WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications

(Extra Material)

Abstract

This document contains the extra material of SIGMOD'24 submission #347. The main content is the formal proof of the correctness of WeBridge.

1 Correctness Proof of WeBridge

In this section, we show the correctness of WeBridge. The correctness condition of WeBridge states that, for any execution possible with WeBridge, there is always a equivalent execution possible in the original application. We first discuss the correctness of sequential requests, which is the Theorem 1.2 in this section.

1.1 Sequential Requests

We will introduce some preliminary concepts and notations. The program states of the original application API is denoted as $\langle D, S \rangle$, where D represents the database state, S represents the application's heap and local variable state. WeBridge extends the program states of the original application with a query result buffer B. Thus, the program states become $\langle D, S, B \rangle$. Additionally, let C_{sp} be the stored procedures WeBridge generated for the original application. Given the definition of program states, we now describe how the application is evaluated upon each user request. Let I be the set of all possible request inputs of an API. The input contains the request parameters for the API, and a system environment (which determine the return values of native method calls). For each input $i \in I$, suppose the initial program state of the API is $\langle D, S \rangle$, the final state of the API is $\langle D', S' \rangle$, then evaluating the API with input i is denoted with: $\langle D, S \rangle \xrightarrow{i} \langle D', S' \rangle$. Similarly, evaluating the API transformed by WeBridge is denoted with $\langle D, S, B \rangle \xrightarrow{i} \langle D', S', B' \rangle$. During the above evaluation, the program will execute along a *path* (Section 4 in the paper) P, which is a finite sequence of n boolean branch decisions denoted as $P = [b_1, b_2, ..., b_n]$. If $b_i = 1$, it signifies the i-th branch decision took the '*then*' branch; otherwise the '*else*' branch was taken. Let Π be the set of all possible paths and Φ be the set of paths optimized by WeBridge, where $\Phi \subset \Pi$. A hot path is a path p where $p \in \Phi$, a cold path p_c is a path where $p_c \in \Pi \wedge p_c \notin \Phi$. Additionally, the execution of each path of the program will lead to a sequence of SQL statements issued to the database. Each SQL statements consists of a *template* string and a list of *parameters*. We use the following notation: $\langle D, S \rangle \overset{i}{\Rightarrow} Q$ to represent that evaluating the original application API with initial state $\langle D, S \rangle$ on input i produces the sequence of SQL statements Q. We use $Q[i]$ to denote the i-th SQL statement in sequence Q, where $i \in \{1, \ldots, |Q|\}$. Additionally, we use $head(Q)$ to denote the first element of sequence Q, and $concat(Q_1, Q_2)$ to denote the sequential concatenation of two sequences Q_1 and Q_2. Similarly, for the application API transformed by WeBridge, we have $\langle D, S, B \rangle \overset{i}{\Rightarrow} Q'$, where Q' is the sequence of executed SQL statements.

We next define and proof the following lemma based on the above definitions.

Lemma 1.1. *Given an API program C and initial states D_0, S_0 and B_0, let I be the set of all possible inputs for C. $\forall i \in I$, If $\langle D_0, S_0 \rangle \overset{i}{\Rightarrow} Q$ under the original* *application,* $\langle D_0, S_0, B_0 \rangle \xRightarrow{i} Q'$ *under the application transformed by WeBridge, then* $Q = Q'$*.*

Proof. The premise contains

$$\langle D_0, S_0 \rangle \xRightarrow{i} Q \tag{1}$$

and

$$\langle D_0, S_0, B_0 \rangle \xRightarrow{i} Q' \tag{2}$$

The conclusion is

$$Q = Q' \tag{3}$$

The conclusion is proved by classifying the type of the path taken by C into two cases.

- A Hot path p. By definition of the hot path, we have

  $$p \in \Phi \tag{4}$$

  indicating that C is taking a path that have already been optimized by WeBridge. If p is optimized by WeBridge, then all the database accesses along the path are replace with calls to C_{sp}, which is the stored procedures generated by WeBridge for p. Let Q_{sp} be the sequence of SQL statements of C_{sp}, in which the SQL statement template string and parameters are extracted by concolic execution in in Algorithm 1. Since the concolic execution in Algorithm 1 is done by a deterministic reply of the hot path p, which implies that the collected concrete SQL templates in Q_{sp} must be the same with Q. Along with (1), we have:

  $$|Q| = |Q_{sp}| \tag{5}$$

  and

  $$\forall k \in \{1, \ldots, |Q|\}.\ Q_{sp}[k].template = Q[k].template \tag{6}$$

