# Leveraging Change History to Understand Development Challenges in Condition Variable Synchronization

[A Case Study of the Apache HTTPD Server Project]

Johanna Goergen
Washington and Lee University
204 W Washington St
Lexington, VA
goergenj16@mail.wlu.edu

## ABSTRACT

Writing multi-threaded programs that are both efficient and preserve correctness/atomicity is a great challenge for which very few automatic tools exist. Using the code repository to the Apache HTTPD Server Project, a highly multithreaded open-source server, we hope to draw some conclusions about what challenges real-world developers experience most often as well as what measures are often taken to address these challenges.

This paper discusses the findings of this study, which was done both automatically (using scripts to parse the contents of the code repository) and manually (by reading and analyzing particularly interesting code changes). It hopes to reveal specifically the challenges encountered when implementing condition variable routines as a means of synchronization. A similar case study was performed by Lu et al [1] regarding critical section change across four open-source multi-threaded projects (including the Apache HTTPD Server Project) and was useful in organizing this condition variable routine study.

## 1. INTRODUCTION

### 1.1 Motivation

It is our hypothesis that by collecting extensive data on the synchronization changes made by developers in the process of creating multithreaded software, we will uncover patterns that will lend themselves to algorithms for automating/assisting multithreaded software development. Studying particularly the motivation behind changes to synchronization variable routines will provide information about how developers are managing to balance efficiency and correctness in multithreaded software. The Apache HTTPD Server Project is an ideal project for such a study because its first publically released version came out in April 1996 and since then it has been revised over 1500000 times by a large team of developers. The Apache Server Project consists of many modules which make use of multiple processes and multiple threads to process HTTP requests and provide HTML responses as quickly as possible, thus requiring a large amount of synchronization.

### 1.2 Contributions

- **Graphing and Analysis of All Changes Over Time:** For each basic type of synchronization change (addition, removal, move, and recontextualize) I created a graph that plots the number of changes against the months from start of development. These show patterns of development that tell us what type of changes most often persist throughout development vs. what changes are required more at the beginning or with new functionality. The data for these graphs was collected using scripts which acquired the diff content from SVN for each revision and considered synchronization-related changes within that diff content, categorizing them automatically.

- **Analysis of Random Sample of Changes:** After the script mentioned above harvested and categorized all the relevant changes from the code respository, I used a script to randomly sample 60 of the nontrivial changes found by the original script. I then inspected each of these changes individually, categorizing each change as motivated by either "Refactoring", "New Functionality", or "Correctness". Within the "Refactoring" category, I further determined the motivation for each change to fall into one of the following subcategories: "Performance", "Code Reusability/Abstraction", "Stylistic", "Removing Unused Functionality", and "Inexplicable" (certain changes seemed to be counterintuitive and were usually changed back by another developer later on in development). Within "New Functionality", the only type of change was "Synchronization Added with New Code", in which the synchronization code was added alongside new structures. The subcategories of the "Correctness" motivation are: "Under-Synchronization" in which new synchronization is added where needed, "Over-Synchronization" in which too much synchronization previously yielded incorrect results (note, if oversynchronization was previously just causing performance slowness, this change would fall under the "Refactoring: Performance" category instead), "New Qualifications for Syncrhonization" in which conditions to be met before calling a condition variable routine are changed, and "Race Condition" in which there had previously been a race condition due to improper synchronization.

- **Analysis of Changes per Condition Variable:** I created a script that counts the number of occurrences of each condition variable within the HEAD revision of the Apache trunk. Then I created a script that counts the number of changes to each condition variable over the course of development. Then I analyzed the relationship between these results.

