Ethereum Difficulty Updates

Aeron Buchanan, version 1

Conclusion

Ethereum difficulty would be more stable if either 1) updated very much less often than every block, and/or 2) updated with a more sophisticated formula. The nature of proof-of-work means block-times vary hugely: with a target block-time of 1, the raw stats give $\sigma > 0.9$. Without any updates, an optimized difficulty setting gives a running 500-block-time average ${}^{500}\sigma < 0.05$. This can be matched with sophisticated enough update strategies, such as a PID controller. Using the update strategy of the white paper V2 ${}^{500}\sigma \approx 0.1$. Demonstrations of stability are given below.

Introduction

Proof-of-Work (PoW) is an unfortunate necessity for dealing with "infinite virtual node" attacks in a consensus based event log (commonly referred to as the block-chain). Anyone can add events to the shared log, so which event log do you take as the master? Being a consensus, it must be the log with the most votes. However, without PoW it would be trivial to create a large number of nodes to vote for (verify) a thief's log in which they erased an entry which they had used to pay someone, thus allowing them to spend that money a second time. In a pure identity-based voting system, the person with the most resources can win control of the system, as they can out-vote an uncoordinated honest majority. Furthermore, if it costs nothing to add events then the system can be spammed by attackers and rendered useless. Introducing PoW solves both these problems. With PoW, every one of the N nodes gets to make many (but time-discrete) attempts to add events to their favored log (and thus vote for it), each with a low chance of success. It provides a better solution to the voting problem, because on average, everyone gets roughly a $\frac{m}{N}$ chance of voting for a particular event log with m being the multiple of the average node resources a particular node has. It also solves the spamming problem by making it take, on average, an appreciable amount of time to add events to the log. However, as the available processing power changes, the average number of attempts that can be made per time will increase, and so the effectiveness of the PoW will change. If processing power increases (faster nodes or more of them), the average time taken will decrease, reducing the effectiveness of the PoW, and vice versa. As such, the difficulty of a PoW attempt is adjusted. Here, I briefly discuss update strategies.

Strategy

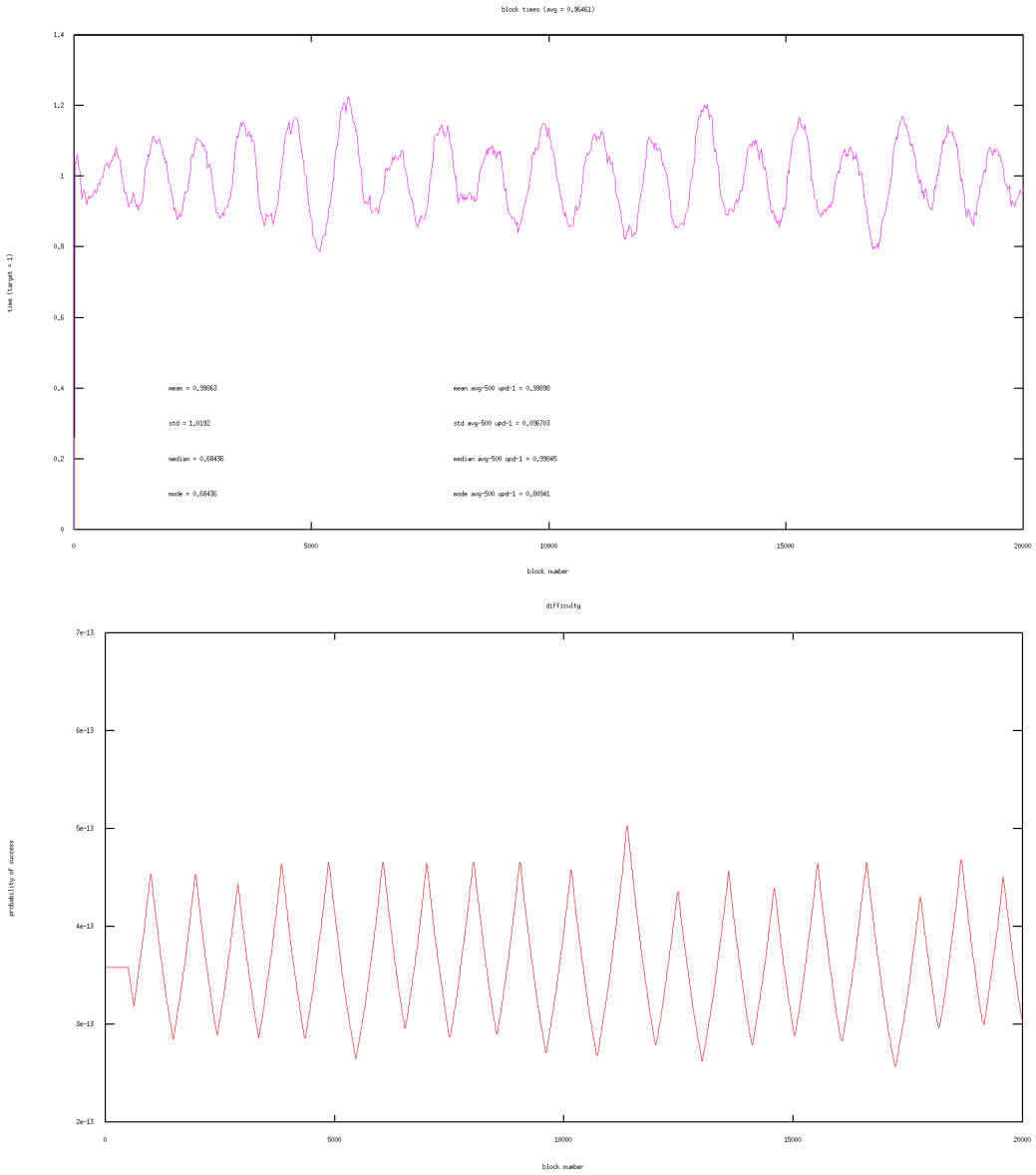
To maintain a target block-time addition rate of one block per minute (or any target time) the difficulty of the atomic mining operation will have to be updated over time. Vitalik's updated white paper suggests the following update strategy:

```
difficulty(0) := 2^36

update := floor( difficulty(n-1) / 1000 )

if (timestamp(n-1) - timestamp(n-500)) / 500 > 60
    difficulty(n) := difficulty(n-1) - update
else
    difficulty(n) := difficulty(n-1) + update
```