  For the query parameters, since WeBridge assumes the absence of global application states, the value of parameters are only determined by the external input states (Section 4 in the paper). All these states have been symbolized and their related computations are tracked in symbolic form via Algorithm 1. These symbolic computations are then transformed into equivalent stored procedure code in C_{sp} by Algorithm 4, by transformation rules that WeBridge assume to be semantic preserving. Consequently, any computations that might change the value of a parameter have been transformed into equivalent symbolic computations in the stored procedure. These computations are the dynamic backward slices [1, 2] of the query parameters. Therefore, given the same set of concrete input states to the original application and stored procedure C_{sp}, the parameter values computed from these input states must be the same:

$$\forall k \in \{1, \ldots, |Q|\}.\ Q_{sp}[k].parameters = Q[k].parameters \tag{7}$$

By the definition of a SQL statement, (5), (6) and (7), we have:

$$Q_{sp} = Q \tag{8}$$

Since p is a hot path, the path conditions for all SQL statements in Q_{sp} should evaluate to `true`, which means that all SQL statements in Q_{sp} will execute. Along with (2), we have:

$$Q_{sp} = Q' \tag{9}$$

By (8) and (9), the conclusion (3) is proved in this case.

- A Cold path p_c. By definition of cold path, we can further divide the type of cold path into two cases.

  - $\forall p' \in \Phi.\ head(p') \neq head(p_c)$. In this case, the cold path p_c diverges on the first branch decision of hot paths. Let $b = head(p_c)$ for the cold path, then the first branch decision for all hot paths must be $\neg b$:

    $$\forall p' \in \Phi. head(p_c) = \neg head(p') \tag{10}$$

    By (10), we divide Q_{sp} into two sub-sequences of SQL statements Q_{sp_1} and Q_{sp_2}, where:

    $$Q_{sp} = concat(Q_{sp_1}, Q_{sp_2}) \tag{11}$$

    And the path conditions of all SQL statements in Q_{sp_1} evaluates to `true`, which represent the queries that issued before the first branch condition is made; the path conditions of all SQL statements in Q_{sp_2} evaluates to `false`. According to the number of SQL statements in Q_{sp_1}, we can divide the proof into two cases:

    * $|Q_{sp_1}| = 0$. In this case, no SQL statement will be issued by C_{sp}, and the application trivially fallbacks to normal execution to issue all SQL statements in interactive mode. This indicates that $Q = Q'$, and the conclusion (3) is proved in this case.
    * $|Q_{sp_1}| > 0$. In this case, the path conditions for Q_{sp_1} in C_{sp} will all be `true`, indicating that these SQL statements will execute unconditionally for both hot paths and cold paths. Thus, with (8) and (9) we have:

      $$\forall i \in \{1, ..., |Q_{sp_1}|\}.\ Q_{sp_1}[i] = Q[i] \tag{12}$$

additionally, since SQL statements in Q_{sp_1} are all executed by stored procedure C_{sp}, and by (2), we have:

$$\forall i \in \{1, ..., |Q_{sp_1}|\}.\ Q_{sp_1}[i] = Q'[i] \tag{13}$$

For SQL statements in Q_{sp_2}, by (10) their path conditions will evaluate to `false`. This indicates that no SQL statement in C_{sp_2} will execute. Thus, the application fallbacks to normal execution to issue the following SQL statements in interactive mode. Then, we have:

$$\forall i \in \{|Q_{sp_1}| + 1, |Q|\}.\ Q[i] = Q'[i] \tag{14}$$

By (12), (13) and (14), the conclusion (3) is proved in this case.

– $\forall p' \in \Phi.\ head(p') = head(p_c)$. In this case, the cold path p_c and hot path p' "share" a common prefix of branch decisions. The hot path p' that has the longest prefix of branch decisions with p_c is:

$$\begin{aligned} \exists p' \in \Phi, n \in \{2, ..., |p'|\}, \forall j \in \{1, ..., n-1\}&. \\ p'[j] = p_c[j] \wedge p'[n] \neq p_c[n] \wedge& \\ (\nexists p'' \in \Phi . p''[n] = p_c[n])& \end{aligned} \tag{15}$$

