Supporting Information
Accurate estimation of protein folding and unfolding times: Beyond Markov state models

Ernesto Suárez[1], Joshua L. Adelman[2] and Daniel M. Zuckerman*[1]

1. Department of Computational and Systems Biology, University of Pittsburgh
2. Department of Biological Sciences, University of Pittsburgh
*ddmmzz@pitt.edu

June 8, 2016

1 MFPTs dependence on τ and the number of states

After examining hundreds of acceptable models for each protein, more than 1500 in total, constructed as described in the main manuscript, we found that the dependence of the estimated Markovian MFPTs on the number of states is in general small as compared to the strong sensitivity to the lag-time (see Figures S1-S4). In the case of the Trp-Cage, for example (see Figure S1), around 100 models were selected so that the MFPTs measured *directly* are within the interval shown in Table S1. However, as shown in Figures S1-S4, in general the Markovian estimates lie in a wider range depending on the lag-time (τ) used. On the other hand, the dependency on the number of states is relatively small, especially the cases with shorter lag-times.

Table S1: Definitions of acceptable models. The models with MFPTs within the intervals shown around the values reported in the literature [1], are considered as good models, and otherwise are discarded.

Protein	MFPT Interval (μs)	
	Folding	Unfolding
Chignolin	0.1 - 3.0	0.5 - 8.0
Trp-cage	8.0 - 10	1.0 - 5.0
NTL9	5.0 - 30	75 - 250
Villin	1.0 - 5.0	0.2 - 3.0

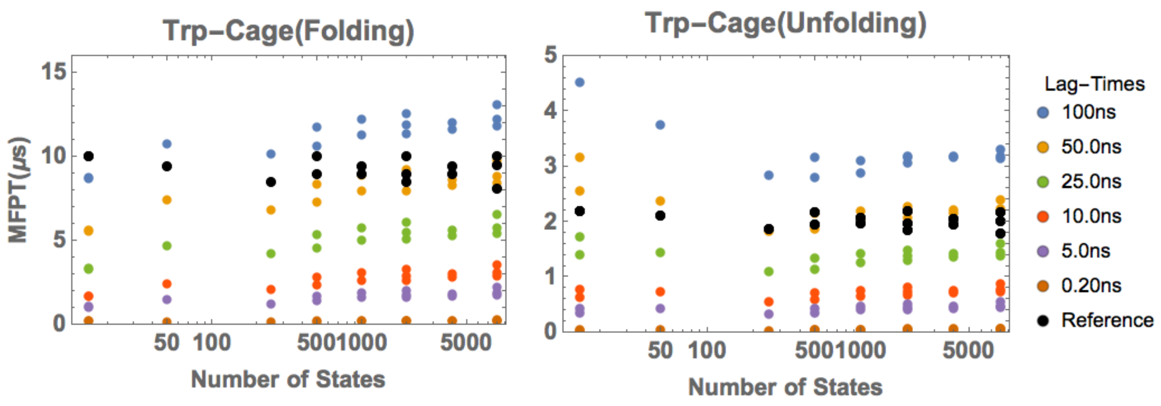

Figure S1: Trp-Cage folding and unfolding MFPTs versus number of states for different models constructed as described in the methods section of the main manuscript. For each model is also plotted the reference MFPT, which is the value obtained by directly averaging the first-passage times from the trajectory projected on the model states.

