Bitcoin Core – Enhancement Proposal

Date: April 9th, 2023

Prepared by:
Daniella Ruisendaal (Team Lead) – 17dar8@queensu.ca

Alina Padoun (Presenter) – 18ap1@queensu.ca

Adam Ciszek (Presenter) – 19ac80@queensu.ca

Aidan Wolfson – 18aaw1@queensu.ca

Camila Izquierdo – 18cni@queensu.ca

Tanner Big Canoe - 19tjb8@queensu.ca

Table of Contents

Abstract

Throughout this report, we propose an architectural enhancement to the current Bitcoin Core system. That being to reduce the functions that require the global lock and mutexes so that the parallelism of the system can be increased, allowing for greater concurrency. We propose how this implementation might look, demonstrate both the primary effects and risks this would have on the overall system, and show how such an implementation can be tested. Throughout this report there will be a detailed analysis on our implementation, along with visual diagrams to further demonstrate the proposed enhancement and differentiate it from the current system.

Introduction

In Assignment 1, a conceptual architecture for the Bitcoin Core system and its subsystems was proposed. The architecture was built around providing users with the chance to take part in the bitcoin network as a full node, allowing users to send and receive payments, verify and validate all transactions, and engage in mining. To support this, multiple subsystems work in coordination with one another including the blockchain, the P2P network, the bitcoin wallet, the transactions module, the mining subsystem, the payment processing subsystem, and the operating mode subsystem. All these subsystems fulfill the core functions of Bitcoin Core.

In Assignment 2, a concrete architecture was derived from the source code of the system and any differences between it and the proposed conceptual architecture were identified. While the concrete architecture implemented many of the conceptual subsystems such as the P2P network, it also included new subsystems and dependencies that were not accounted for in the conceptual architecture. Each of these new dependencies were between a previously discussed subsystem and a newly discovered one. While the concrete architecture did implement the P2P network, there were differences in subsystems such as the bitcoin wallet, which was found to have 2 main types: the deterministic wallet and the non-deterministic wallet. New dependencies included the multi-threaded nature of Bitcoin Core, which introduced new threads to the concrete architecture that serve important roles in fulfilling the system's functions. While the concrete architecture shows an overall well-made and efficient system, there were areas of growth identified.

The goal of this report is to propose an enhancement to the Bitcoin Core system and the changes required for the existing architecture to support it. An identified drawback of the Bitcoin Core architecture throughout the previous assignments has been the system's lack of parallelism for many critical paths and limited concurrency, which has negative impacts on the system's performance and response times. The enhancement we propose in this report is reducing the number of global locks in the Bitcoin Core system to increase parallelism and improve the concurrency, with an end goal of increasing performance and reducing bottlenecks.

Such an enhancement would impact many functions throughout the Bitcoin Core system that, in its current implementation and concrete architecture, are limited in their ability to run concurrently. Whenever a global lock is acquired by a function or subsystem, other functions are denied the ability to access and/or modify the same data simultaneously. By removing such locks, many more processes would be able to run concurrently, and the performance of the system would increase.

This report first discusses the architecture involved in the enhancement, including the current state of the system, the effects of the enhancement on the maintainability, evolvability, testability, and performance of the system, as well as the interactions of the enhancement between subsystems. An analysis of alternative options to implement this enhancement is then discussed, including a SAAM analysis, testing discussions, potential risks, and adjusted use case diagrams with and without the implemented enhancement.

Architecture

Current state of system

In its current state, Bitcoin Core uses a multitude of threads as well as locks that guard shared data structures and sync the multiple threads. Bitcoin Core's system, although multi-threaded in many aspects, is mostly single-threaded when it comes to many critical paths (Newbery & Towns, 2021). Certain functions are required to grab a global lock prior to starting work and must therefore wait for other threads to complete their work and release the lock. Such vital actions include, but are not limited to, running wallet tasks, completing transactions, and validating blocks. This limits the concurrency that can be achieved by the system (Newbery, 2017). One of the most important states is protected by a single mutex called cs_main, as defined in the file 'bitcoin/bitcoin/blob/master/src/kernel/cs_main.h' and employed in the file 'bitcoin/bitcoin/blob/master/src/validation.cpp'. cs_main is a recursive mutex that guards validation-specific data (such as reading and updating the chainstate), updating the transaction pool, and protecting additional elements mostly in net_processing (Newbery & Dong, 2021). Splitting up such a mutex can further serve to increase parallelism, and as such, the enhancement we propose is that of reducing such global locks to improve modularity and increase decoupling within the current system's state.

