance of server software. This paper presents a simple and effective model that defines the hang issue: a hang occurs when time-critical operations invoke blocking calls. We have built several tools and evaluated against several large, real-world applications. The results show that our static analysis tool is effective to uncover hang bugs with an acceptable false alarm rate. To the best of our knowledge, this is the first work that addresses the issue of hang at this scale.

Our study also yields a number of insights of the causes of hang. One of the dominant causes is that the legacy storage API hides the fact that the underlying storage may be remotely mounted. While backward compatibility is always important, the cost is creating more opportunities to obscure the fact that invocations can be blocking. Compounding the issue further, modern software typically reuses existing external libraries. Calls into library interfaces that may hide blocking invocations reside many layers below within the libraries. A lack of clear knowledge about the timing specifications of these library APIs is another source of hang bugs. We have discussed various code refactoring principles and techniques, as well as automatic and transparent runtime fix for legacy software.

We are actively pursuing this line of research further. In particular, we are investigating more efficient and scalable analysis to compute hang bugs more precisely, and pattern extraction techniques to automatically synthesize conditional blocking patterns. We are also continuing to understand more complex interactions among processes that result in responsiveness issues.

Acknowledgments

We would like to thank our shepherd, Terence Kelly, and the anonymous reviewers for their insightful comments. Thanks to John Whaley for useful discussions about bddbddb. We thank Frans Kaashoek, Wei Lin, Linchun Sun, Jian Tang, Ming Wu, Yuan Yu, Lintao Zhang and Lidong Zhou for valuable feedback. We are grateful to Ben Canning, Ben Ross, Vlad Sudzilouski, Igor Zaika, and the Microsoft Office team for their support. Xi Wang and Xiaoge Wang are supported in part by National High-Tech Research and Development Plan of China under Grant No. 2006AA01Z198 and by Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology.

References

Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

Z. Anderson, E. Brewer, J. Condit, R. Ennals, D. Gay, M. Harren, G. C. Necula, and F. Zhou. Beyond bug-finding: Sound program analysis for Linux. In *HotOS*, 2007.

T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.

P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

S. Basu, J. Dunagan, and G. Smith. Why did my PC suddenly slow down? In *SysML*, 2007.

R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

A. Chanda, A. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *EuroSys*, 2007.

M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, 2007.

M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, 2007.

K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *USENIX Annual Technical Conference*, 2004.

Y. Endo and M. Seltzer. Improving interactive performance using TIPME. In *SIGMETRICS*, 2000.

D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

Z. Guo, X. Wang, X. Liu, W. Lin, and Z. Zhang. BOX: Icing the APIs. Technical Report MSR-TR-2008-03, Microsoft, 2008a.

Z. Guo, X. Wang, X. Liu, W. Lin, and Z. Zhang. Towards pragmatic library-based replay. Technical Report MSR-TR-2008-02, Microsoft, 2008b.

B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *ICSE*, 2007.

M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS*, 2005.

J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21 (3):92–100, 2004.

X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *NSDI*, 2008.

V. B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS*, 2005.

G. Maffeo and P. Sliwowicz. Win32 I/O cancellation support in Windows Vista. `http://msdn2.microsoft.com/en-us/library/aa480216.aspx`, 2005.

M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

V. S. Pai, P. Druschely, and W. Zwaenepoely. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, 1999.

D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. In *EuroSys*, 2007.

T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic Publishers, 1994.

U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format-string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, 2007.

J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, 2005.

Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *POPL*, 2005.

C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.

N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Maziéresy, and M. F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*, 2003.

F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.