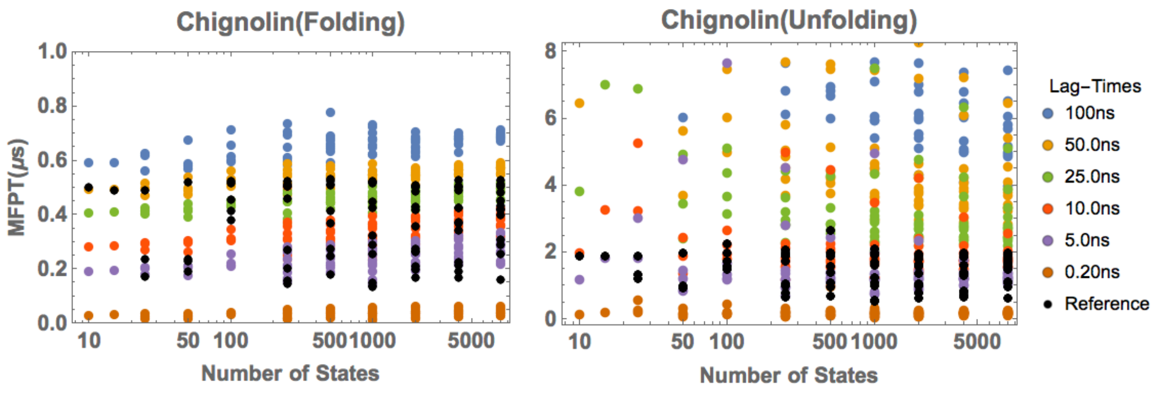

Figure S2: Chignolin folding and unfolding MFPTs versus number of states for different models constructed as described in the methods section of the main manuscript. For each model is also plotted the reference MFPT, which is the value obtained by directly averaging the first-passage times from the trajectory projected on the model states.

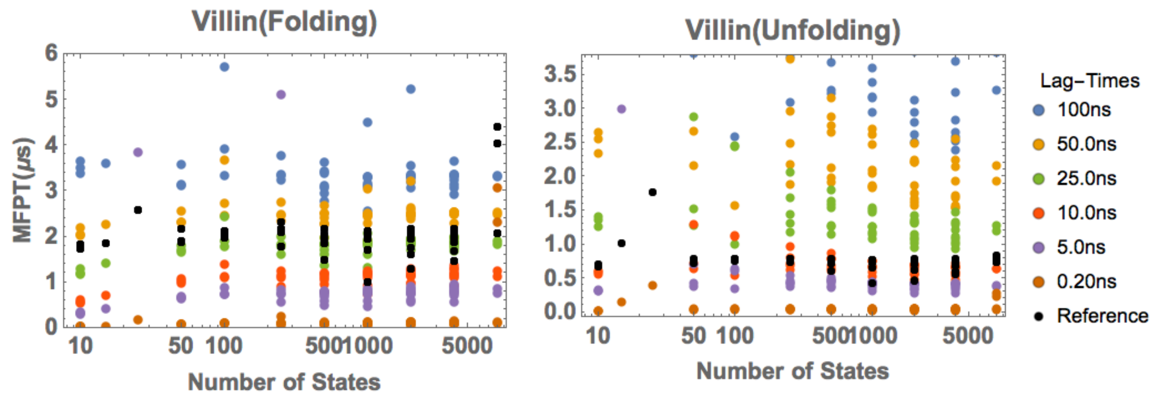

Figure S3: Villin folding and unfolding MFPTs versus number of states for different models constructed as described in the methods section of the main manuscript. For each model is also plotted the reference MFPT, which is the value obtained by directly averaging the first-passage times from the trajectory projected on the model states.

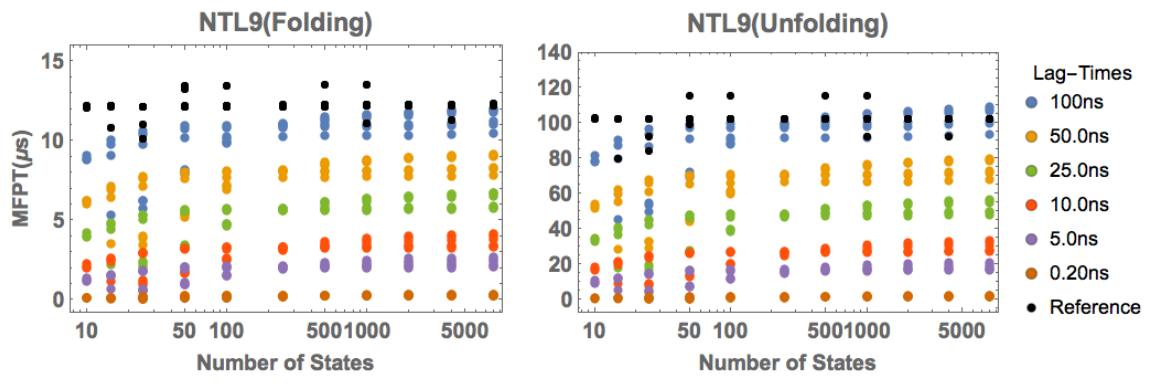

Figure S4: NTL9 folding and unfolding MFPTs versus number of states for different models constructed as described in the methods section of the main manuscript. For each model is also plotted the reference MFPT, which is the value obtained by directly averaging the first-passage times from the trajectory projected on the model states.

