

CS 474

**OBJECT ORIENTED
LANGUAGES AND
ENVIRONMENTS**

**PROJECT REPORT
BITCOIN**

By

-Arushi Mishra

-Shilpa Amarendrababu Sujatha

-Sireesha Basamsetty

-Upasna Menon

Table of Contents

SR.NO.	TOPIC	PAGE NO.
1.	Introduction	
2.	Characteristics of bitcoin	
3.	Advantages of bitcoin	
4.	Disadvantages of bitcoin	
5.	What to download	
6.	How to compile bitcoin core from source on ubuntu	
7.	Run the application	
8.	Open the aprof interface	
9.	Experience with the application	
10.	Operations that bitcoin performs	
11.	Steps	
12.	Methods using locks	
13.	Inputs to the application	
14.	Bottlenecks in the application	
15.	Graphs	
16.	Execution times	
17.	Costs	

Introduction

Bitcoin is a form of digital currency, created and held electronically. Bitcoin can be used to buy things electronically. In that sense, it's like conventional dollars, euros, or yen, which are also traded digitally. However, bitcoin's most important characteristic, and the thing that makes it different to conventional money, is that it is decentralized. No single institution controls the bitcoin network. This puts some people at ease, because it means that a large bank can't control their money.

A software developer called Satoshi Nakamoto proposed bitcoin, which was an electronic payment system based on mathematical proof. The idea was to produce a currency independent of any central authority, transferable electronically, more or less instantly, with very low transaction fees.

Bitcoin is created digitally, by a community of people that anyone can join. Bitcoins are 'mined', using computing power in a distributed network.

Bitcoin is based on mathematics. Around the world, people are using software programs that follow a mathematical formula to produce bitcoins. The mathematical formula is freely available, so that anyone can check it.

The software is also open source, meaning that anyone can look at it to make sure that it does what it is supposed to.

Characteristics of Bitcoin

1. Decentralized

There is no one central authority that is controlling bitcoin. This means that one authority cannot tamper with the monetary policy and cause a complete meltdown or take away bitcoins from people.

2. Easy to set up

Conventional banks make you jump through hoops simply to open a bank account. However, you can set up a bitcoin address in seconds, with no fees payable.

3. Anonymous

Users can make use of several bitcoin addresses as they are not linked to names, addresses or any other personally identifying information.

4. Transparent

Bitcoin stores details of every single transaction that ever happened in the network called the **blockchain**. If you have a publicly used bitcoin address, anyone can tell how many bitcoins are stored at that address. They just don't know that it's yours. People can take measures to make sure that their activities are opaque by not using the same bitcoin addresses constantly or transferring to the same address again and again.

5. Fast

You can send money anywhere and it will arrive minutes later, as soon as the bitcoin network processes the payment.

6. Transaction fees are miniscule

Your bank may charge you a \$10 fee for international transfers. Bitcoin doesn't.

7. Non-repudiable

Once the bitcoins are sent from one account to another, there is no way of getting them back unless the sender sends them back to you.

Advantages

- You can send and receive bitcoins anywhere in the world at any time. No bank holidays. No borders. No bureaucracy. Bitcoin allows its users to be in full control of their money.
- No fee is required to receive bitcoins. Fees are unrelated to the amount transferred, so it's possible to send 100,000 bitcoins for the same fee it costs to send 1 bitcoin.
- Bitcoin transactions are secure, irreversible, and do not contain customers' sensitive or personal information. This protects merchants from losses caused by fraud or fraudulent chargebacks.
- Bitcoin users are in full control of their transactions. Bitcoin payments can be made without personal information tied to the transaction. This offers strong protection against identity theft. Bitcoin users can also protect their money with backup and encryption.
- All information concerning the Bitcoin money supply itself is readily available on the block chain for anybody to verify and use in real-time. No individual or organization can control or manipulate the Bitcoin protocol because it is cryptographically secure. This allows the core of Bitcoin to be trusted for being completely neutral, transparent and predictable.

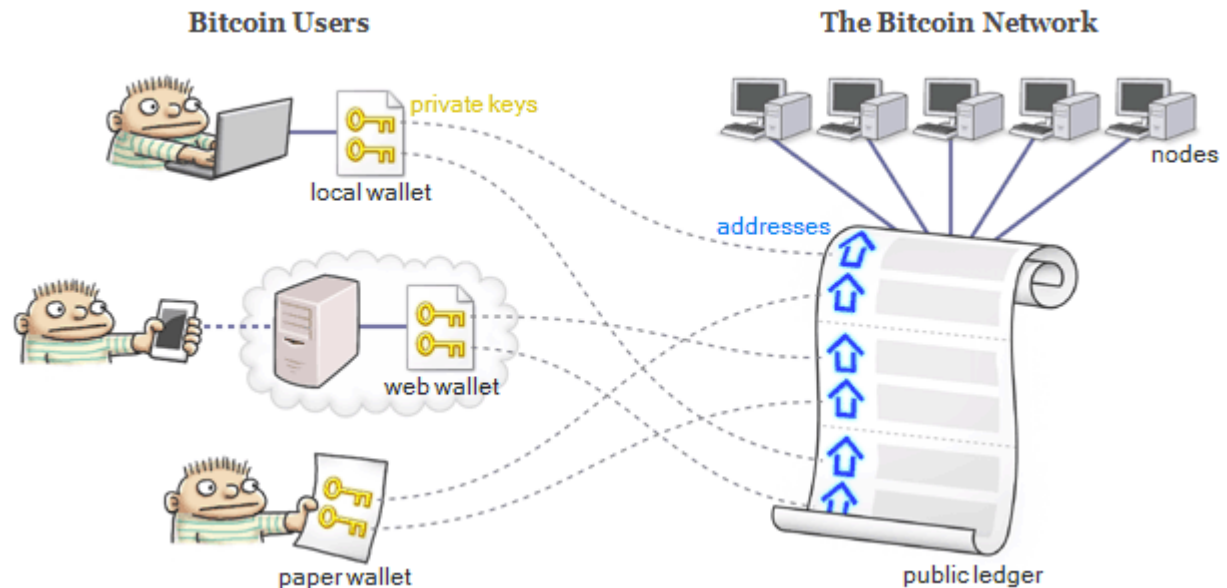
Disadvantages

- Many people are still unaware of Bitcoin. Every day, more businesses accept bitcoins because they want the advantages of doing so, but the list remains small and still needs to grow in order to benefit from network effects.
- The total value of bitcoins in circulation and the number of businesses using Bitcoin are still very small compared to what they could be. Therefore, relatively small events, trades, or business activities can significantly affect the price. In theory, this volatility will decrease as Bitcoin markets and the technology matures.
- Bitcoin software is still in beta with many incomplete features in active development. New tools, features, and services are being developed to make Bitcoin more secure and accessible to the masses. Some of these are still not ready for everyone. Most Bitcoin businesses are new and still offer no insurance. In general, Bitcoin is still in the process of maturing.

How Bitcoin Works!

1. Distribution of Bitcoins

There is no central server that keeps an account of all the bitcoins. There are thousands of such servers.



Each server in the Bitcoin network is known as a full node. This is like an electronic bookkeeper which can be set up by anyone and run. Every such node has a Public Ledger which is a record of every transaction which has ever been made. In order to make use of bitcoins a device is needed which functions like a wallet. It can be computer , mobile app , etc.

2. Transaction in bitcoin


Bitcoins can be sent from one person to the other by reassigning the bitcoins from one person's wallet to another by adding a transaction to the public ledger.

	Debit	Credit
Transaction e14768c1d648b98a52cb796af30af186140c5209a2fb53f1c8097db579f01cc0		
INPUTS		
Previous Output	Signature	
6120ceab25cfee257... : 0	3046022100aaf227f9...	0.0145
6eb36c1d347f8fdb6e... : 1	3046022100810c9d7a...	0.0923
OUTPUTS		
Address	Spent	
1NqUa3rFeStshjad1bhrEFFzW5Qw63Hbqv	<input checked="" type="checkbox"/>	0.0122
1FrtRypBwstUQ4X9KQdQByx6fWXLGGUPnt	<input checked="" type="checkbox"/>	0.0945
Transaction b6f4ec453a021ac561b01039f78e7168a653af176353c86d607343cc77e779b9		
INPUTS		
Previous Output	Signature	
e14768c1d648b98a52... : 0	30450221008a396b69...	0.0122
OUTPUTS		
Address	Spent	
1H1MoMgBaAikFhgAt3M4YJtetp4Hrns1Xu	<input type="checkbox"/>	0.001
1Q3Jw1wRXZyJ767mVtEpBVTH49HNBAB3v	<input checked="" type="checkbox"/>	0.0111
Transaction ee7df4afb472ae93427824d39191ec5940282c4da8cad326a3eade81884fbc36		
INPUTS		
Previous Output	Signature	
b6f4ec453a021ac561... : 1	3045022100f112ff63...	0.0111
OUTPUTS		
Address	Spent	
12MBAVJZ8pcVQQLMZHmWdxNMVFCxEgfk7P	<input type="checkbox"/>	0.001
1MuhZ3LbdJsUS431aZbRwD1Cd5gLDQm8m8	<input type="checkbox"/>	0.01



This depicts a transaction made in bitcoins where the input indicate that the bitcoins are being spent and output will assign those to another owner. The real names of the persons are not used instead we use addresses to transfer bitcoins.

3. Where do addresses come from

In order to transfer and receive bitcoins we need addresses. These addresses are generated by the bitcoin wallet. The wallet generates a private key which is approximately between 1 and 2256. This is encoded into letters and numbers for simplicity.

5JrFqG1rMLy6SkoWcZktr3HGqTSaXj63VAvQJNrrJ78Yhb1FtB 

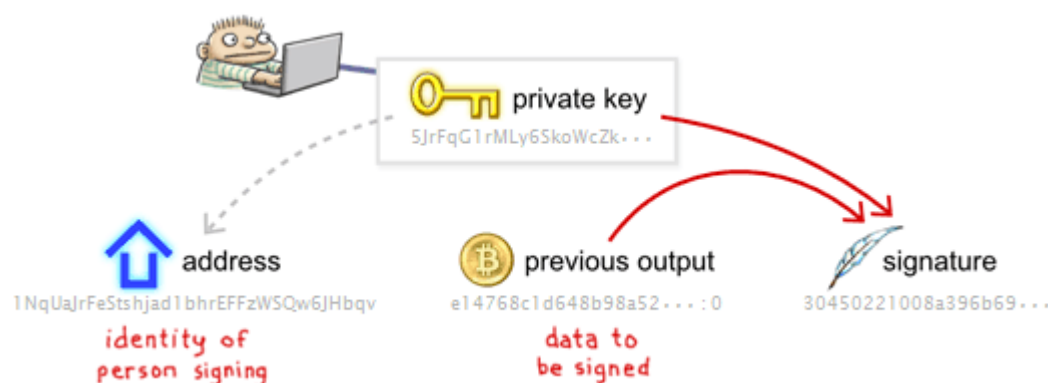
Now the wallet converts the private key to bitcoin address.

5JrFqG1rMLy6SkoWcZktr3HGqTSaXj63VAvQJNrrJ78Yhb1FtB   1NqUaJrFeStshjad1bhrEFFzWSQw6JHbqv

It is absolutely safe to give bitcoin addresses to people but we must keep our private keys to ourselves.

4. Transaction Authorization

A valid digital signature is as a proof of the fact the that address's owner has authorized the transaction.



A particular signature should satisfy an equation for it to be valid. Whenever a new transaction is received it is check to see the validity of it.



The digital signature is based on the concept of public key cryptography. This is how bitcoin works!

Aprof Introduction:

Aprof is a Valgrind tool which from one or more runs of a program automatically measures how the performance of individual routines scales as a function of the input size.