Effects of Enhancement

By reducing the number of functions that access the global lock, which is responsible for preventing multiple threads from accessing and modifying the same data at the same time, the number of processes which can run concurrently increases. The most important effect of this enhancement is the obvious increase in parallelism possible in the system by reducing which functions have access to this lock. Since many functions use the lock to grab exclusive access to data while they run, limiting these locks means more subsystems can be active concurrently, increasing response time and performance in the system. Additionally, removing the global lock feature would reduce the downtime of subsystems, which inherently reduces any performance time lost to subsystem start-up or wind-down costs. By allowing more processes to run concurrently, the loads of the system can be better spread across multiple subsystems and

processes. This could lead to increased job capacity and throughput for the system, again increasing performance and response times across the board.
This enhancement has notable effects on the maintainability, evolvability, testability, and performance of the Bitcoin Core system. Firstly, depending on how the enhancement is implemented, the maintainability of the system should remain the same as its current state, although it may be affected by the increased modularity. Limiting or removing the global locks of the system may increase the risk of issues, but the maintenance of those should remain relatively unchanged. Next, while the evolvability of features will increase due to performance increases from higher concurrency in the system, evolutions of the system will have to be monitored to ensure that corruption of unprotected and unlocked data is minimized and avoided. With more functions being added, the risk of data corruption and other issues will increase. Next, this enhancement increases testability, as the system will be able to handle higher workloads with a higher throughput, allowing for faster testing processes and responses. Finally, in terms of performance, limiting the number of global locks or removing them entirely increases the concurrency of the Bitcoin Core system, allowing for higher throughput of jobs and increasing response times and performance overall.

Interactions of enhancement

Many subsystems make use of the global lock in order to ensure minimal file/data corruption by simply preventing threads from accessing and/or modifying the same data simultaneously. By reducing the number of functions that access this global lock, we are essentially limiting the prevention of other processing running concurrently. As mentioned earlier, many functions make use of the global lock, therefore, many subsystems would be affected by a reduction in this. Many processes would be able to run concurrently, and subsystems would be able to interact more simultaneously. These subsystems include:

- Validation/Consensus Algorithm
    - The consensus algorithm gives the validation engine its protocol and is what controls and validates transactions and blocks. It uses the global lock to prevent inconsistencies and make sure that only a single validation state is being modified.
    - The *validation.cpp* file, along with its header file contain information on the validation engine which makes use of the global lock, as well as other files in the consensus directory which provide information to the validation engine. Reducing some of the functions which use the global lock can impact the concurrency.
- Transactions
    - To prevent conflicts such as race conditions, the global lock is used. The transactions module makes use of this so when there are multiple processes on the same transactions data, only one of these processes works at each time.

    - Within the transaction's module, there are many files that interact with other affected subsystems. Some of these include *attributes.h, txdb.cpp, arith_uint256.cpp,* and *policy.cpp.*
        - All these files implement and interact with other functions that implement global locks. It goes to show the extent of these interactions.
- Storage
    - This module is responsible of all the data, thus, reading and writing data simultaneously can cause issues and corruption if the data being accessed is the same. The global lock is implemented to prevent this.
    - The storage contains information from a variety of other subsytems, and all of this data is being interacted with. This forces the Storage module to implement global locks, and some files include *coins.cpp, blockencodings.cpp, chain.h, bitcoin-tx.cpp,* and other directories such as *leveldb and txdb*
- Peer Discovery
    - When connecting to other peer nodes, it is important that sending and receiving messages and information is done in coordination which is why global locks are used. The effect is much less then the validation and storage engine, but exchanging such information cannot be done simultaneously when writing it as well.
    - Primarily the *protocol* files and the *connection manager* directories are the ones responsible for implementing global locks as they extend to and receive information from other modules which then implement global locks.
- Mempool
    - The mempool is similar to the storage module, such that it contains information of unconfirmed/not validated transactions. The global lock is responsible to make sure these unvalidated blocks are not removed or added, as well as make sure they do not become corrupted.
    - Files such as *txmempool.cpp, mempool_limits.h*, and *mempool_entry.h* implement the global locks to manage the unvalidated information. They also implement and extend to the other modules, but mainly validation and transactions to further validate and store the blocks. Step by step, the functions responsible for validation and storage all implement global locks and would be affected by a reduction.