One expects the estimated MFPTs will grow monotonically with the lag-time only due to considering the discretization error. That is true for the MFPT estimated directly without the construction of any model: the longer lag excludes intermediate time points which may have exhibited events, hence increasing the MFPT. We should also expect the same behavior for any Markovian estimate, since as τ increases, the Markov model becomes a better approximation *for the given lag-time.* In other words, the results from the Markov model will be closer to the direct estimation inheriting the same discretization error.

Figures S5 and S6 show the dependence of the MFPTs on the lag-time in our models, where the macrostates (Folded and Unfolded) are defined as described above. The horizontal green dashed line labeled (MFPT - True Value) is the MFPT measured directly from all trajectory data with highest time resolution (0.2 ns), and it is our best estimate. The confidence intervals with 95% of confidence are shown as green horizontal stripes.

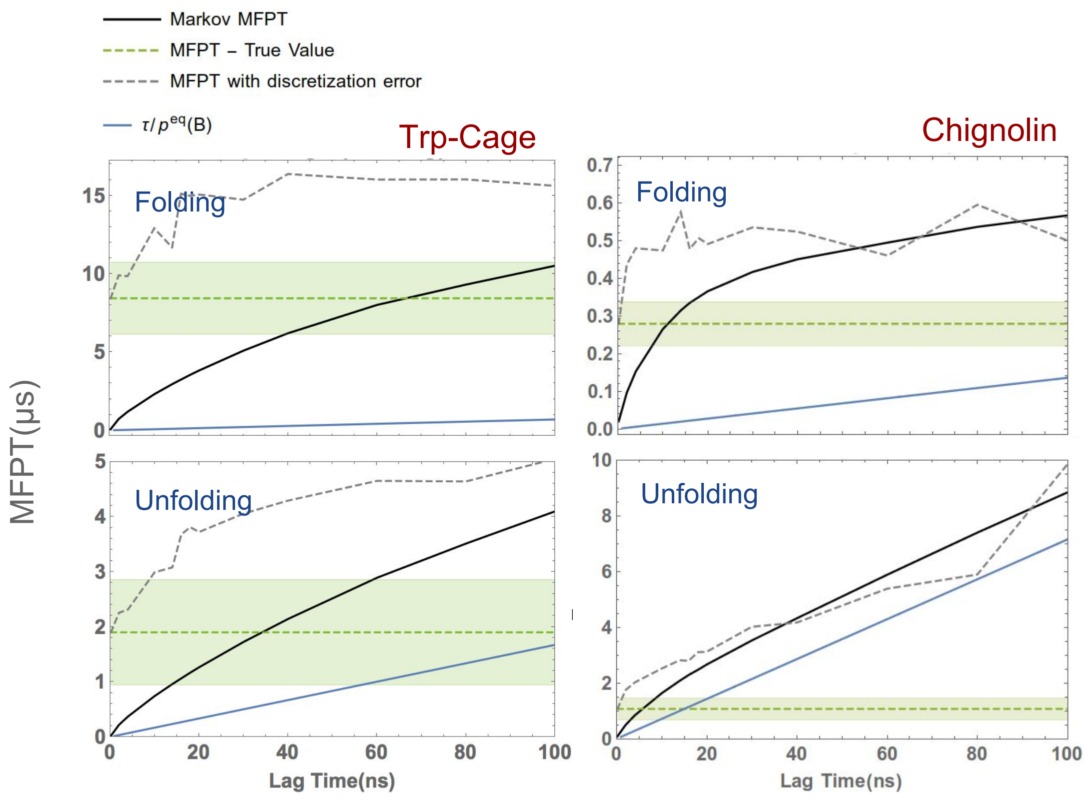

Figure S5: MFPTs versus lag-time in Trp-cage and Chignolin

What we called "MFPT with discretization error" is the MFPT measured directly from the trajectory for the given lag-time. Of course the "MFPT - True Value" also has discretization error, but we can not examine the trajectory at higher time resolution so we will consider this value as reference. As noted in the main paper, the estimate $\tau/p^{\mathrm{eq}}(B)$ corresponds to the non-dynamic estimate expected in the limit long lag times τ, where B denotes the target state, whether folded or unfolded.

As expected, the Markov MFPT estimates tend to approach the MFPT with discretization error, and often significantly exceed the 'true value' with the smallest discretization error. The Markov data, notably, do not appear to exhibit any inflections which might suggest larger lag times which could be justified in the absence of the long-trajectory analysis.