We have used the three graphs that are evaluated using aprof namely

- RMS Frequency plot
- Cost plot

RMS Frequency plot:

RMS metric, which we call read memory size is used for estimating the input size of a routine invocation:

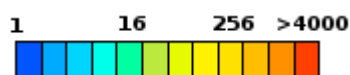
The read memory size (RMS) of the execution of a routine f is the number of distinct memory cells first accessed by f , or by a descendant of f in the call tree, with a read operation.

This graph uses the frequency of input sizes as its y axis and the RMS values are denoted by the x axis. The routine profile, shown below, can be useful if we want to know the exact frequency (or cost) of each RMS value.

Routine profile				
rms	min ...	avg c...	max...	freq
1	2	2	2	355910
3	18	18.6	20	90273
4	22	34.1	42	54068
5	40	50.6	66	35427
6	46	68.4	92	25239
7	66	87.1	120	19033
8	72	106.8	148	14703
9	94	127	172	11734
10	106	148.3	202	9569
11	126	169.7	226	8091
12	148	192.2	264	6818
13	170	214.5	304	5777
14	188	237.4	318	5058
15	208	261.4	332	4328
16	230	285.1	372	3979

Cost Plot:

On x axis we have the RMS metric. This can be seen as an estimate of input sizes on which your function was executed. Instead, on y axis we find an execution cost metric (by default aprof counts the number of executed basic blocks). Points of the chart are colored based on their frequency (number of times your routine was executed on a specific RMS). The legend on the top specifies the frequency value of each color:



Experience with the application

- The Bitcoin application makes use of a secure server. Every user who uses the bitcoin wallet has to provide valid credentials in the authorization step.
- The wallet provides a platform for the transactions to be made.
- Since, the application deals with currency exchange which is an expensive operation to be worked on in real-time using the core wallet, we made use of the bitcoin testnet which is the testing platform.
- The bitcoin testnet is used to generate various input combinations, study the performance issues and understand the application bottlenecks.
- We used the bitcoin testnet faucet to obtain the free bitcoins through the testnet wallet where we can perform various transactions.
- The downloading of the blockchains to get the application updated to the current date is a laborious task.
- The download of the transactions is indeed time consuming which takes about 4 to 5 days.
- We can fasten up the process by blocking the other operations that are running using the command `$sudo mount -o remount,nobarrier /`. This helps to get the application updated fast.
- After this operation we can use the command `$ sudo mount -o remount,barrier /` which unblocks the operations that were being blocked.
- This helps to update the bitcoin wallet and keep it up to date. The further transactions are done from this point.
- Different values of input are sent from the testnet faucet. Initially we sent four inputs each of size 0.024 bitcoins. Next, we sent milli and micro bitcoins from the wallet to the coinbase account generated.
- The graphs generated from these inputs for the various functions are studied and variations in the output is studied.
- The threaded functions are studied and the biggest bottleneck being the lock contention of `cp.main` is studied.
- The details of the above mentioned points are explained in the following sections below.

Operations that Bitcoin performs:

1. Startup

Starting the bitcoin application.

2. Initialization

Upon startup, some initialization routines are performed that can handle concurrent operations.

3. Node discovery

Finds out about other bitcoin nodes that may exist.

4. Node connectivity

Connects to those bitcoin nodes.

5. Sockets and messages

Processes messages from one node to the other using socket connections.

6. Block exchange

Advertising the inventory of blocks to each other and exchange of these blocks also.

7. Transaction exchange

Exchange and relay of transactions with each other.

8. Wallet services

Client creates transactions using the local wallet. Uses bitcoin addresses for that purpose.

9. Rpc interface

Client offers an rpc interface over HTTP over sockets to perform functions and operations to manage the wallet.

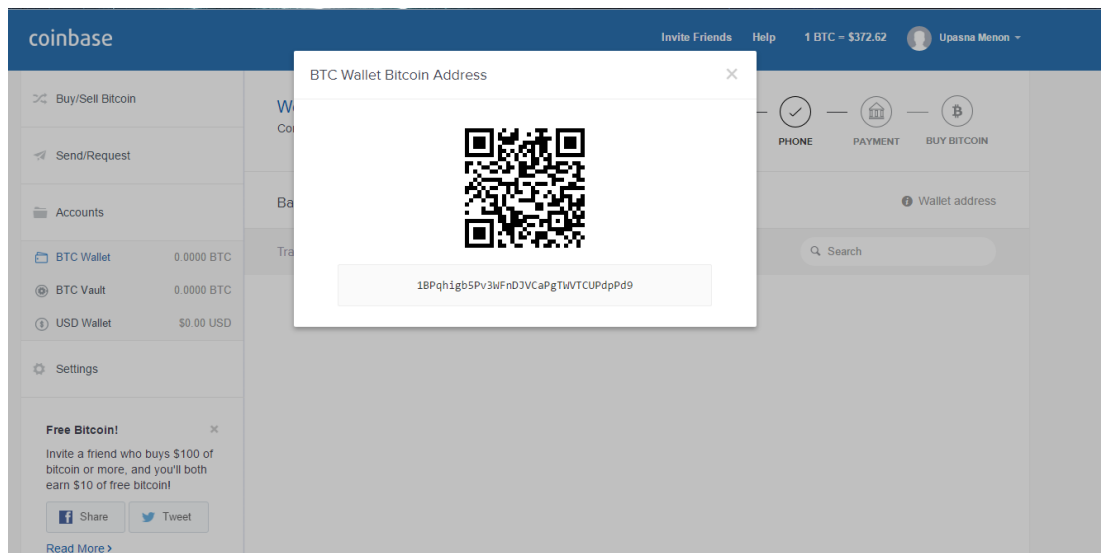
10. User interface

This is the bitcoin-qt interface.

Getting Started with sending bitcoins:

1) Create an account in Coinbase :

- Firstly we need to create a bitcoin account . This can be done using Coinbase.
- A coinbase account can be created using <https://www.coinbase.com/>. Each coinbase account has a unique Bitcoin address as shown below. This is the address of your bitcoin wallet. While sending transactions this address is used to recognize a person's account.



2) Sending bitcoins as input :

In our Bitcoin application we used Testnet faucet (testing interface of Bitcoin for developers) , where you can send free test bitcoins from one account to another.

TP's TestNet Faucet



This is a Bitcoin TestNet Faucet and eWallet. This website is for testing purposes only. Please keep in mind that the accounts may be periodically wiped without warning.

The service is provided for free at a cost to ourselves, so please donate some Bitcoins to [1DRnurMWfTWXL9oG8iVED8r6qubqPKw7Vj](https://blockchain.info/address/1DRnurMWfTWXL9oG8iVED8r6qubqPKw7Vj) if you wish to support us. Thank you in advance.

TestNet Faucet

The Faucet currently has 2.32799572 TestNet Bitcoins

We are giving away 0.023 BTC per request

Please input your TestNet Address to receive some free coins:

Please remember - **don't hoard TestNet coins or try to sell them**. TestNet coins are worthless, but useful. They are useful because they are worthless. If you will add value to them, they will be useless, therefore worthless.

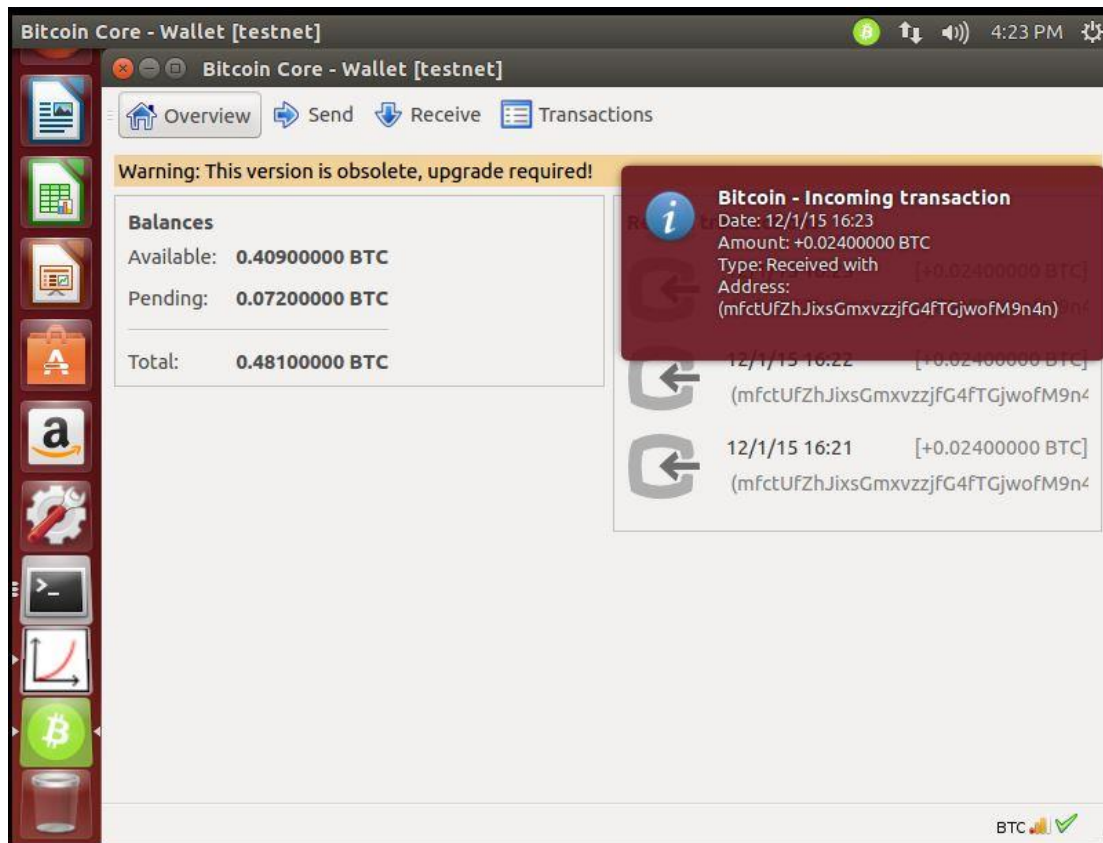
If you try to drain the Faucet dry, you won't be doing anyone a favour and you will only piss off the core developers. If you try to sell the coins, the devs will just reset the TestNet and you will lose everything.