Overall, many files must implement the global lock or implement other files that implement the global lock in order to prevent corruption of certain files. However, most of these functions extend to the modules we've discussed above, which generally relate to validation and storage – this being where the information passes through and is stored. Below is a screenshot of our understand graph, and we have highlighted files that implement global locks. The graph demonstrates further how our implementation may affect other modules and the high-level architecture itself.

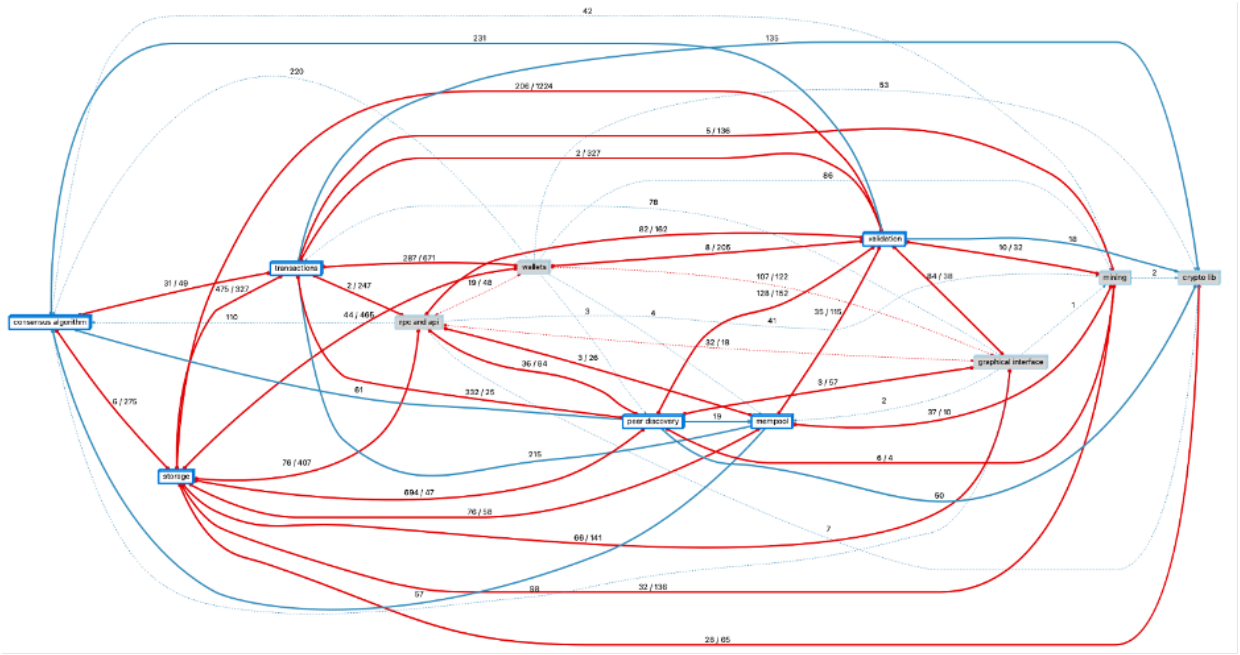

Figure 1: Dependency Diagram of Bitcoin Core as seen in Understand with files using global locks highlighted

Alternative Analysis

There are multiple ways that removing a feature can be implemented within the existing Bitcoin Core architecture. This section outlines two such possible implementations, which will be done below in the SAAM analysis.