## 2. METHODOLOGY

### 2.1 Collecting and Categorizing Synchronization Changes Automatically

The first step in performing the motivation analysis for the condition variable-related changes in the Apache repository was to create a Python script that automatically gathered all relevant changes and determined as much as possible about that change. Namely, without maual analysis, it was possible to write a script that categorized each change as one of the following:

- **ADD:** Some routine from either the POSIX condition variable routine library or the Apache Portable Runtime condition variable routine library was added without that routine being removed from anywhere else in the same change block within the same revision of a file. By "change block", we mean a piece of the diff content consisting of many added, removed, and unchanged lines of code in consecutive order and delineated by something resembling the following: `@@ -905,7 +905,6 @@`. This change block delimiter means that the lines of code following the delimiter in the diff content are lines 905-912 of the old version of the file and they are lines 905-911 of the new version of the file. The diff content following this delimiter makes up a change block, which we will use in the next few classifications as well.
*The Algorithm:* Given one change block for some file, if a line appears in an "add" line (marked by a "+"), and the condition variable routine within that line (including its parameters) does not appear in a "remove" line anywere else in that same change block, then the line is considered an ADD line.

- **REMOVE:** Some routine from either the POSIX condition variable routine library or the Apache Portable Runtime condition variable routine library was removed without that same routine being added anywhere else in the same change block within the same revision of a file.
*The Algorithm:* Given a change block for some file, if a line appears in a "remove" line (marked by a "-"), and the condition variable routine within that line (including its parameters) does not appear in an "add" line anywere else in that same change bloick, then the line is considered a REMOVE line.

- **MOVE:** The same routine from either the POSIX or the APR condition variable routine library was removed from one location and added back elsewhere in the same change block of the same source code file.
*The Algorithm:* Given a change block for some file, if the exact same line (disregarding whitespace characters) appears in a "remove" line and an "add" line, the line is considered a MOVE. Note that this requires the whole line to be the same, not just the actual condition variable routine within the line.

- **RECONTEXT:** A certain POSIX or APR condition variable routine was moved only within the same "replace section" as a result of changes to the code surrounding that function call, indicating that the conditions under which the function is called have changed but the call itself is still necessary in relatively that same place. I define "replace section" to be a series of lines in the diff content such that unchanged lines (marked by leading whitespace) are followed by removed lines (marked by a leading "-") which are followed by added lines (marked by a leading "+"). The following is an example of a replace section:

```
    apr_status_t ap_queue_push(fd_queue_t * queue, ...
                        conn_state_t * cs, apr_pool_t * p);
-   apr_status_t ap_queue_pop(fd_queue_t * queue, ...
-                       conn_state_t ** cs, apr_pool_t ** p);
+  apr_status_t ap_queue_push_timer(fd_queue_t *queue,
...
+  apr_status_t ap_queue_pop_something(fd_queue_t * queue, ...
+                      conn_state_t ** cs, apr_pool_t ** p,
+                      timer_event_t ** te);
    apr_status_t ap_queue_interrupt_all(fd_queue_t * queue);
```

*The Algorithm:* Given just one change block from a file (a smaller block of change, not the entire diff content), if one line of code is removed and another line is added where the condition variable routines are the same but the lines are not exactly the same, this is considered a recontextualization.

- **MODIF:** Some routine from either the POSIX condition variable routine library of the Apahe Portable Runtime condition variable routine library is left in the same location with only its condition variable parameter changed.
*The Algorithm:* Given a change block, if any condition variable routine line is removed and then a new one is added in the exact same place with a new condition variable as a parameter, this is considered a modification. Interestingly, the algorithm did not find any instances of modifications.

```
- if (pthread_cond_init(&queue -> not_empty, NULL)
     != 0){
```

appears in the same change block as

```
+ new_cond = pthread_cond_init(&queue-> not_empty,
     NULL);
```

then we would consider this a recontextualize since the code surrounding the condition variable routine has changed, along with the line itself, but the routine and its parameters remain unchanged.

The script resulted in very accurate results and allowed for easily interpreting the change data into graphical data as well as performing an organized manual analysis of the motivation behind the changes.