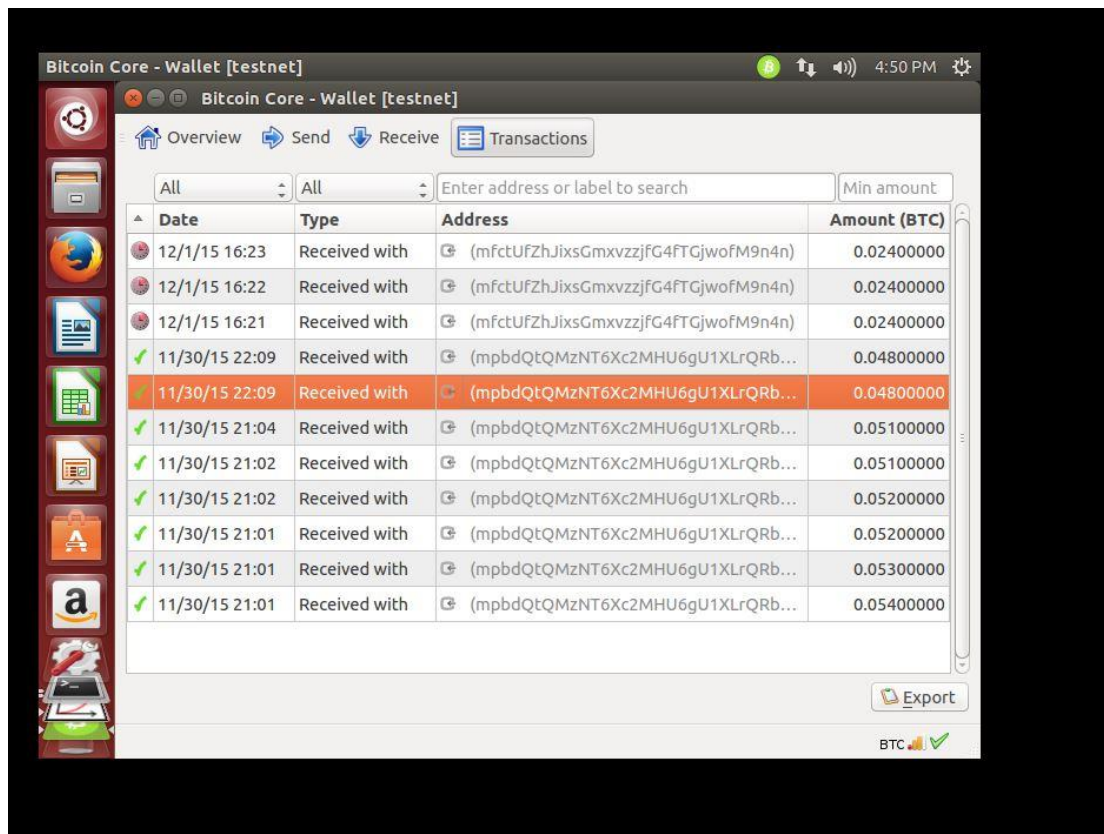
The testnet faucet website is obtained from the following link:

<http://tpfaucet.appspot.com/>

- When you send a transaction from Testnet it appears in our bitcoin wallet.
- The snapshot below shows an incoming transaction that was sent from testnet.



- The snapshot shows the list of transactions that we have made. Some of them are received and appear with green tick mark.
- The pending transactions are the ones represented with a red circle.
- The pending state of transactions are because they haven't yet been sent by the server.



What is actually happening?

Flow of the program :

The client calls `SendMessage()` method in `main.cpp` periodically which then calls `ResendWalletTransactions` to send transactions generated locally. This routine checks whether there has been a new block since last time, if so, and the local transaction is not in the block, the transactions are sent to all the nodes. This takes place in an interval of 30 minutes.

For the advertisement, the client calls `SendMessage()` periodically in `main.cpp` which determines whether a message should be sent to remote node or not.

When the client receives a tx message via a transaction, it calls `RelayMessage` which calls `RelayInventory`, which then makes sure that everything in the inventory is queued up to be sent to all other nodes.

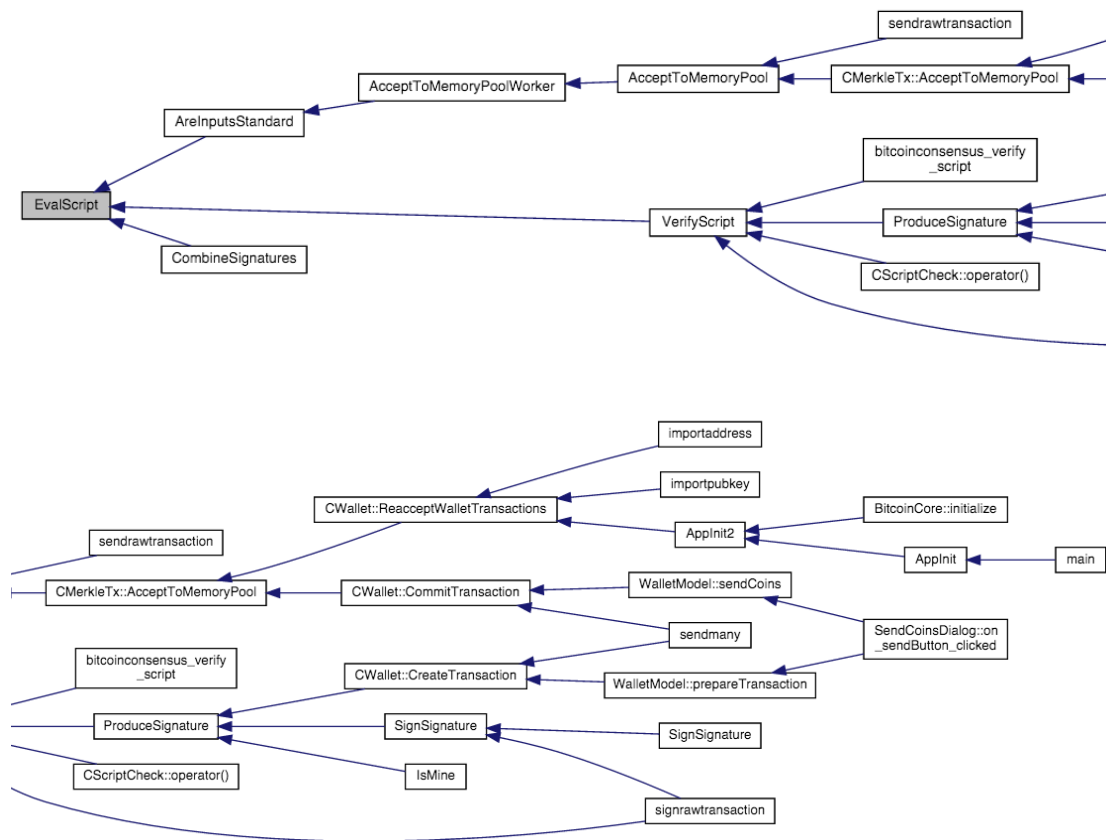
Take for example – `interpreter.cpp` and the method `EvalScript`

1. EvalScript(interpreter.cpp):

Steps:

1. Checks if inputs are standard.
2. If so, accept to memory pool walker and add transaction to memory pool.
3. Draw Transaction or accept or accept to memory pool
4. If accept to memory pool, reaccept wallet transactions or commit transaction.
5. If reaccept wallet transaction then, either import address, import pubkey or initialize bitcoin which then goes to the main function.
6. If commit transaction then, send coins.
7. Then `VerifyScript` method is called which does the following:
8. Produce a script signature using a generic signature creator.
9. Create a new transaction paying the recipients with a new set of coins and send coins after that.
10. Or, produce a script signature for transaction and sign signature.
11. In the end, combine the two script signatures using a generic signature checker.

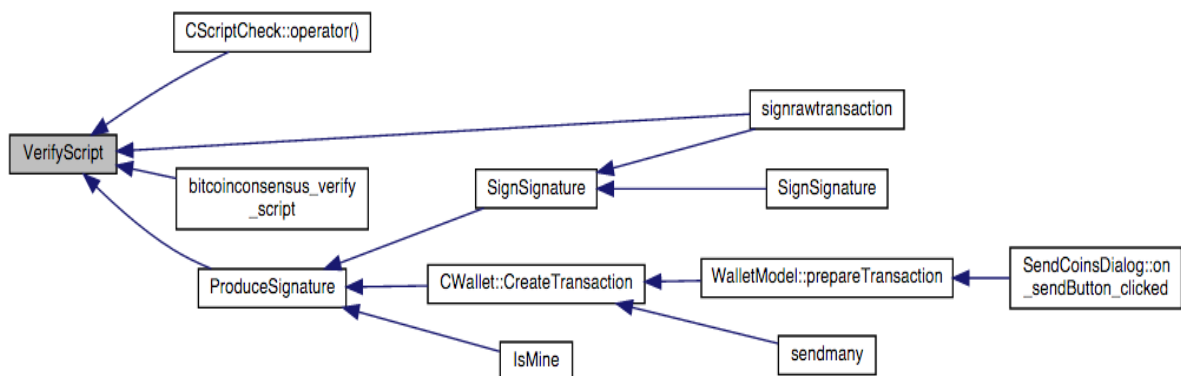
The caller graph for this method is as follows:



2. VerifyScript(interpreter.cpp):

Steps:

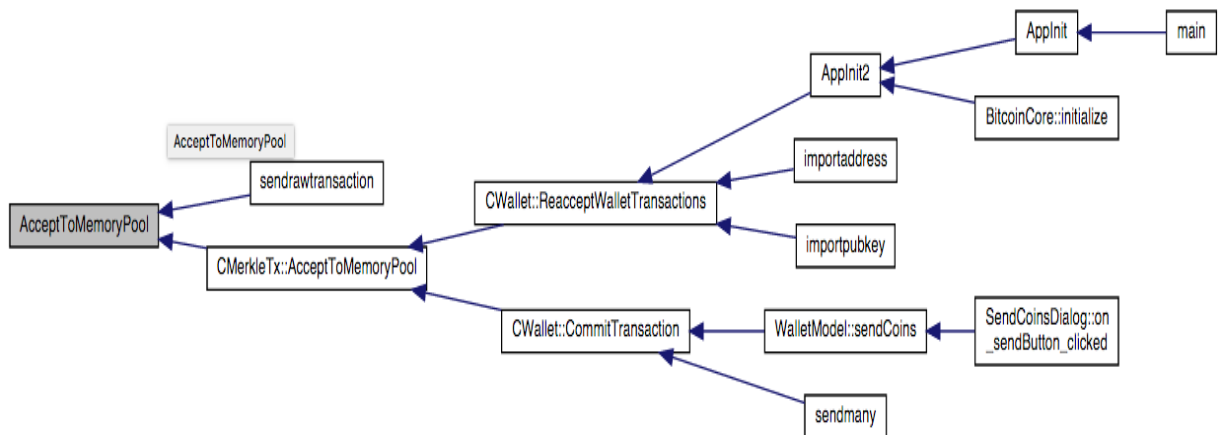
1. Either, produce a script signature using a generic signature creator.
2. Create a new transaction paying the recipients with a new set of coins selected by `SelectCoins()`.
3. Then prepare transaction and send as many coins.
4. This method returns 1 if the input of the serialized transaction pointed to by `txTo` correctly spends the script.



3. AcceptToMemoryPool(main.cpp):

Steps:

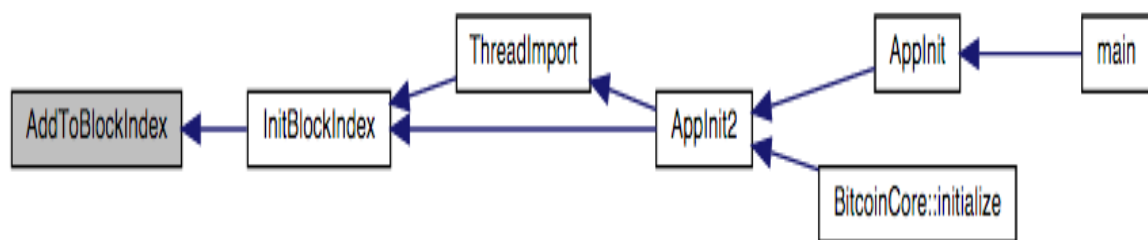
1. Either send draw transaction or accept to memory pool.
2. If accept to memory pool, then, reaccept wallet transactions or commit transactions.
3. If reaccept wallet transactions, then initialize bitcoin and go into main function.
4. Else, send as many coins on user input.



4. AddToBlockIndex(main.cpp):

Steps:

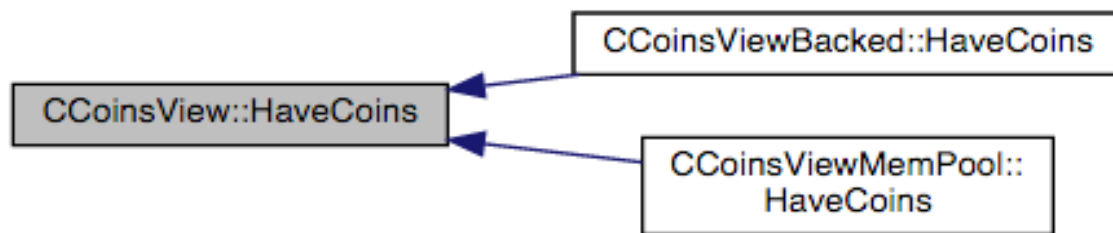
1. Initialize a new block tree database and block data on disk.
2. Thread import.
3. Initialize bitcoin.
4. Bitcoin core is initialized and then main function is called.