Where $n-1$ in (15) is the length of the longest prefix branch decisions for p_c and p'. Additionally, any SQL statement in Q_{sp} with the n-th branch decision encoded in path conditions will not get executed, as $p'[n] \neq p_c[n]$. Let Q_{bf} be the sequence of SQL statements that do not encode the n-th branch decision in their path conditions, and let Q_{af} be the sequence of SQL statements that encode the n-th branch decision in their path conditions, we have:

$$Q_{sp} = concat(Q_{bf}, Q_{af}) \tag{16}$$

Then, by (15) and (16), we know that only the SQL statements in Q_{sp_1} will execute along path p_c. Then we know that:

$$\forall i \in \{1, \ldots, |Q_{bf}|\}.\ Q_{bf}[i] = Q[i] \tag{17}$$

and since SQL statements in Q_{bf} are executed by C_{sp}, and by (2), we have:

$$\forall i \in \{1, ..., |Q_{bf}|\}.\ Q_{bf}[i] = Q'[i] \tag{18}$$

For SQL statements in Q_{af}, their path conditions must include the n-th branch decision, which evaluates to `false` in the cold path p_c. This indicates that no SQL statement in C_{sp} will execute after the n-th branch decision is made. Thus, the application fallbacks to normal execution to issue the following SQL statements in interactive mode. Then, we have:

$$\forall i \in \{|Q_{bf}| + 1, |Q|\}.\ Q[i] = Q'[i] \tag{19}$$

By (17), (18) and (19), the conclusion (3) is proved in this case.

Then we have proved the conclusion (3), and Lemma 1.1 is proved.

□

Next, we prove the Theorem 1.2.

Theorem 1.2. *Given an API program C and initial states D_0, S_0 and B_0, let I be the set of all possible inputs for C. $\forall i \in I$, if $\langle D_0, S_0\rangle \xrightarrow{i} \langle D_c, S_c\rangle$ under the original application, $\langle D_0, S_0, B_0\rangle \xrightarrow{i} \langle D_{c'}, S_{c'}, B_{c'}\rangle$ under the application transformed by WeBridge, then $D_c = D_{c'} \wedge S_c = S_{c'}$.*

Proof. The premise contains

$$\langle D_0, S_0\rangle \xrightarrow{i} \langle D_c, S_c\rangle \tag{20}$$

under the original application, and

$$\langle D_0, S_0, B_0\rangle \xrightarrow{i} \langle D_{c'}, S_{c'}, B_{c'}\rangle \tag{21}$$

under the application transformed by WeBridge.

The conclusion includes

$$D_c = D_{c'} \tag{22}$$

and

$$S_c = S_{c'} \tag{23}$$

We first proof (22). By Lemma 1.1, (20) and (21), we know that the SQL statements issued by the original application and the application transformed by WeBridge are the same. Since the initial database state D_0 could only updated by the SQL statements, the database state after executing the same sequence of SQL statements should be the same. Thus, we have $D_c = D_{c'}$, and conclusion (22) is proved.

We next proof (23). Because WeBridge assumes the absence of global states in the application, the application state could only be affected by external input states. Thus, given the same input i for the original application and the application transformed by WeBridge, it is sufficient to proof (23) by showing the query execution results of each SQL statement in the original application and the application transformed by WeBridge are the same. We proof (23) by classifying the type of the path taken by C into two cases.

- A Hot path. In this case, all SQL statements are issued by the stored procedure, and the query execution results are first stored in the stored procedure buffer, which is initially B_0, then returned to the application upon following query invocations in sequential order. As the initial database states for the original application and transformed application are both

D_0, by Lemma 1.1, we can conclude that the query execution result of each SQL statement in the original application and the execution results stored in the buffer of the transformed application should be all the same. Consequently, the final execution result retrieved by the transformed application should be the same with the originally application. Thus, (23) is proved in this case.

- A Cold path. We can further divide the cold path into two cases by the number of SQL statements that executed by stored procedure.
  - No SQL statement is executed in the stored procedure. In this case, there will be no query execution result stored in the buffer. At the same time, all possible SQL statements will be issued interactively, which is exactly the same with the original application without a buffer. Thus, both applications should observe the same query execution results, and (23) is proved in this case.
  - At lease one SQL statement is executed in the stored procedure. In this case, we can divide the sequence of executed SQL statements Q_{sp} into two sub-sequences of SQL statements Q_{sp_1} and Q_{sp_2}, where:

$$Q_{sp} = concat(Q_{sp_1}, Q_{sp_2}) \tag{24}$$

    In (24), Q_{sp_1} is the sequence of SQL statements executed in the stored procedure, and Q_{sp_2} is the sequence of SQL statements executed interactively. Query execution results of Q_{sp_1} are processed just the same as how the results are processed in a hot path, except for the hot path will "terminate" after the last SQL statement of Q_{sp_1} is executed. The equivalence proof of query results for SQL statements in Q_{sp_1} is then the same with the hot path we proved before. For Q_{sp_2}, no SQL statement will be executed in the stored procedure, and the equivalence proof of query results follows the same proof of the previous case. Consequently, (23) is proved in this case.

As a result, Theorem 1.2 is proved.