### 2.2 Manual Analysis

As stated in the "Contributions" section above, the goal of the manual analysis was to break down 60 condition variable-related changes into three categories and many subcategories. I will describe each of these categories and subcategories in greater detail below:

- **REFACTORING:**
  - **Performance:** A certain change was made to improve performance, but not to remedy any actual correctness issues
  - **Code Reusability/Abstraction:** Some functionality was placed into a more reusable structure or function, or abstraction was added
  - **Stylistic:** Code was moved, added, or removed for the sake of improving the overall readablity of the code or for the sake of improving consistency in function return values, for example
  - **Removing Unused Functionality:** A variable, structure, or function was removed after being found unnecessary
  - **Inexplicable:** Certain changes just did not make sense
- **NEW FUNCTIONALITY:**
  - **Synchronization Added With New Code:** A new module or structure or a number of new functions were added and this new functionality includes new condition variable routine synchronization
- **CORRECTNESS:**
  - **Under-Synchronization:** Code was experiencing buggy behavior due to a need for more synchronization and was changed to remedy this
  - **Over-Synchronization:** Code was experiencing buggy behavior due to excessive synchronization and was changed to rememdy this
  - **New Qualifications for Synchronization:** Possibly due to newly added functionality or restructuring elsewhere in the code, the qualifications for synchronization have changed and the context of the synchronization lines were changed to reflect this
  - **Race Condition:** A race condition was occurring involving a condition variable routine and was changed to remedy this

An overview of these manual analysis results can be found in tables 1, 2, and 3.

## 3. SYNCHRONIZATION CHANGES

### 3.1 Overall Classification Breakdown

As expressed in the graphs below and to the right, the trend of classifications shows a sharp peak in additions and removals of condition variable synchronization changes in the early stages of development while moves and recontextualizations persist at a steadier rate throughout the development process.

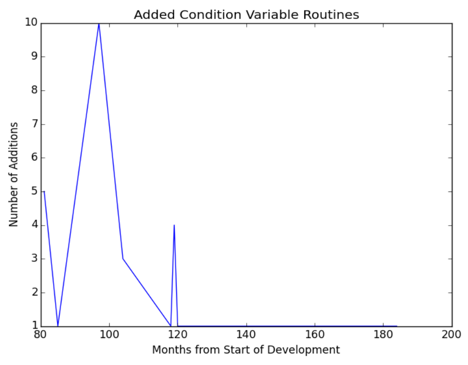

**Figure 1: Number of Condition Variable Synchronization Lines Added v. Months from Start of Development**

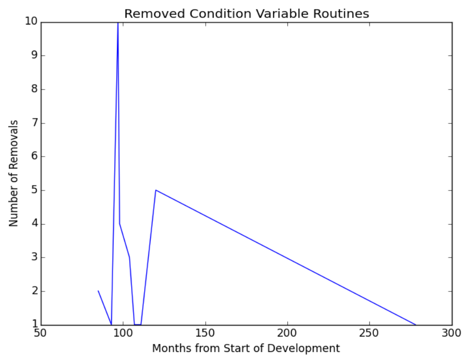

**Figure 2: Number of Condition Variable Synchronization Lines Removed v. Months from Start of Development**

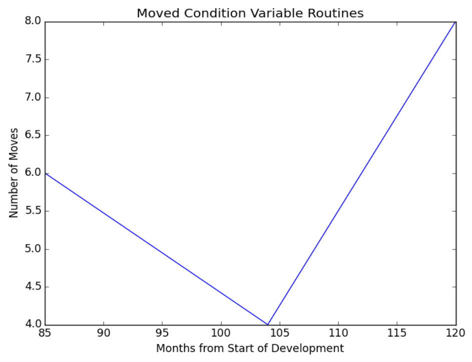

**Figure 3: Number of Condition Variable Synchronization Lines Moved v. Months from Start of Development**

**Table 1: Frequency of Refactoring Changes Per Subcategory in Random Sample**

| Classification | Number of Changes | Number of Revisions Containing Changes |
|---|---|---|
| Performance | 7 | 2 |
| Code Reusability/Abstraction | 3 | 2 |
| Stylistic | 16 | 7 |
| Removing Unused Functionality | 5 | 2 |
| Inexplicable/Trivial | 4 | 1 |

**Table 2: Frequency of New Functionality Changes Per Subcategory in Random Sample**

| Classification | Number of Changes | Number of Revisions Containing Changes |
|---|---|---|
| Synchronization Added with New Code | 6 | 3 |