Implementation One: Local Lock Feature is Distributed Across the Existing Module Pathways

In this implementation the global lock would become a local lock that exists within each of the modules. This simply removes the global aspect of the lock and changes it to only locking other modules that are dependent on the current module running. This means that module pathways can run in parallel, as long as they do not have dependencies. This implementation speeds up the time of the whole software system but also prioritizes the security of the previous version. This would be very time consuming to implement into the software system as not only are developers deleting the global lock, but also finding the pathways that exist and creating local locks.

Implementation Two: Remove Global Lock Feature

In this implementation there will be no global lock. This implementation will be extremely fast as there is no longer a global lock at all, so all subsystems and modules can run their processes in parallel. This implementation sacrifices the security of the validation algorithm, mempool and storage in exchange for performance. This solution is much easier to implement since the global lock is simply being removed from the software system and modules are allowed to run in parallel.

SAAM Analysis

The SAAM analysis was created to qualitatively assess a software architecture by evaluating a variety of use scenarios for it. A set of stakeholders and cases are investigated which analyze the core functionalities of the software architecture, or in this case a feature of a software architecture, to either assess, refine, or create the most optimal model.

Below are the major stakeholders in the proposed global lock function reduction to increase parallelism capabilities for both implementations:

- Bitcoin Core Developers
- Independent Developers
- Bitcoin Customers (Users)
- Investors

The following nonfunctional requirements (NFRs) were chosen as the critical factors for both implementations of the global lock reduction for the Bitcoin Core software system. Each NFR applies to different stakeholders depending on their roles in Bitcoin and the global lock.

Major Stakeholders	**Nonfunctional Requirements**
Bitcoin Core Team	Performance, Maintainability, Scalability
Independent Developers	Performance, Maintainability, Scalability
Bitcoin Users	Performance, Safety
Investors	Performance, Safety

Brief descriptions of each NFR are as follows:

- Performance: Faster speeds due to the increase in parallelism within the software system.
- Safety: Ensure safe movement of data throughout the software system as the global lock acts as a guard rail.
- Manageability: Easy to keep track of for developers, no ambiguity, manageable to update/maintain the feature.
- Scalability: Ensure the ability to improve and scale the feature for future modules and additions to the system.

After the NFRs have been realized, it is now important to compare the two implementations of the global lock reduction and take stock of the NFRs based on their application to different cases for each of the implementations. Below is a table consisting of evaluations for each NFR; Low, Medium, and High are the tags given depending on the quality of each NFR with respect to

either implementation 1 (Local Lock Feature is Distributed Across the Existing Module Pathways) or implementation 2 (Remove Global Lock Feature).

NFRs	**Implementation 1**	**Implementation 2**
Performance	Medium – With local locks the speed of the system increases, but not maximally.	High – This is the maximum increase in performance.
Safety	High – With local locks we still maintain maximum security from the previous version.	Low – This implementation sacrifices security as there is no longer a global lock to safeguard against issues.
Manageability	Medium – The only reason this does not achieve high is because of the number of pathways that will exist in creating a local lock. This will be difficult to manage.	Low – Although it does not take a lot to implement, managing a system with no guard rails will be very difficult.
Scalability	Medium – With every new iteration of the system there is a chance that the local locks will have to be updated. It is medium not low because splitting the global lock/mutexes can improve modularity which can improve scalability.	High – There are basically no changes that need to be made for future versions of the system.

SAAM Analysis Conclusion

After completing the SAAM analysis, the first implementation, while not maximizing performance and scalability, increased the previous system's performance and scalability without comprising other non-functional requirements, and in some cases increasing these as well.

Testing

To test the enhancements regarding the global lock in Bitcoin Core, there are two main tasks that can be completed. To accurately test the enhancement, a task must be chosen that, in the original code, experiences the downtime of a global lock and, after being enhanced, does not have the global lock anymore.