5. CCoinsView::HaveCoins(coins.cpp):

Steps:

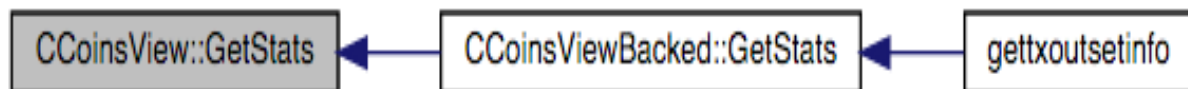
1. Check whether we have data for a given txid in memory pool.
2. Check for same in backed data.



6. CCoinsview::GetStats(coins.cpp):

Steps:

1. Calculate statistics about the unspent transaction output set.
2. Collect all this information.



Different inputs given to the application:

1. Size of the input i.e. amount transferred from one account to the other account. Here we have transferred from Testnet to our bitcoin application.

INPUT 1 - 0.096 Bitcoins

INPUT 2 - 10 MicroBitcoins

INPUT 3 - 10 MilliBitcoins

2. Types of input i.e. Bitcoin, milliBitcoin and microBitcoin

PROFILING

INPUT 1 : 0.096 BITCOINS

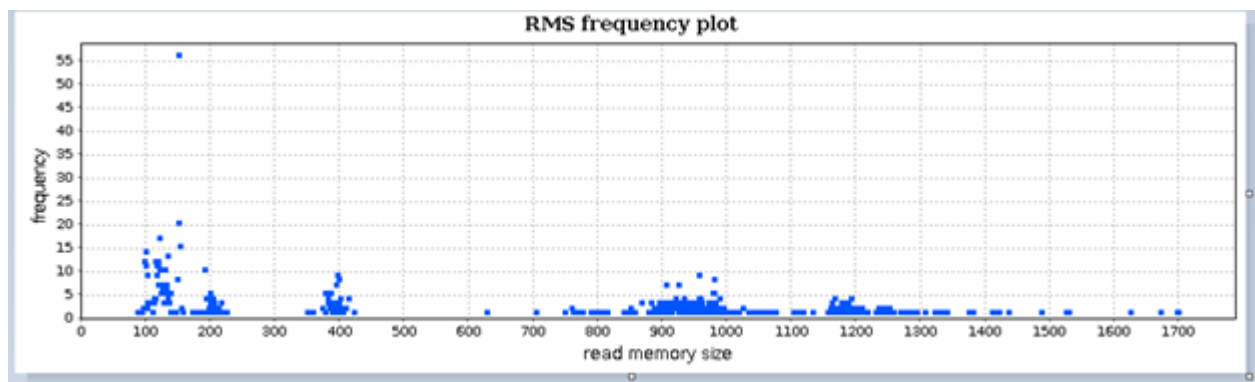
- After performing four input transactions, where each input transaction sends 0.024 bitcoins from Testnet to our bitcoin application ,we have analysed the .aprof files that were created from these transactions.
- By examining these .aprof files We noticed that a few routines that were responsible for excessive execution time and resource usage .

1) EvalScript:

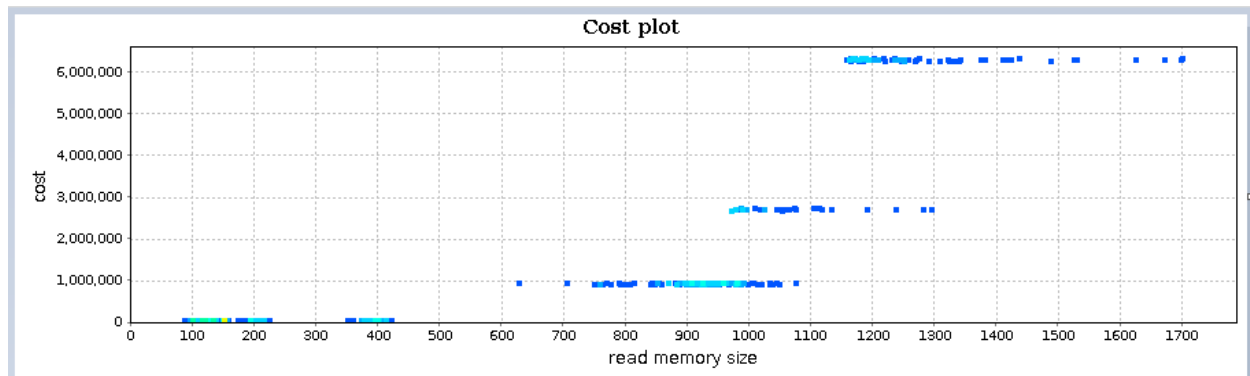
This method does the following

It checks for standard accepted inputs .These accepted inputs are added to the memory pool. Once they are accepted , either it re-accepts the transactions or commits it to save it to the current transaction. If it chooses to reaccept wallet transactions then the address is imported and bitcoin is initialized .

The cost and RMS plots for this method is as follows:



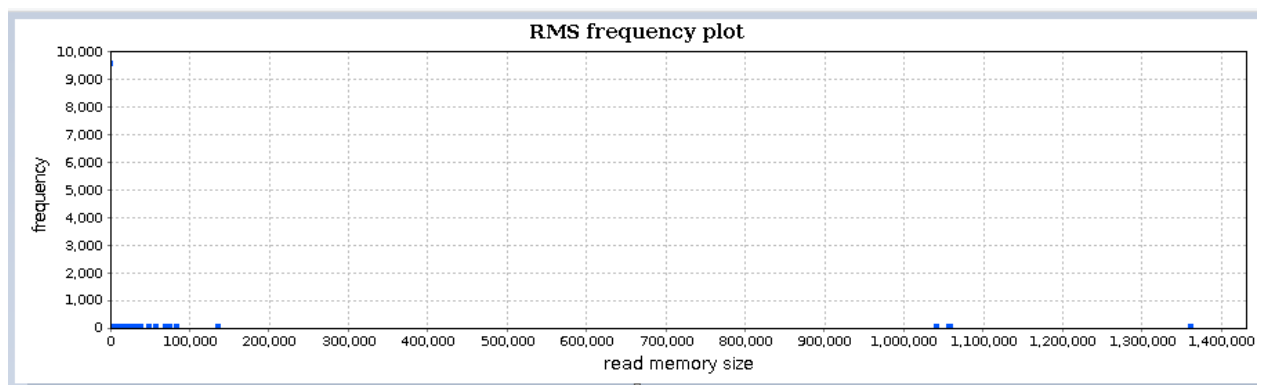
RMS Plot : The RMS plots show high read values at the time the input is being read based on the memory space or size the plot would be taking. The accepted inputs show different RMS values with respect to frequency based on the various factors of the input. The plot is shows distributed RMS values for similar levels of frequency.



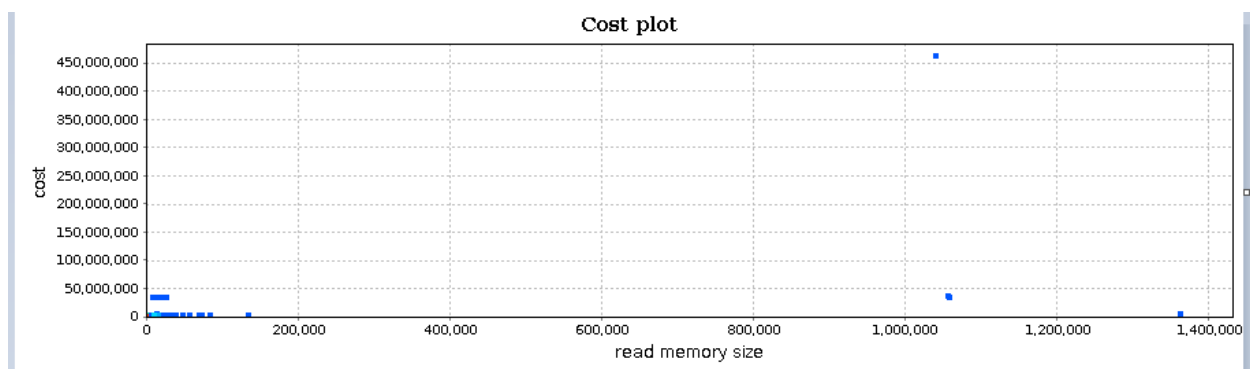
Cost Plot : The costs more or less , have dispersed values across the plot. The cost depends on the inputs that are accepted and the read memory size . Each accepted input has arbitrary values of cost . As the memory size increases thereby the cost also increases.

2) ProcessMessages(main.cpp)

Processes the incoming messages by maintaining the order of the incoming messages. If the buffer is too full it doesn't bother to respond to the message. Otherwise it gets the next message and debugs it. It performs a check on the memory size received and checking the message format.



RMS plot : The RMS plot for the ProcessMessage routine lays low . The transaction input size being 0.096 bitcoins. Most of the processing is done early and the frequency is low with increasing RMS size. Once the processing is done there is no change in the plot.

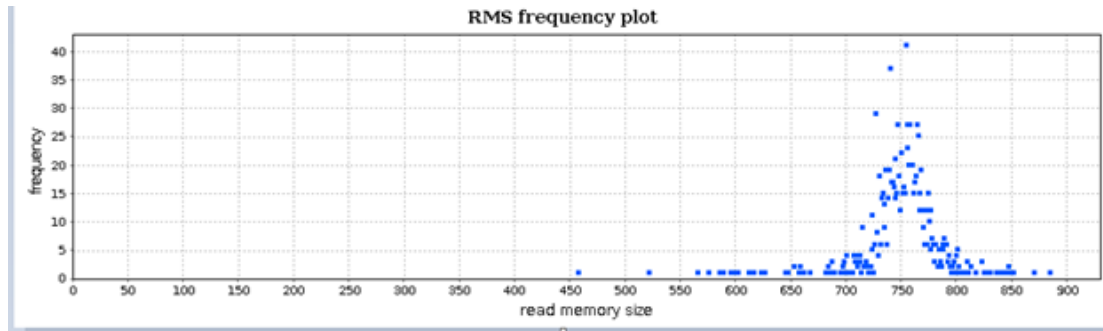


Cost plot : The graph shows values of low cost initially. For the incoming messages the processing cost with respect to the read memory size is shown in the plot. The cost also increases suddenly for a

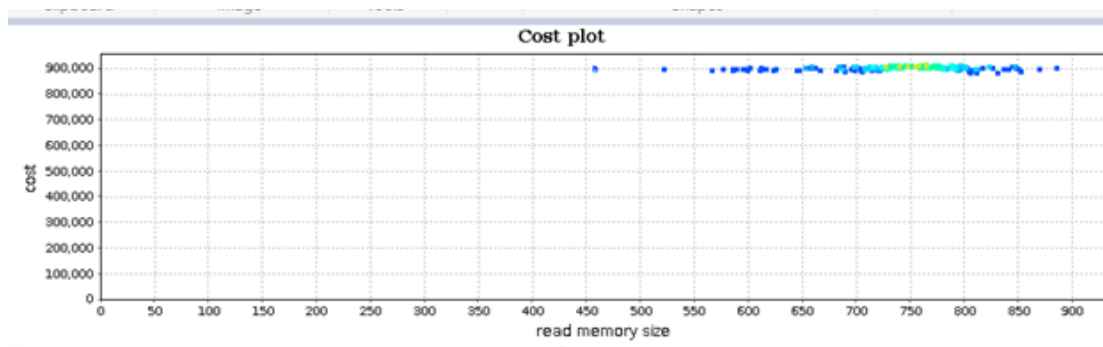
read memory size of 1,000,000 after which it drastically drops to zero. This increase can be implied to a buffer overflow.