Additionally, the correctness of the speculative execution optimization in Section 7 of the paper can be shown as follows. First, the validation and recovery algorithm in Section 7 ensures that the SQL statements executed by WeBridge is exactly the same with the original application, which implies (3). Next, by Lemma 1.2 and (3), the (22) and (23) can be proved just as as shown before, indicating Theorem 1.2 holds. □

1.2 Concurrent Requests

Before delving into the scenario of concurrent requests, we will establish some notations and definitions. We represent the executions of concurrent APIs as an *execution of events*. We identify six types of events:

- A SQL statement's invocation to the database in the original application, denoted as e_{db}.
- An invocation of an external (native) call, denoted as e_{ext}.
- Local computation(s) performed by the application, denoted as e_{loc}.
- An invocation of a SQL statement by the stored procedure, denoted as e_{sp}.
- An invocation of a SQL statement in the application by WeBridge, answered with the results from the stored procedure buffer, denoted as e_{buf}.
- An empty event, denoted as $\varnothing$.

Each event records the values of the arguments provided to the event during execution, along with the return values of the event. We define an *execution* to be an finite sequence of events. We consider two events e_1 and e_2 *conflict*, if given the same input parameters to e_1 and e_2, executing e_1 before e_2 produces return results different from e_2 executes before e_1. We then formally define the equivalence of executions:

Definition 1 (Equivalence of executions)**.** *Given two executions E_1 and E_2, if:*

- *The sequences of SQL statements executed by the database from events of e_{db} and e_{sp} are the same in E_1 and E_2.*
- *The sequences of external calls from events of e_{ext} are the same in E_1 and E_2.*
- *The sequences of local computations performed by the application from events of e_{loc} are the same in E_1 and E_2.*
- *If e_1 and e_2 are two conflicting events in E_1, and e_1 proceeds e_2, then e_1 and e_2 must also exist in E_2, and e_1 proceeds e_2.*

then E_1 and E_2 are equivalent.

Proof. The correctness condition of WeBridge is that, for any execution possible with WeBridge, there is always an equivalent execution possible in the original application. For sequential API requests, the correctness condition mush hold due to Theorem 1.2. When there are concurrent requests, we need to additionally consider how one API's SQL statement sequence issued inside a stored procedure might be interfered by another API's conflicting events, including 1) SQL statements that have conflicting writes, and 2) external calls that have conflicting writes. We need to prove that, when such writes interleave with a stored procedure, the execution is equivalent to one under the original application with those writes interleaving corresponding source SQL statements in the same way. We proof by showing that for any execution produced by the application with WeBridge, we can construct an equivalent execution, which is a valid execution of the original application.

We first introduce how we construction such an execution and why the constructed execution is equivalent to the execution of WeBridge. The construction follows a rule-based manner. Specifically, for each event in an execution of WeBridge:

- For an invocation of a SQL statement by the stored procedure, construct an empty event $\varnothing$.
- For an invocation of a SQL statement in the application by WeBridge, answered with the results from the stored procedure buffer, construct an event of an invocation of a SQL statement to the database in the original application with the same SQL statement.
- For an event of local computation(s) performed by the application, An invocation of an external (native) call, and an empty event, construct the same event.

Given an execution E_1 from WeBridge, we construct E_2 by the above construction rules. For any pair of events e_1 and e_2 that are conflicting in E_1, we highlight the arguments of some of the event types, others are proved similarly. If e_1 is of type e_{sp}, and e_2 of type:

- e_{sp}: this means two SQL statements in the stored procedure have a conflict. Since WeBridge collects SQL statements in the execution order of SQL statement, e_1 must proceeds e_2.
- e_{ext}: in this case, an external call is conflict with a SQL statement executed in the stored procedure. According to the Algorithm.1 in the paper, WeBridge "splits" the stored procedure generation upon each external call, so there is no way e_{ext} executes before e_1 in E_2.
- e_{db}: in this case, a SQL statement outside of stored procedure is executed after a SQL statement executed inside stored procedure. This indicates that the application must fall backed to execute queries normally, due to a failed validation in the speculative execution. As the failure recovery must happens after stored procedure invocations, e_1 proceeds e_2.

Consequently, conflicting events are the same in E_1 and E_2. Additionally, it is apparent to see that the sequences of SQL statements executed by the database from events of e_{db} and e_{sp} are the same in E_1 and E_2, the sequences of external calls from events of e_{ext} are the same in E_1 and E_2, and the sequences of local computations performed by the application from events of e_{loc} are the same in E_1 and E_2, from the construction rules. As a result, E_1 and E_2 must be equivalent. We next show that the constructed execution must also be a valid execution of the original application. Because we have shown that the conflicting events are the same in E_1 and E_2, then proving concurrent requests is reduced to proving sequential requests, which follows the previous proof. $\square$

References

[1] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[2] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 319–329, 2003.