First, simple timing processes can be implemented in the Bitcoin Core code to measure the time that it takes to complete before and after removing the global lock. In C++, timing can be implemented using the chrono library (cppreference.com, 2022). The duration of the process can

be found by declaring a start time variable before running the process, then declaring a stop time variable once the process has finished and finding the difference between these two. Sample code sourced from GeeksforGeeks to implement a timer can be seen below:

```cpp
// C++ program to find out execution time of
// of functions
#include <algorithm>
#include <chrono>
#include <iostream>
#include<vector>
using namespace std;
using namespace std::chrono;

// For demonstration purpose, we will fill up
// a vector with random integers and then sort
// them using sort function. We fill record
// and print the time required by sort function
int main()
{

    vector<int> values(10000);

    // Generate Random values
    auto f = []() -> int { return rand() % 10000; };

    // Fill up the vector
    generate(values.begin(), values.end(), f);

    // Get starting timepoint
    auto start = high_resolution_clock::now();

    // Call the function, here sort()
    sort(values.begin(), values.end());

    // Get ending timepoint
    auto stop = high_resolution_clock::now();

    // Get duration. Substart timepoints to
    // get duration. To cast it to proper unit
    // use duration cast method
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "Time taken by function: "
         << duration.count() << " microseconds" << endl;

    return 0;
}
```

Figure 2: A code snippet sourced from GeeksforGeeks that implements a timer (GeeksforGeeks, 2022).

The next method of testing the performance of the enhancement will be load testing. Parallel programming is a useful practice to simultaneously run many processes that usually leads to shorter process times. However, if too many processes run at once, the device memory may reach capacity and slow down the processes rather than completing them faster. To find the load capacity, that is, the number of processes at which process times are optimal before increasing, load testing tools can be used. One such tool is Googletest, a C++ compatible framework that can be used to start a given number of processes under the new enhancements and measure performance times. This tool is also useful as it can produce reports (Google, 2023).

This enhancement upholds testability as testing tasks are easily to implement and give valuable results as to whether the enhancements will be beneficial to Bitcoin Core.

Potential Risks

Due to the dynamic and collaborative nature of Bitcoin Core's code, making changes to such an ingrained part of it such as this has the potential to cause many issues. There would need to be major changes to pieces of the code surrounding many of the effected functions in order to

integrate this enhancement. Doing so could lead quite quickly to the introduction of new bugs and errors that would also need to be solved and patched. It could also cause developers to miss a section of relevant code to update and cause the code to break and not run at all, or even worse, cause a runtime or logic error that might not be immediately obvious to those checking the code over.

Another potential risk with implementing this change is that doing so could potentially lead to file and data corruption with multiple threads potentially being able to access and modify the same data simultaneously. This is, of course, something that would be considered and accounted for when changing the code but should still be mentioned as it still poses a relevant risk.

Some of the modules that would need to be changed could cause their own issues as well. The validation/consensus algorithm has the potential to cause issues; their interaction and the lack of a global lock between them could cause inconsistencies between validation states, with the potential for multiple validation states to be modified at the same time. Changing the global lock for the transactions module could cause race conditions, with multiple processes trying to access the same transaction data at the same time. With global locks, this wasn't an issue because the lock permitted only one process to complete at a time, but with its removal, the code becomes vulnerable to the possibility of multiple processes claiming the same transaction.

Diagrams

There are two use cases for which sequence diagrams have been created.

Use Case 1

The first use case depicts a User A that would like to send a bitcoin balance to User B. Below is the sequence diagram as it stands.