3) TransactionSignatureChecker(Interpreter.cpp)

When one sets up a transaction, the input of the transaction needs to have a script with a signature to show that you can spend those coins. The transaction signature routine takes care of this.



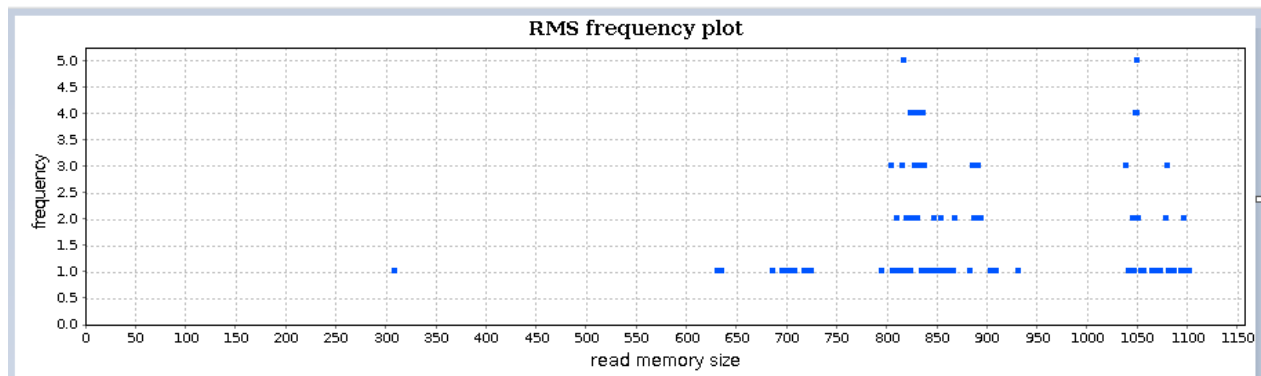
RMS plot: This graph shows clearly that when the bitcoins were transferred to the application there was a gradual increase in the value of the coins that could be spent. The transaction signature routine shows high value at a given point. Once the transaction is completed the value of it falls below gradually which is evident from the graph.



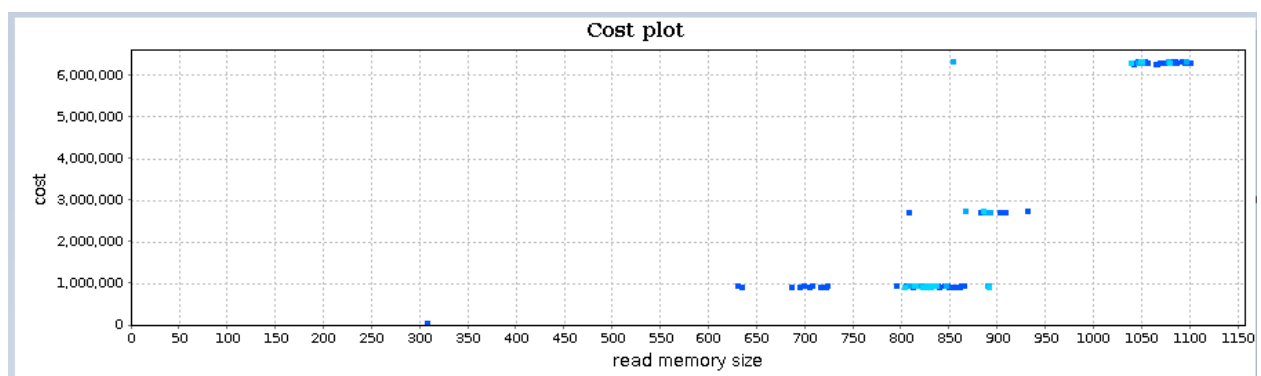
Cost plot: Due to the transaction a block of memory is being assigned at the point when the incoming transaction signal is received. this causes the scattered graph to be shown which indicates the memory used by the block with the cost per unit of the transfer being done.

4) VerifyScript(Interpreter.cpp)

The functionality of this routine is that it produces a script signature using a generic signature creator. Creates a new transaction paying the recipients with a new set of coins and send these coins after that to produce a script signature for transaction and sign signature. In the end, combination of the two script signatures is done using a generic signature checker.



RMS plot: This plot shows how the process of creating a new transaction to send and receive coins is being affected with the creation of new wallets. The distributed plots indicate that a wallet was created with the new creation of coins. The memory allocated is shown.



Cost plot: This plot shows the overheads of adding the new wallet when a transaction is created. This happens only with the high values of the overhead. The split values indicate an instance at that point.

Cost per method:

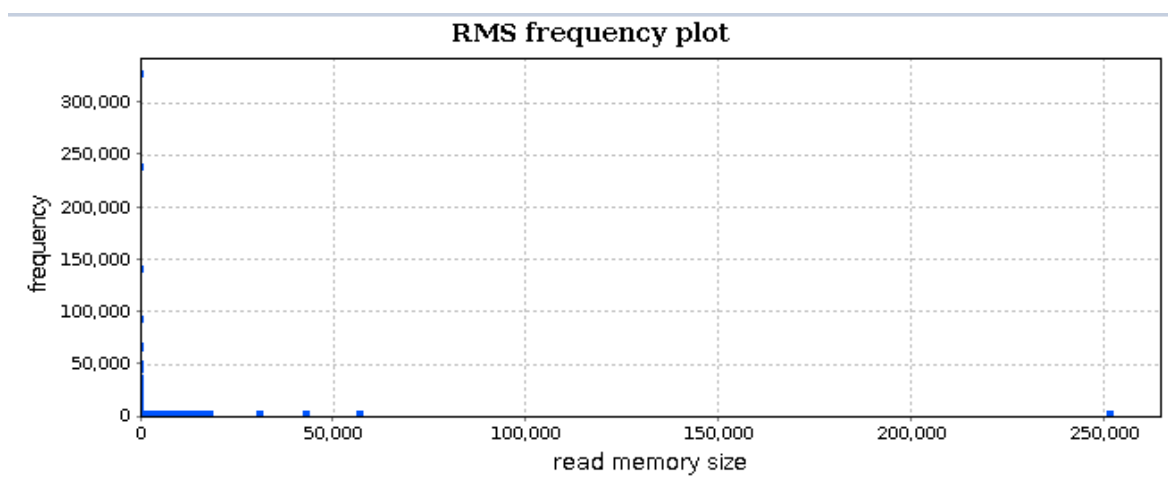
EvalScript	36927844375
TransactionSignature	368678729290
hVerifyScript	36936695743
CPubKey	200854875
ProcessMessages	183754567

INPUT 2 : 10 microBitcoins

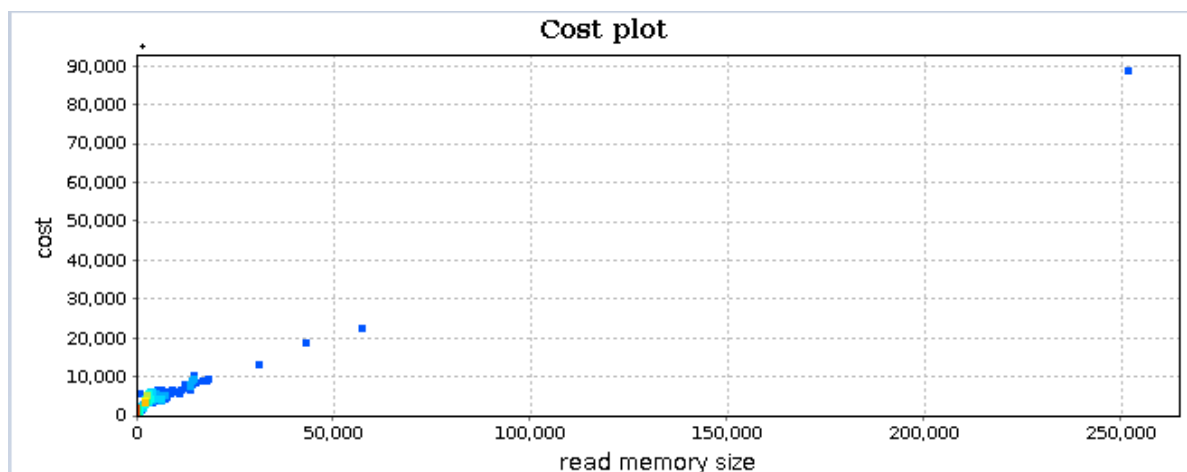
- Now we analyze the .aprof files that are generated from a second input.
- As second input, we have used micro-Bitcoins and studied the graphs generated by aprof.
- The following are the graphs with excessive execution time and resource usage.

Method 1 : TableBuilder

TableBuilder is the method mainly used to write while the different blocks were being downloaded. We ensure that the data of the current block is kept until we see the next block. We do not emit the index entry for a block until we have seen the first key for the next data block. This allows us to use shorter keys in the index block.



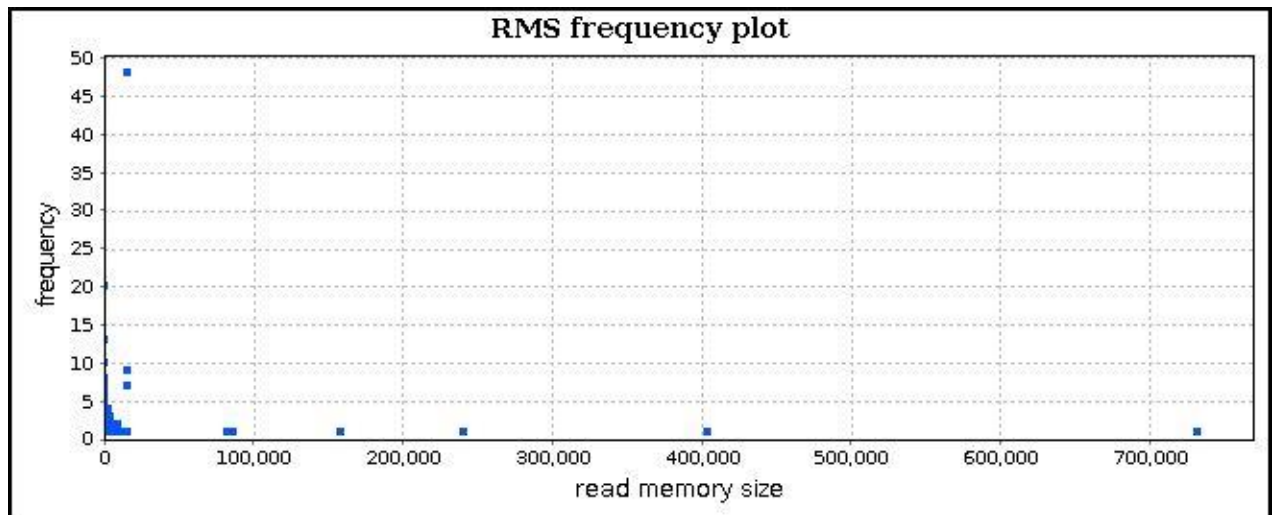
- The Table Builder function mainly focuses on loading the blockchain and setting up our application to make the transactions possible . Therefore , it is clearly an input sensitive method. From our cost plot , we can infer that the graph is linear. This shows that with smaller inputs the blockchain transactions are loaded faster.