where `difficulty(n)` is the level for the current block and `timestamp(n)` returns the time of the $\mathtt{n}^{th}$ block in seconds. The genesis block is `n = 0` and all blocks before it are assumed to have the same `timestamp` as the genesis block.

The averaging effectively adds lag to the system, so the result of an update is not seen until many blocks later. In the interim, the algorithm continues to increase the difficulty. This leads to

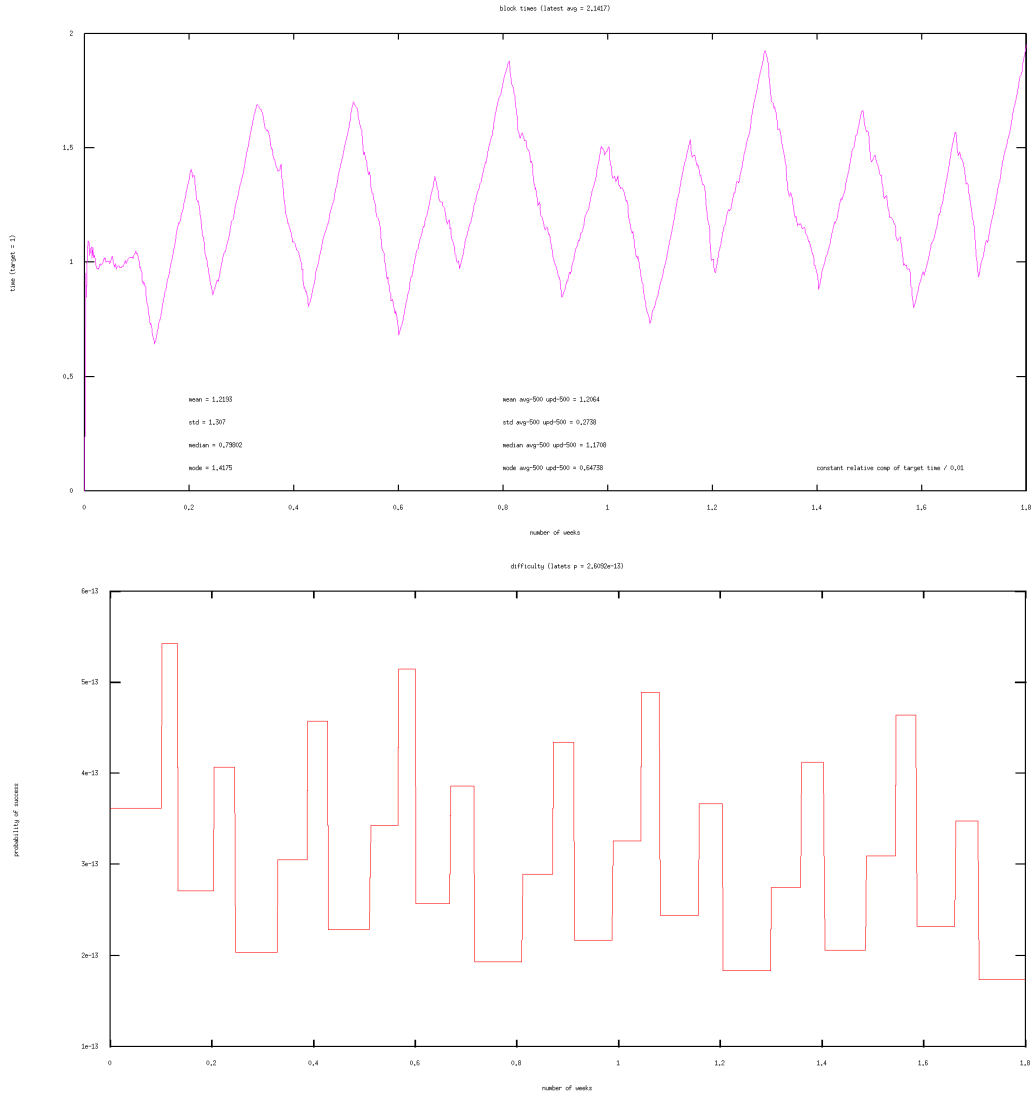
oscillations taking the 500-block average to ±20% when the underlying signal is steady-state (note that for the stability of the simulation, the probability of success is used instead of the difficulty):