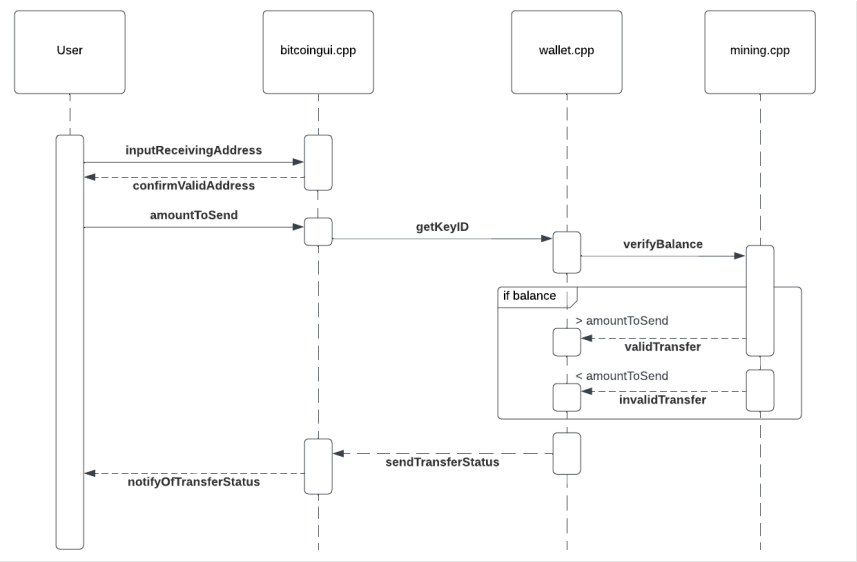

Figure 3: A sequence diagram of use case 1 with global locks.

If the global locks are removed, the process can be represented by introducing another User C, which User A would like to send bitcoin to right after sending to User B. The sequence of the process can be seen in the diagram below.

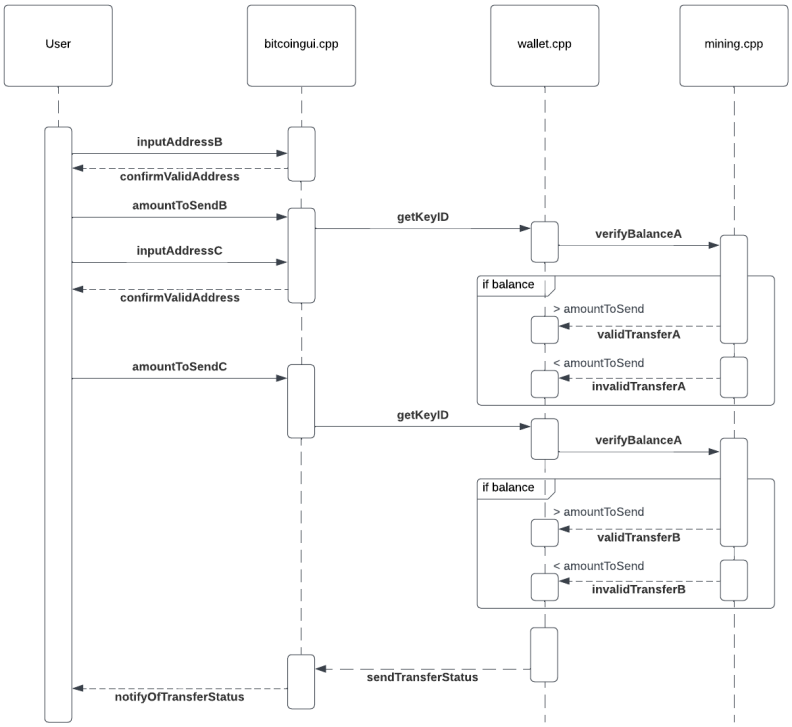

Figure 4: A sequence diagram of use case 1 without global locks.

This causes issues because the result transfer status to User B can be shadowed by the result of User C without global locks, which skews the current account balance of User A. In addition, if the account tries to overdraw the account, both transactions could be denied, even if one is within the account balance.

Use Case 2

The second use case concerns the activity of mining on Bitcoin Core. The user opens their console from their account details and begins or ends the mining session. To fully examine the changes that would be caused by introducing the enhancement in the mining process, two extra levels have been added to the sequence diagram. The diagram below shows the current use case. The extra levels show what happens once mining starts – as long as there are blocks to validate, the loop will continue to validate a block and change its validation state. It is important to note here that before moving onto the next block, the loop waits for a signal.