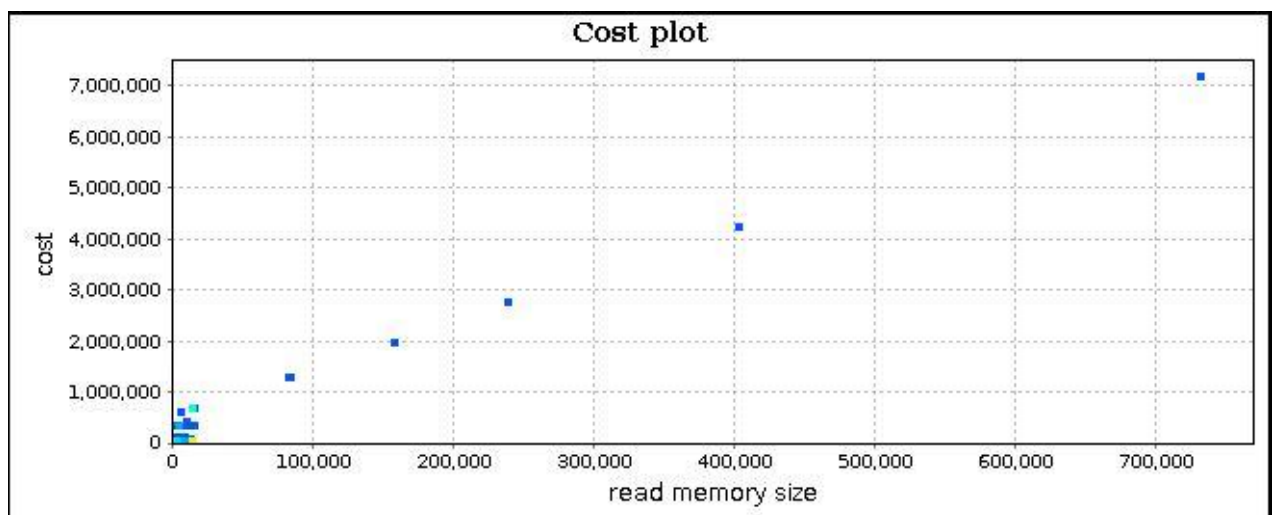
- As the read memory size increases our cost increases linearly. This shows that the TableBuilder method is RMS sensitive.

Method 2: ReceiveMessage

This method is used to get the current input message or to get a new message. It is common for nodes with good ping times to suddenly become lagged, due to a new block arriving or other large transfer. Merely reporting ping time might fool the caller into thinking the node was still responsive, since ping time does not update until the ping is complete, which might take a while. So the ReceiveMessage method makes sure that if a ping is taking an unusually long time in flight, the caller can immediately detect that this is happening.



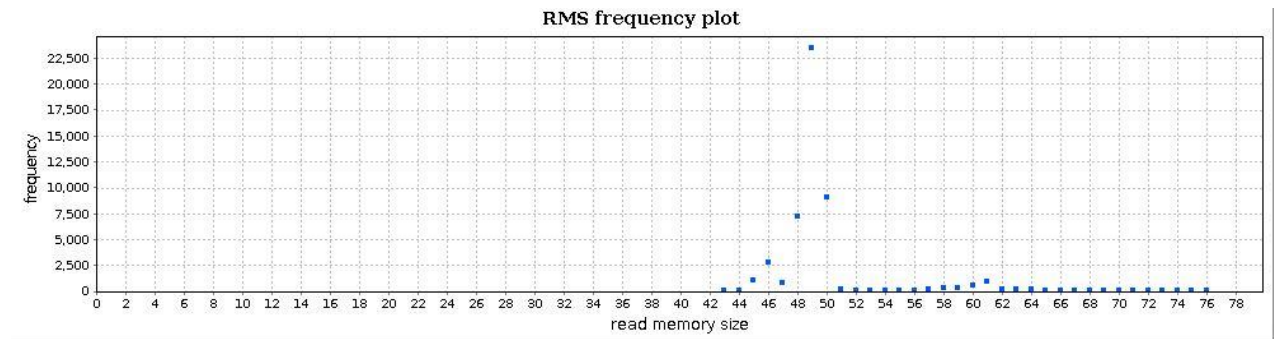
RMS plot : Initially for smaller Read memory sizes , the frequency of incoming messages is higher . This explains the initial blockchain overhead. Therefore the RMS plot is cluttered in the beginning when $RMS < 50,000$. As the transaction of sending 10 microbitcoins continues the frequency of incoming messages becomes stable and remains low.



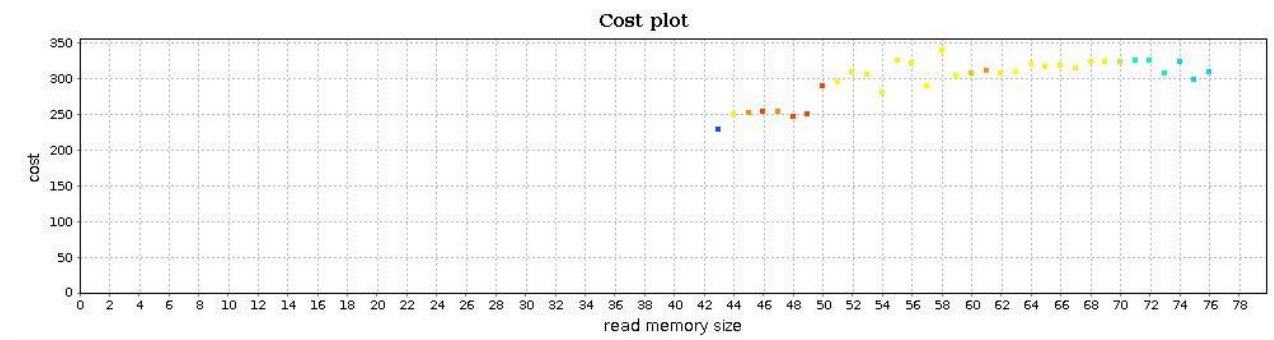
Cost plot : With increasing read memory size , the cost increases linearly. As the memory size builds up the cost of handling the messages increases linearly.

Method 3: LevelDBBlockHandler

This method is used to download the bitcoin blockchains which uses the leveldb database. It handles all the blockchains that are related to the bitcoin. It supports batching writes, forward and backward iteration, and compression of the data .



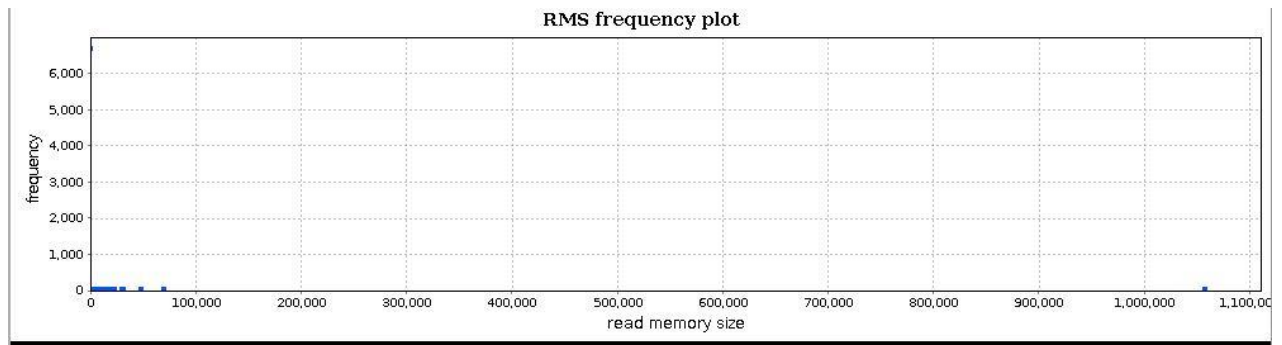
RMS plot : Once the Read memory has the loaded transaction , the LevelDB stores keys and values in arbitrary byte arrays, and data is sorted by key. This explains that until the transaction is loaded there are no changes in frequency with respect to read memory size. Once the transaction is available in the memory , the frequency rises a little and then stabilizes .



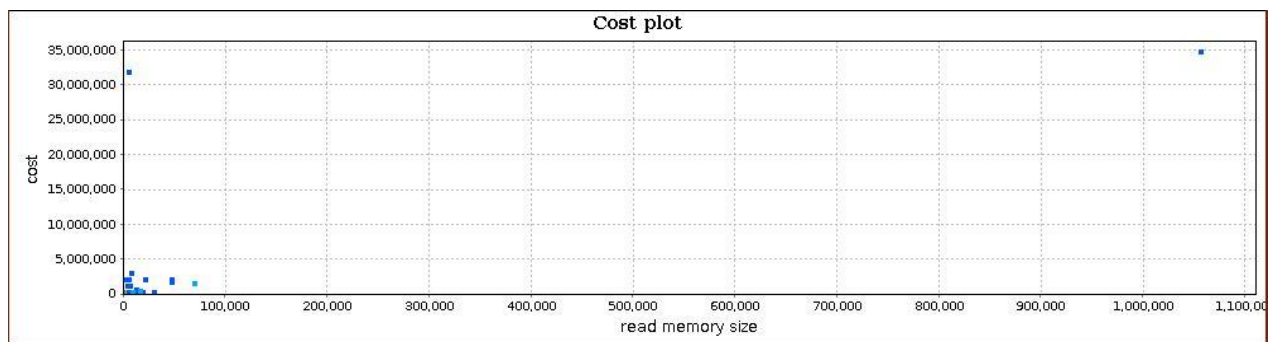
Cost Plot : Similarly for the cost plot , there is no cost until a read memory size of around 42 - 45 . After the initial stage , the levelDB performs the operations of storing the keys and values in byte arrays , after which it sorts the available data by the respective keys. Therefore the cost increases in a stable manner with no drops.

Method 4: ProcessMessage

This method is used to process the incoming messages by maintaining the order of the incoming messages. If the buffer is too full it doesn't bother to respond to the message. Otherwise it gets the next message and debugs it. If an incomplete message is found it checks for any failure. If any found then it deletes the current message and scans for the message start. It performs a check on the memory size received and checking the message format. The message is checked for the size , checksum value , data and the message start.



RMS plot : The RMS plot for the ProcessMessage routine lays low . The transaction input size is 10 microbitcoins . This is a very low value and so most of the processing is done early and the frequency is low with increasing RMS size. Once the processing is done there is no change in the plot.

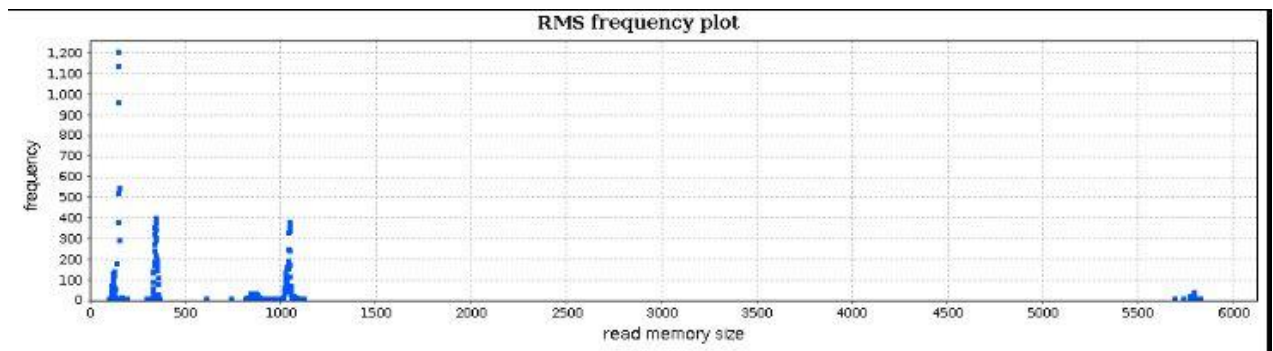


Cost plot : With the low input of 10 microbitcoins the processing is done quickly. The graph shows values of cost only in the first square which implies that for the incoming messages the processing cost with respect to the read memory size is shown in the plot. The cost remains low due to low processing required for the small input size.