This can be easily overcome by simply only updating every `M` blocks:

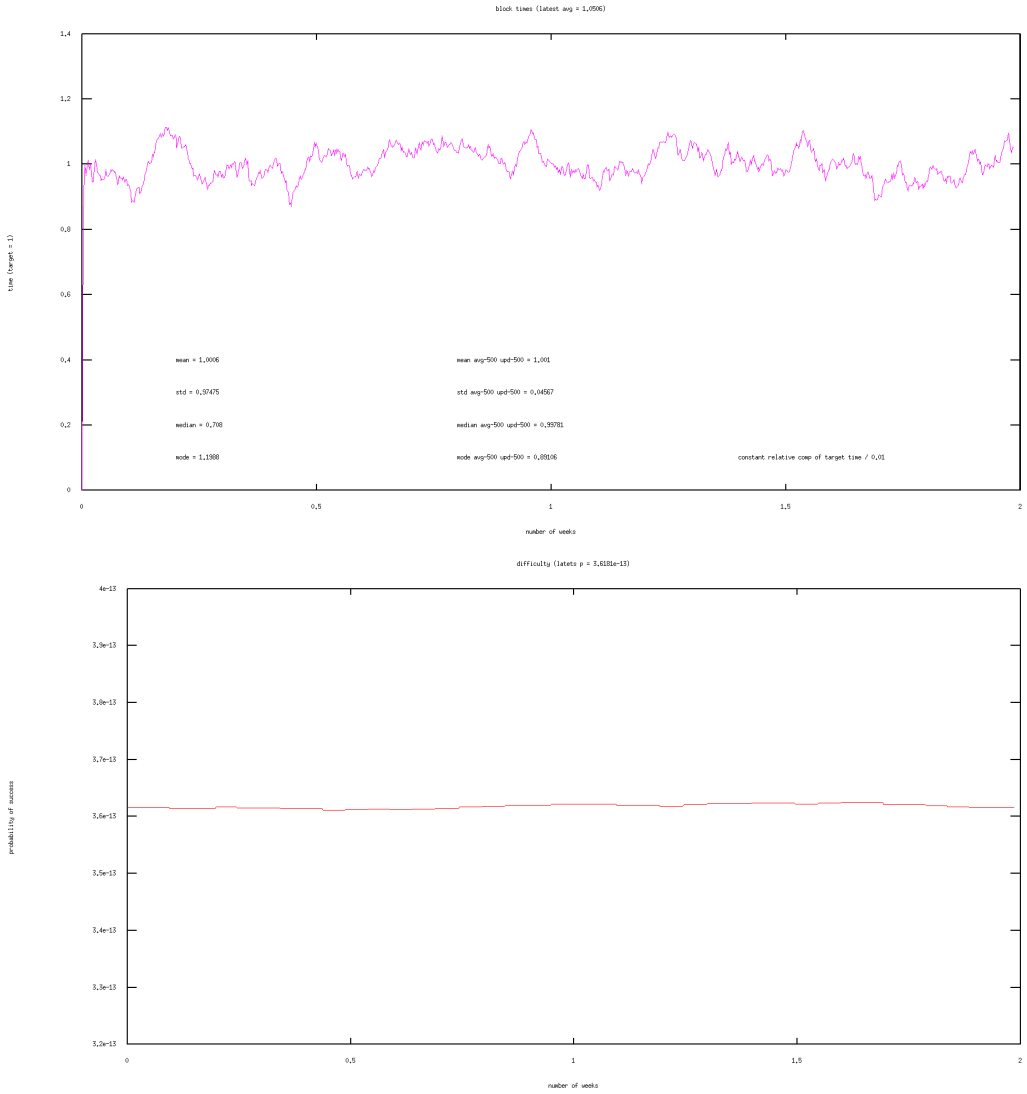
```
if mod(n, M) == 0
{
    update := floor( difficulty(n-1) * k )

    if (timestamp(n-1) - timestamp(n-500)) / 500 > 60
        difficulty(n) := difficulty(n-M) - update
    else
        difficulty(n) := difficulty(n-M) + update
}
else
{
    difficulty(n) := difficulty(n-1)
}
```