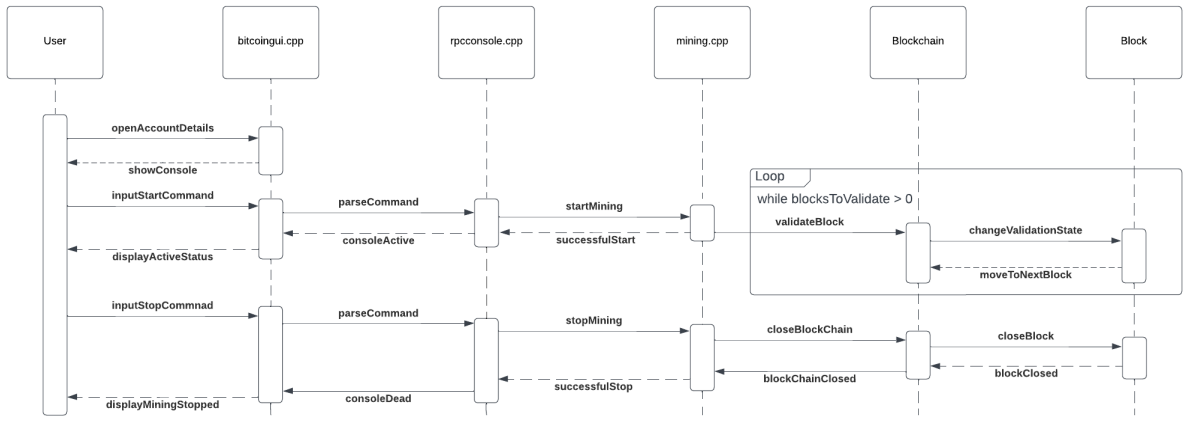

Figure 5: A sequence diagram of use case 2 with global locks.

If the global lock is removed, the state of the use case changes to the sequence diagram below. The signal that was initially present to signify that the mining could move to the next block is removed, and loop can now initiate many validations in parallel. This could cause errors in validation where the blocks are not receiving the correct state, or their state is being changed too many times.

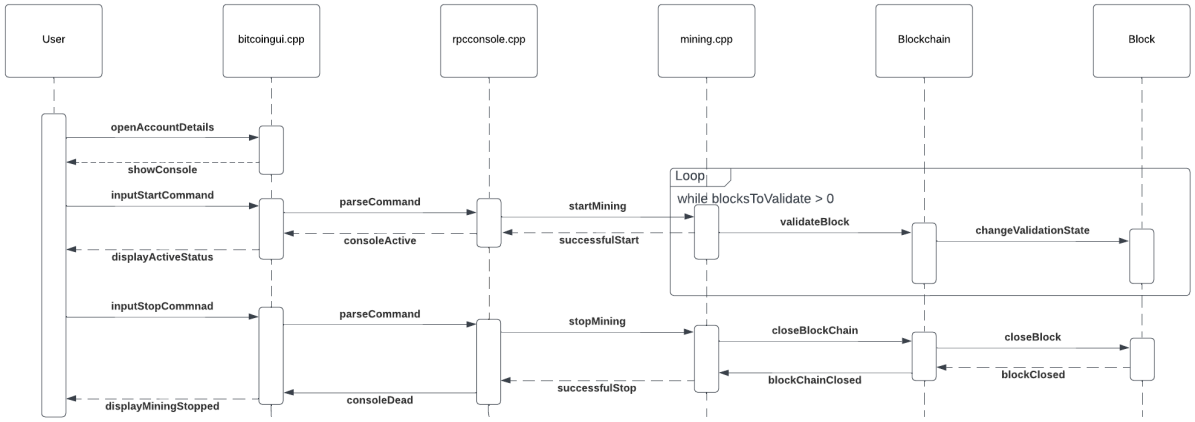

Figure 6: A sequence diagram of use case 2 without global locks.

Use Cases

Two of the three use cases that were identified and updated to reflect the concrete architecture will be compared to their forms using the potential removal of global locks. The remaining use case has not been included because its functionality relies on the user interface (UI) response, which is not significantly affected by global locks.

The first use case depicts User A that would like to send a bitcoin amount to User B. This requires that User B's receiving address be entered, checking if the amount to send is within the spending limits of User A, and a notice of the final transfer state from the server. This use case leverages the Transactions sub-system that verifies the keys of both users, specifically the User A which is sending the bitcoin balance. By removing the global lock in this use case, the

transactions would benefit from quicker times, however it is imperative that race conditions are considered and minimized. If this consideration is not made, a risk of multiple overlapping processes could occur. For example, if User A tries to send another amount to a new User C after User B, but does not have enough bitcoin, the transaction would fail. However, without global locks, these processes could run in parallel and try to overdraw the total bitcoin at once or in the wrong order, in which case both User B and User C are negatively affected.