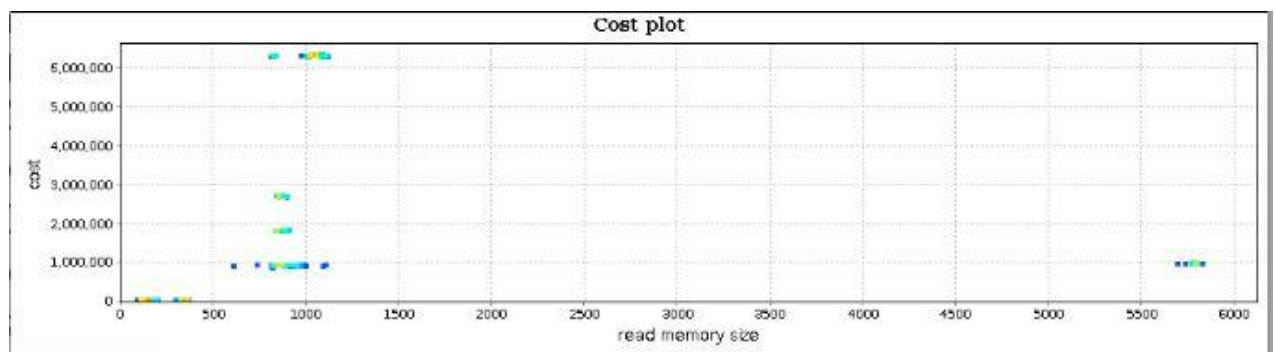
Input 3- 10 milliBitcoins

1. EvalScript (Interpreter.cpp)

As mentioned above this method checks for standard accepted inputs .These accepted inputs are added to the memory pool. Once they are accepted, either it re-accepts the transactions or commits it to save it to the current transaction. If it chooses to reaccept wallet transactions then the address is imported and bitcoin is initialized .



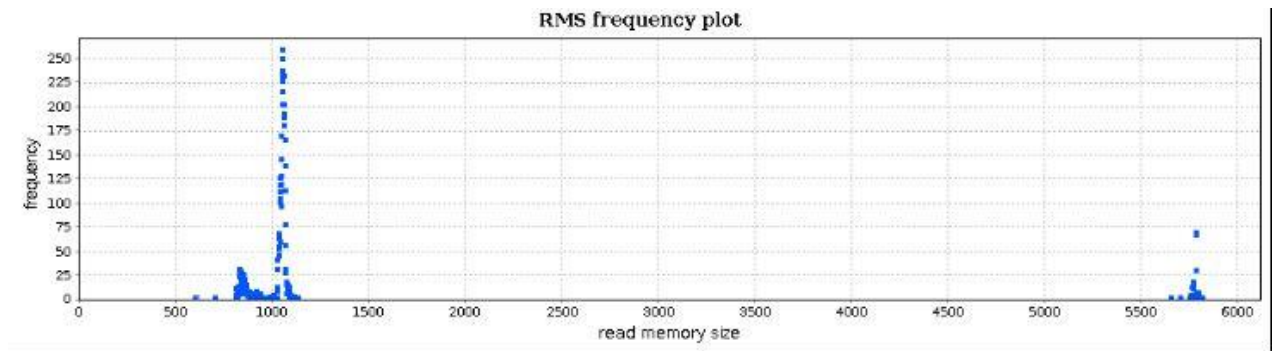
RMS Plot : The RMS plots show high read values at the time the input is being read based on the memory space or size the plot would be taking. For our low input of 10 millibitcoins with read memory size of < 500 and >200 the RMS frequency significantly rises. The same thing happens at RMS value of 1000.



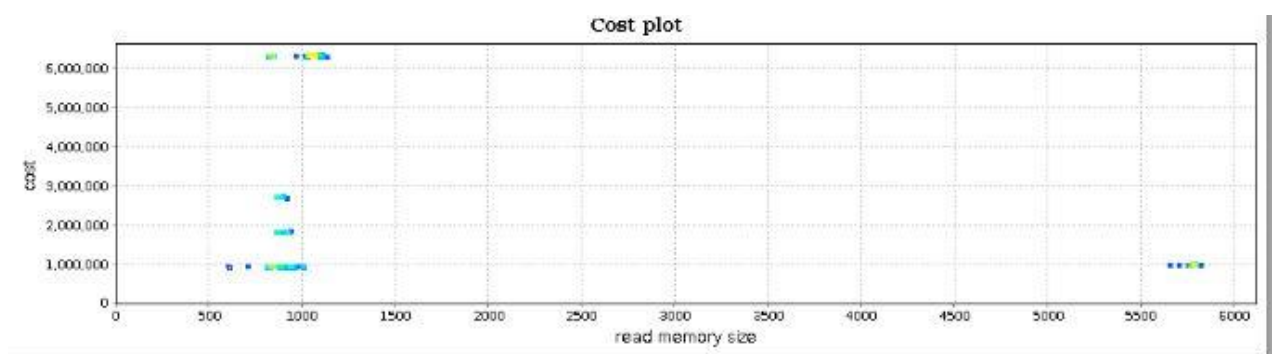
Cost Plot: The cost plot for EvalScript shows low values of cost from memory size <0> to <500>. The cost remains low here due to low processing required for small size. Once it reaches around <1000> the cost value rises very sharply.

2) VerifyScript (Interpreter.cpp)

Produce a script signature using a generic signature creator. Create a new transaction paying the recipients with a new set of coins and send coins after that. Or, produce a script signature for transaction and sign signature. In the end, combine the two script signatures using a generic signature checker.



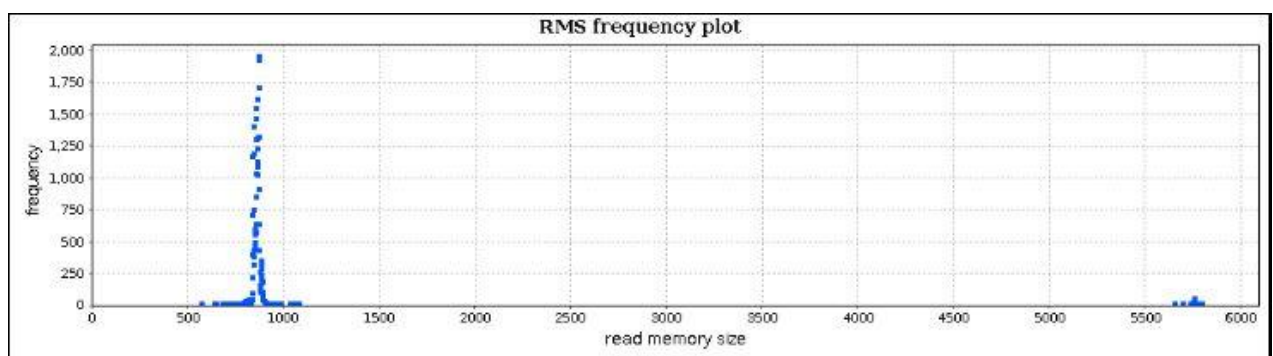
RMS Plot: The RMS Plot for VerifyScript shows how maximum frequency is concentrated in the memory interval of <800> to <1200>. The same thing happens in the interval of <5700> to <5800> also.



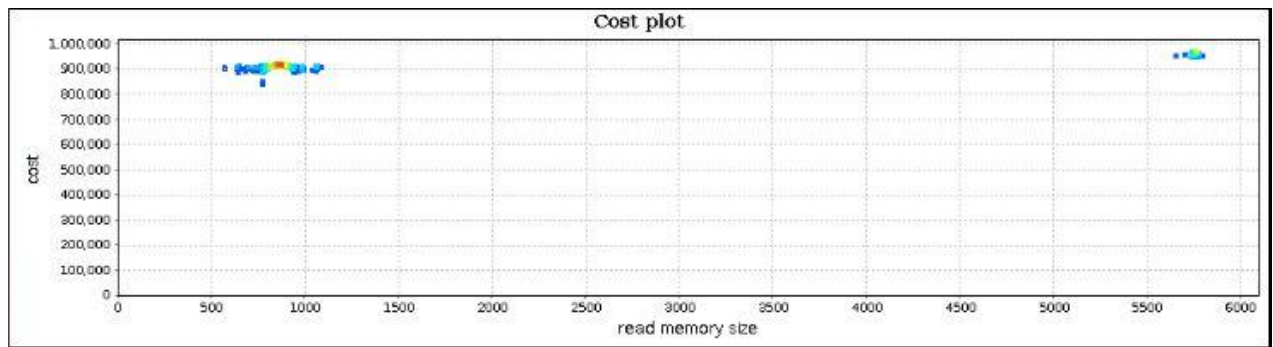
Cost Plot: The cost plot for VerifyScript shows how the cost is maximum from <700> to <1000> memory size. This is due to high processing in this interval.

3) TransactionSignature (Interpreter.cpp)

When one sets up a transaction, the input of the transaction needs to have a script with a signature to show that you can spend those coins. The transaction signature routine takes care of this.



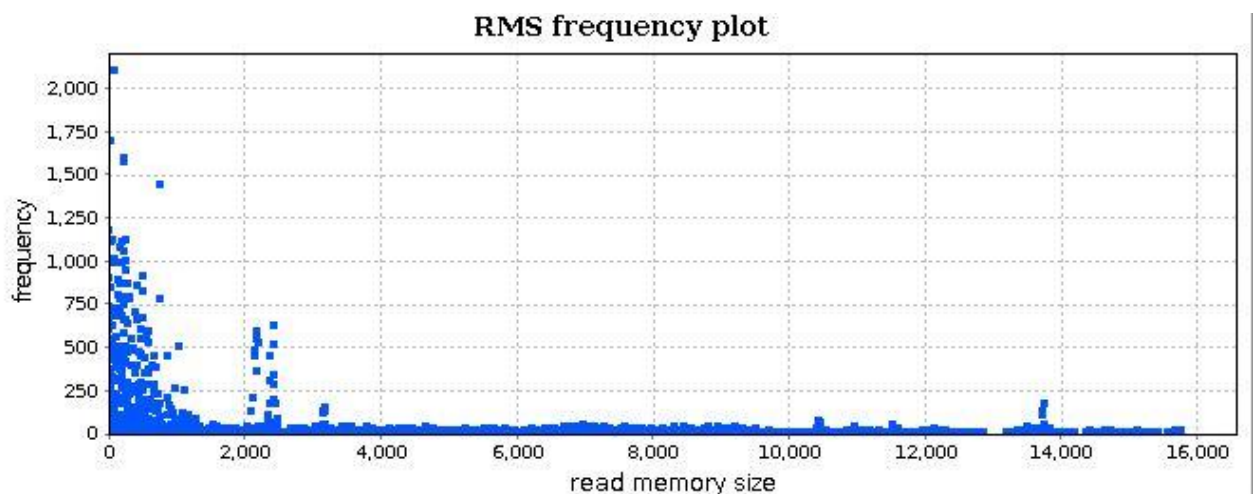
RMS Plot: The RMS Plot for TransactionSignature shows how the frequency is maximized from <500> to <1000>. This is due to high processing happening in that interval.