at the expense of responsiveness. However, the statistics of the mining nodes is not going to change dramatically in the order of 8 hours, so effective responsiveness is potentially adequate. The update `k` must be chosen to achieve a particular maximum update rate. The rate needs to be set high enough to be able to keep up with changes, but not so high as to create oscillatory behaviour when the mining power is not changing. Matching the first algorithm's ability to double in less than half a day, for example, is too much to update in one go:



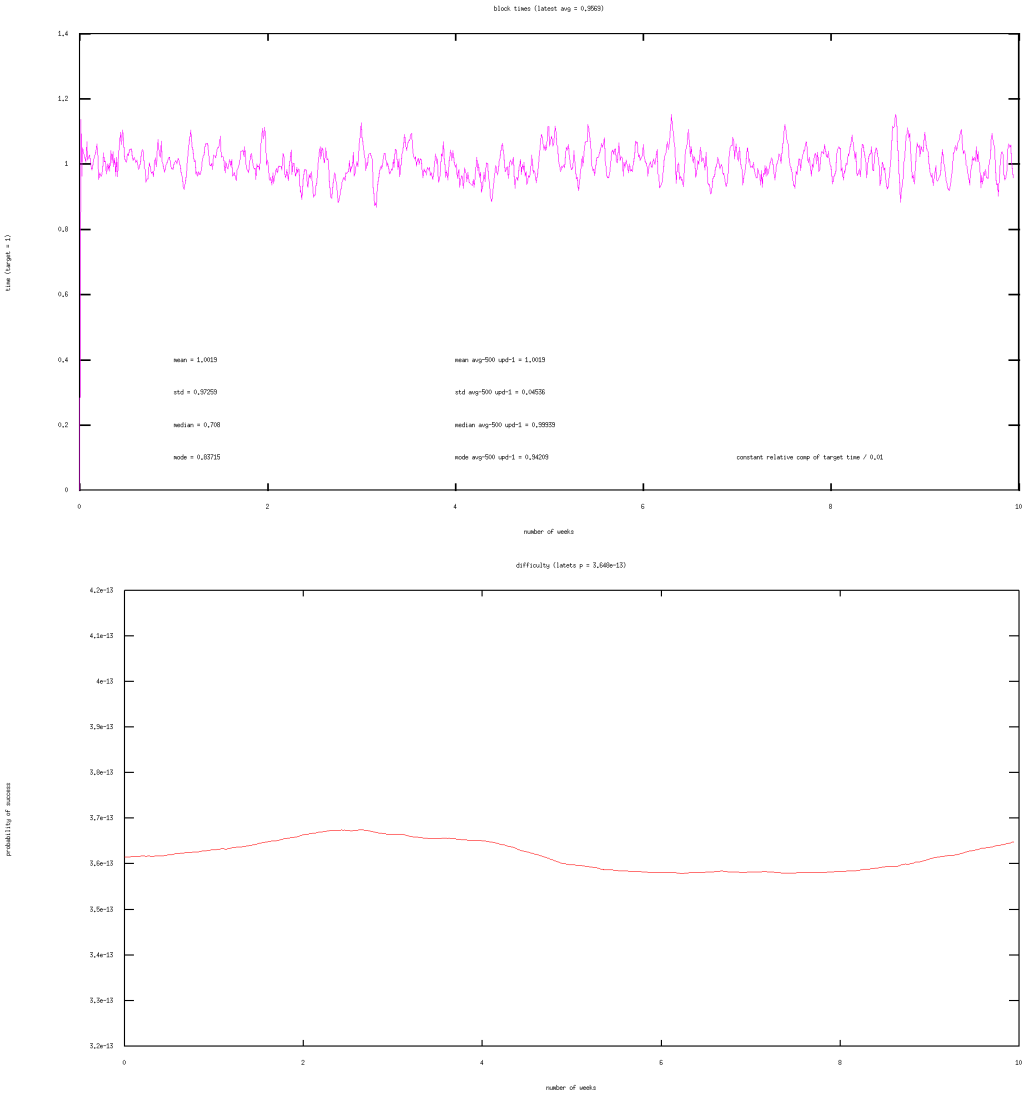
Another approach is to use a pseudo-proportional controller that updates by an amount proportional to the difference between the target and the actual (average) block-time, for example using the time difference in minutes divided by a constant (tuned here to 100):

```
if mod(n, M) == 0
{
    error := 1 - (timestamp(n-1) - timestamp(n-500)) / (500 * 60)

    difficulty(n) := difficulty(n-M) - error / 100
}
else
{
    difficulty(n) := difficulty(n-1)
}
```