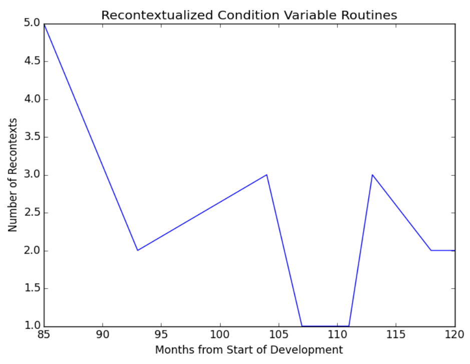

**Figure 4: Number of Condition Variable Synchronization Lines Recontextualized v. Months from Start of Development**

Evidently, the most persistent type of change throughout the development process is recontextualizing condition variable routines. Note that the Move graph also includes lines that were moved AND recontextualized, while the Recontextualize graph contains only purely recontextualized synchronization functions. This means if a condition variable synchronization routine was moved from one ”replace section” to another **and** the line surrounding the routine itself was changed in any way, this change is included in the Move category instead of the Recontextualization category.

By looking at the graphs, it is evident that lots of synchronization is added and removed in the initial stages but adds and removes do not persist throughout later stages of development except for the (rather rare) case of new functionality. On the other hand, condition variable routines are often recontextualized as a result of refactoring and remedying various subcategories of correctness problems.

## 3.2 Observations

### *3.2.1 Changes Per Condition Variable*

Apache's HTTPD Server Project relies heavily on reusable data structures like its ”worker stack” which is specifically for holding idle worker threads or its ”FD queue” which holds server sockets. Many of these structures defined in the different Apache modules contain condition variables as part of their state, automatically receiving these attributes upon initialization. For this reason, no global condition variables are ever initialized in the entire body of code. Therefore, although the study of changes made to different condition variables can tell us a good deal about common trends in condition variable change, it is not as conclusive as it would be if Apache depended on global variables.

Of the 54 condition variable routines (including condition variable initializations and terminations) used in the head revision of the project, 16 routines were on the condition variable ”`queue -> not_empty`”, with 7 signals and 3 waits. Here clearly ”`not_empty`” is an attribute of the FD queue structure. The next most common condition variable used in the head revision was ”`queue_info ->wait_for_idler`” with 6 routines with 4 signals and 2 waits, where `queue_info` is a structure containing various data attributes about the queue of server sockets.

Of the 86 condition-variable related changes made in the development history, 25 changes were made to the condition variable ”`queue->not_empty` ” (mentioned above) and 34 were made to the variable ”`queue->not_full`. This can be explained by the fact that the condition variable attribute ”`not_full`” was entirely removed (along with all routines involving it) from the FD queue structure and then later added back only to be removed again. This accounts for the high number of modifications involving the variable despite it not being a part of the head revision of the project. The next most modified condition variable is ”`queue_info -> wait_for_idler`” at 5 modifications. This is an interesting statistic since the ”`ap_queue_info`” structure was added to the project at revision 94824 –very late in the development process– but managed to require only one change on its condition variable after being added to the project.

From this, it is difficult to draw any concrete conclusions, however it does seem that the more times a condition variable is used in a project, the more often there will be changes required for routines pertaining to that condition variable.

### *3.2.2 Commonly Encountered Changes*

This section will refer to the findings of the manual analysis of 60 randomly sampled changes.

**Common Race Condition** All 6 of the race conditions found in this random sampling of changes were caused by some variation of calling signal, wait, or broadcast without

**Table 3: Frequency of Correctness Changes Per Subcategory in Random Sample**

| Classification | Number of Changes | Number of Revisions Containing Changes |
|---|---|---|
| Under-Synchronization | 6 | 3 |
| Over-Synchronization | 1 | 1 |
| New Qualifications for Synchronization | 1 | 1 |
| Race Condition | 6 | 3 |

holding the associated mutex. The following is one of the problematic code segments that caused a race condition to occur before it was changed:

```
while (stack->nelts) {
        worker = stack->stack[--stack->nelts];
        worker->csd = 0;
        apr_thread_mutex_lock(worker->mutex);
        apr_thread_mutex_unlock(worker->mutex);
        apr_thread_cond_signal(worker->cond);
    }
```

Of course, in certain situations this will lead to a race condition. For example, consider the thread $T_1$ encountering this code segment with `stack->nelts == True` (this simply means there are elements in the stack). Then by locking and unlocking the worker mutex, it wakes up a blocking thread $T_2$ and a context switch occurs, allowing $T_2$ to process and potentially change the value of `stack -> nelts`. Now the value that triggered the signal on the worker condition may no longer be true, causing a race condition when $T_1$ signals and wakes up $T_3$. By moving the signal into the critical section, this race condition will never happen.