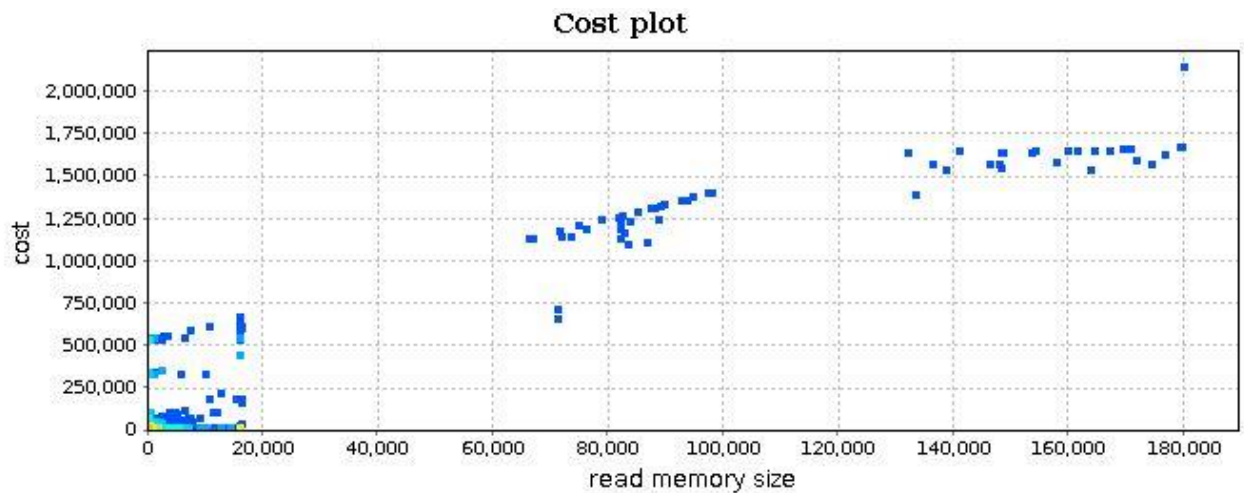
Cost Plot: The cost plot for TransactionSignature shows how from the interval of <500> to <1200> of the memory size the cost is of the range of 900,000. This is due to high processing of inputs.

4) CNode::getTotalRecvsize (Net.cpp)

- This method is used to receive the blockchains during the load of the bitcoin application. While it is being received the initial overhead is high. As the downloads continues we get the stabilization of the memory which is as seen above.



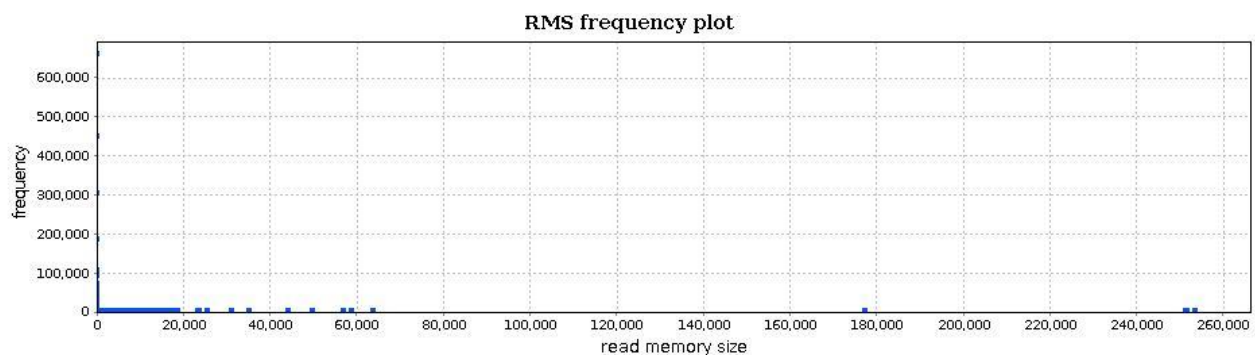
RMS Plot: The high read memory seen to be quadratic perhaps almost linear. This high value at the axis gives the shows that the data received at the start of the application is very high due to the load of the blockchains which requires the access of the methods. It is execute on a small input size most of the times as it is a divide and conquer sorting algorithm.



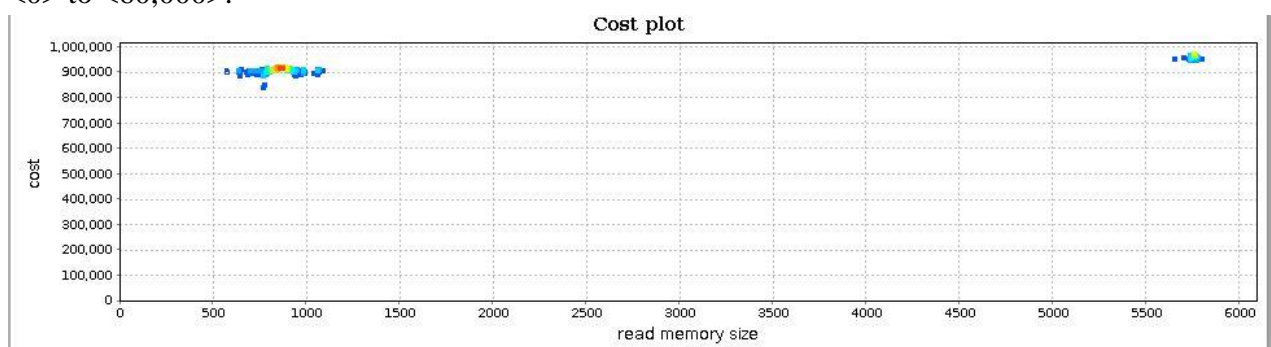
Cost Plot: This shows the application of the divide and conquer strategy to the method plot we get a distributed graph with the values over the line when the high cost values are being inputed.

5)TableBuilder (tablebuilder.cc)

This method consists of the sequence of blocks where each block consists of a certain size, block data, type and crc unit.



RMS Plot: The RMS Plot for TableBuilder shows extremely low frequency values throughout from $\langle 0 \rangle$ to $\langle 60,000 \rangle$.



Cost Plot: The cost plot shows huddled up values at an RMS size of around 500 - 1200 . It is bundled at a point showing only those areas with the data block.

Methods using Locks

1. ProcessMessages in main.cpp
2. ConnectNode in net.cpp
3. VerifyScript in interpreter.cpp
4. Dbimpl in Dbimpl.cc(mutex locks)
5. BackgroundCall in Dbimpl.cc
6. DoCompactionWork in Dbimpl.cc

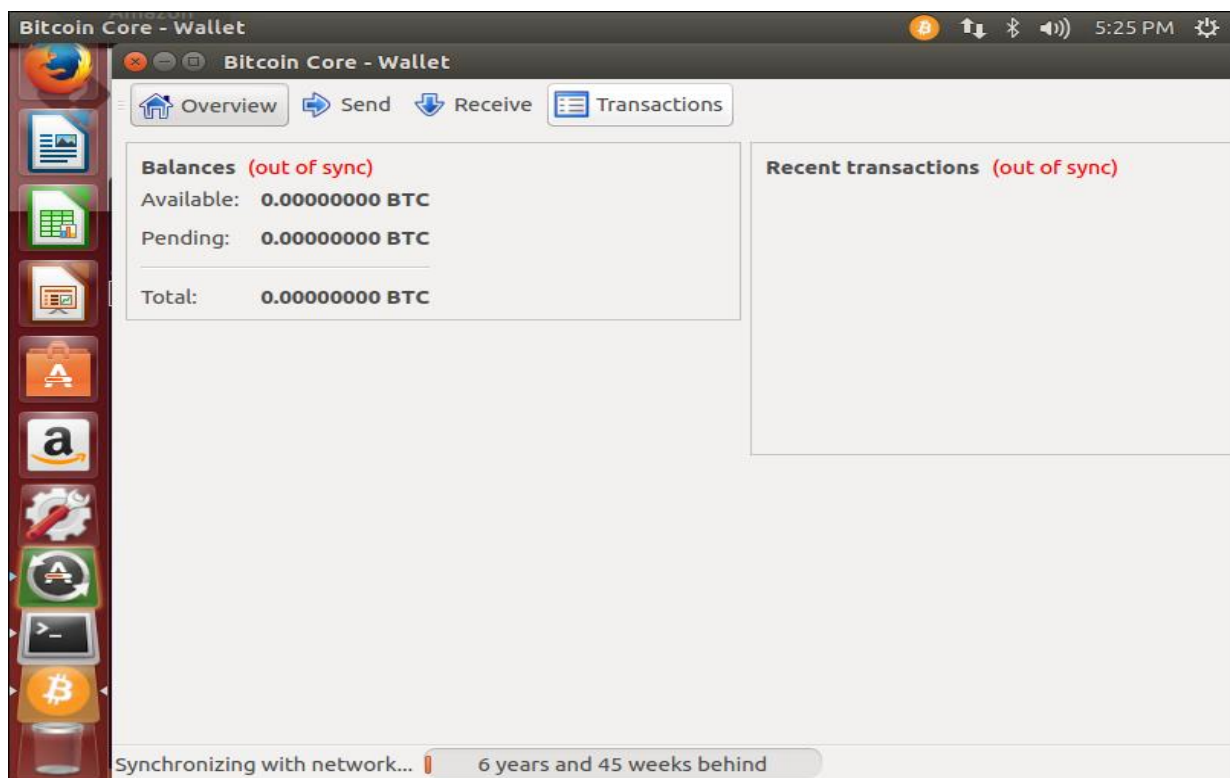
These are the methods that worsen the performance of the AUT.

Lock Contention and Bottleneck

- The Bitcoin core application is continuously freezing during synchronization making the RPC unresponsive. This happens as bitcoin uses more than one thread to analyze the downloaded blocks.
- What is expected from this application is that the RPC server and the GUI must not lose their connection with other nodes. There also seems to be heavy lock contention during some of the synchronization phases. This causes the RPC to act slowly.
- All of this happens because cs_main lock gets held up for a longer time. Whenever a new transaction takes place, it will get information from the transaction wallet, which takes both a cs_main and wallet lock. It can get stuck for a long period of time.
- So a solution to this problem is to copy the relevant data in the notification function. The root cause would be mitigated by changing ActivateBestChain/ActivateBestChainStep/ConnectTip logic to release the cs_main lock between attaching blocks.
Another thing that can be avoided is the cs_main lock at qt/transactionmodel.cpp.
- The cs_main lock is highly contented. Many execution paths acquire the lock and the cs_vSend, one such example is ProcessMessage('tx'). It locks the cs_main after which it can call pfrom->PushMessage if it rejects the tx. PushMessage() obtains pnode->cs_vSend.
- One more example is CheckBlock() which call PushGetBlocks() after it locks cs_main and this makes a call to PushMessage() which in turn locks pnode -> cs_vSend.
- As per our understanding , there is only one single execution path which acquires the locks in opposite order. ThreadMessageHandler() needs to lock cs_main before making SendMessage calls.
- This looks like a potential deadlock situation , however, SendMessage() does a Try_lock on cs_main first and it returns early if it doesn't get the lock before it calls IsInitialBlockDownload(). This would prevent deadlock scenario but can lead to thread starvation or livelock.
- From this we can observe that EvalScript, TransactionSignature, VerifyScript take maximum cost out of all the methods. These methods are from interpreter.cpp

Blockchain Bottleneck

- The bitcoin blockchain takes a long time to load. Initially while deploying the application it takes a tremendous amount of time in order to synchronize with the network.
- We used multiple windows and tabs when we worked to switch between them often and fast and we could not do this while Bitcoin was chugging along and slowing everything down.
- The downloading of the blockchains to get the application updated to the current date is a laborious task.
- The download of the transactions is indeed time consuming which takes about 4 to 5 days.



- We can fasten up the process by blocking the other operations that are running using the command `$sudo mount -o remount,nobarrier /`. This helps to get the application updated fast.
- After this operation we can use the command `$ sudo mount -o remount,barrier /` which unblocks the operations that were being blocked.
- This helps to update the bitcoin wallet and keep it up to date. The further transactions are done from this point.

References

Websites referred

1. <http://bitzuma.com/posts/compile-bitcoin-core-from-source-on-ubuntu/>
2. <https://github.com/bitcoin/bitcoin/blob/master/doc/build-unix.md>
3. https://dev.visucore.com/bitcoin/doxygen/interpreter_8cpp.html
4. https://dev.visucore.com/bitcoin/doxygen/class_c_coins_view.html