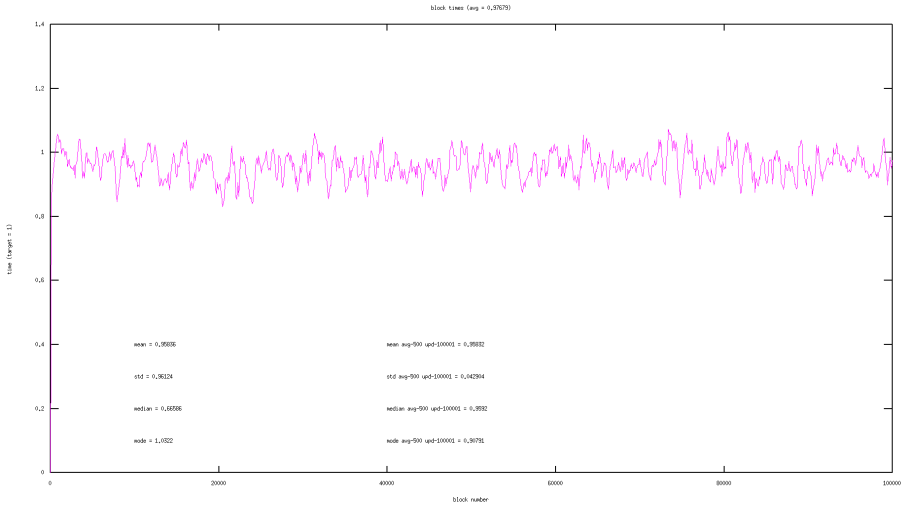
Tuning this to perform adequately over the full range of processor speeds is at best tricky and time-consuming and might not actually be possible. A more sophisticated controller, which is potentially more stable over a larger range of network sizes, is the PID controller (see wikipedia). The update algorithm then takes the form:

```
last_error := error
this_error := 1 - (timestamp(n-1) - timestamp(n-500)) / (500 * 60)

integral := integral + this_error

derivative := this_error - last_error

K_p = 10^7;
K_i = 1.2 * K_p / 2200;
K_d = 10^-3;

update := K_p * error + K_i * integral + K_d * derivative
difficulty(n) := difficulty(n-1) + update
```



The PID controller offers the potential for good stability in steady state as well fast response to changes in node processor profiles. Many more simulations would need to be run to get anything close to optimal parameters. Although far from optimal, initial tests suggest the above parameters are an adequate start.

For comparison, this is what steady state looks like with an single optimized difficulty setting:



Finally, the other alternative is to model the difficulty transfer function so we can guess the difficulty required to achieve the target block-time given the current distribution. This is bitcoin's strategy and we could simply copy that. I haven't done any simulations of this so cannot comment, but it seems to work for bitcoin fairly adequately.