In another of the six race condition cases, wait was called on the worker wakeup stack's condition variable after releasing the lock for the stack. Of course, this is actually known to be incorrect practice. All the other sampled race conditions were caused by signaling or broadcasting outside a critical section like the example above, which is less of a known transgression but still often considered to be bad practice due to possibilities like the one described above. Thus, checking for the mistake of condition variable routines outside critical sections would be an extremely feasible and helpful feature to implement within automated synchronization tool support.

**Adding Abstraction for Code Reusability** In a few cases within the "Refactoring" category, lines of condition variable synchronization were removed and placed into new functions for the purpose of creating abstraction that made the code more reusable and less redundant. This was due often to certain pieces of code being repeated, such as the code that awakens the next idle worker thread in the stack of idle worker threads. This was removed from multiple blocks of code and given its own function called `"worker_stack _awaken_next"`. Changes such as this were common and although they might be hard to fully automate, it would be easily possible to develop an algorithm that locates often-repeated pieces of synchronization code and recommends abstraction.

### *3.2.3 Trends in Adding New Functionality*

When new functionality was added to the project, it was occasionally necessary to insert new condition variable synchronization along with it. This happened three times in the random sampling of 60 changes. Upon further analysis of these 3 cases, the average number of new condition variable routines added alongside new functionality was 3. In only one of these three cases was a new condition variable initialized. Otherwise, due to the large use of abstraction and code reuse in the Apache project, the new signal/wait routines refered to pre-existing condition variables. This may be a practice unique to Apache since global condition variables are not used and specially created data structures are used for multiple parts of the server.

### *3.2.4 Move Vs. Recontextualization*

Unlike the mutex lock and unlock functions which delineate critical sections, when condition variable routines are moved, they are often changed too. It is rare the a signal/wait-/broadcast function simply stands alone when called. For example, it was most common to see these condition variable functions in a format like the following:

```
if ((rv = apr_thread_cond_signal(wakeup -> cond))
        != APR_SUCCESS){
        return rv;
}
```

Therefore, when condition variable routines are moved, they are often also recontextualized in terms of the condition that triggers them and the way their return values are stored. This makes the task of creating automatic algorithms to detect correctness issues especially challenging. However, the script which collected the move and recontext data was easily able to determine which condition variable routines were the same routine despite having been moved and recontextualized by simply considering the function call and its parameters and disregarding the surrounding code. This shows that luckily, there is often only one of each condition variable routine per condition variable in each change block, allowing for pretty easy detection of these combination move and recontextualization changes. This will be a good thing to evaluate in other applications besides Apache just to make sure this is a general practice and not unique to the Apache Server Project.

## 4. CONCLUSION

Although Apache is a relatively small project in terms of the amount of code it contains (compared to larger projects also studied by Professor Lu's lab, such as Mozilla [1]), the findings of this study present a few ideas that might be useful to incorporate into future multithreaded development tools. For example, it seems that Apache's heavy reliance on abstraction and highly reusable data structures makes it so that developers don't often have to manually implement condition variable routines and they almost never have to initialize new condition variables. Evidently, judging by the strikingly small total number of modifications made to the condition variable routines within this project throughout its almost 20 years of development, these practices did make successful synchronization easier in the long run despite seeming complex upon first glance. While lots of new functionality was added throughout the course of development, only three large functionality additions involved new condition variable routines and even these additions involved only an average of three new condition variable routines, indicating that this high level of abstraction and code reuse allows for code extensibility.

Hopefully the findings of this case study will help inform some decisions when creating multi-threaded code development tools in the future.

## 5. REFERENCES

[1] R. Gu, G. Jin, L. Song, and S. Lu. What change history tells us about thread synchronization.