The second use case revolves around the mining aspect of Bitcoin Core. A user can access their account details through the UI and navigate to the Console which will start or stop mining upon receiving the respective command. This use case leverages the validation sub-system in its operations, which in turn triggers a global lock to ensure that only one validation state in question is being changed. Although removing the global lock and introducing parallelism could speed up mining, which is notoriously time-consuming, the accuracy of the blockchain could be compromised. It is possible that the system could err and begin assigning the incorrect validation state to a block. In addition, mining is already quite computationally intensive so the additional memory needed to introduce parallelism may not be optimal in this use case.

Naming Conventions

NFR: Non-functional requirement (s)
UI: User Interface

Conclusion

This enhancement proposal for Bitcoin Core focused on improving the system's performance by increasing the speed at which it completes tasks as well as reducing bottlenecks. The enhancement proposed was to reduce the number of global locks that are used by the system. While the idea behind this is sound, by reducing the number of locks more processes can run at a time and therefore finish faster, there are certainly risks and impacts to the system that must be considered before deciding to implement this idea in the main code.

The increased parallelism that would occur with a reduction in global locks would greatly benefit the speed at which the code is able to execute. Processes such as the transactions module and the storage module would benefit due to the ability to run concurrently, allowing such modules to interact more simultaneously than before.

The risk of data corruption would increase with this enhancement due to more data being unprotected and unlocked, but this can certainly be monitored to avoid most if not all of this occurring. The risk of inconsistencies in validation states or race conditions in the transactions module must also be considered when deciding how best to implement this change.

Overall, the implementation of this enhancement would greatly benefit Bitcoin Core in regard to its performance, but it does of course come with risks and would need plenty of time and a careful eye to implement and to ensure that nothing was missed in the code that could cause errors when switching over to this new method of reduced global locks.

Lessons Learned

Amongst the valuable lessons learned throughout assignment 1 and 2, we uncovered potential issues and opportunities for improvement related to the concurrency in Bitcoin Core's system, which we analyzed in this final assignment. As it stands, although the system is multi-threaded, there is limited parallelism due to global locks and mutexes that force vital actions to be single-threaded. Global locks and mutexes such as cs_main, can be split up in order to improve modularity and increase concurrency capabilities. Such a change would impact many different subsystems, including the Validation/Consensus Algorithm, Transactions, Storage, Peer Discovery, and Mempool modules. The SAAM analysis identified key stakeholders and non-functional requirements, which helped in weighing between alternative implementations of either removing locks entirely or splitting global locks into local locks per module. Various testing methods and tools, such as chrono library and Googletest, can be employed to test performance and load capacity, as well as mitigate against potential risks such as corruption, contention, and other logic related errors. Overall, while the removal of a global lock may speed up the execution time of tasks, the side effects may negatively affect the system more than it helps without proper planning, testing and risk mitigation.

References

cppreference.com. (2022, October 1). *Date and Time utilities*. Retrieved from cppreference.com: https://en.cppreference.com/w/cpp/chrono

GeeksforGeeks. (2022, February 14). *Measure execution time of a function in C++.* Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

Google. (2023, January 17). *Googletest User's Guide*. Retrieved from Googletest: https://google.github.io/googletest/

Newbery, J. (2017, July 21). *What did Bitcoin Core contributors ever do for us?* Retrieved from Medium: https://medium.com/@jfnewbery/what-did-bitcoin-core-contributors-ever-do-for-us-39fc2fedb5ef

Newbery, J., & Dong, C. (2021, May 12). *Prune g_chainman usage in auxiliary modules (refactoring)*. Retrieved from Bitcoin Core PR Review Club: https://bitcoincore.reviews/21767

Newbery, J., & Towns, A. (2021, June 2). *Net_processing: lock clean up (p2p, refactoring)*. Retrieved from Bitcoin Core PR Review Club: https://bitcoincore.reviews/21527