2 Software and algorithms used for calculating MFPT estimates

2.1 Markov State Modeling

There is more than one way to compute the MFPT in a Markov mode. Here the MFPT is obtained from the flux entering the target state [2] via

$$\mathrm{MFPT}(A \to B) = \frac{p(\alpha)}{\mathrm{Flux}(A \to B|\alpha)}, \tag{1}$$

where $\mathrm{Flux}(A \to B)$ is the average probability arriving, per unit time, at the target state B in the α steady state (trajectories that where last in A). Finally, $p(\alpha)$ is the total probability in the α steady state.

The numerical value of $p(\alpha)$ and in general the μ-labeled probabilities in each bin p_i^μ in a Markov model are obtained by imposing the steady state condition

$$\mathcal{K}^T p^\mu = p^\mu, \tag{2}$$

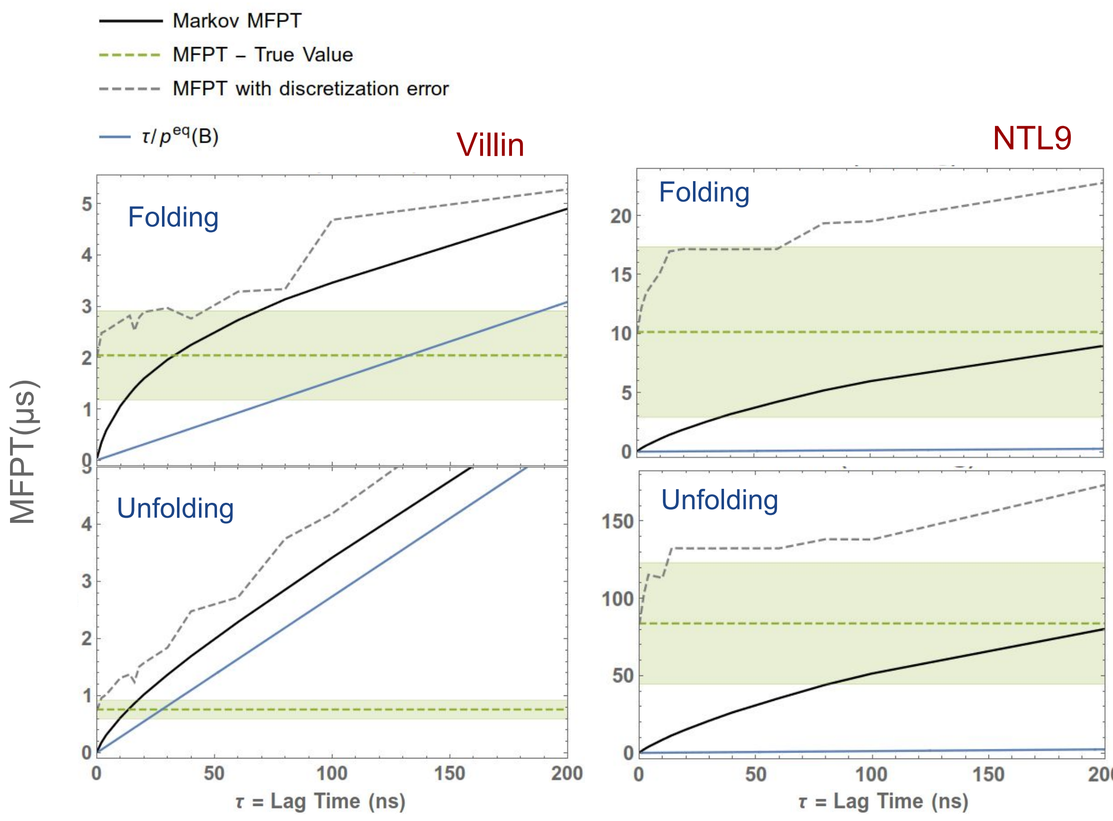

Figure S6: MFPTs versus lag-time in Villin and NTL9

where $\mathcal{K}^T$ is the $2N \times 2N$ "colored" matrix [8], where the transition probabilities are chosen to be history independent (i.e. $k_{ij}^{\mu\nu} = k_{ij}$)

Then, the probability flux entering B is computed as

$$\text{Flux}(A \to B|\alpha) = \sum_{i \notin B,\, j \in B} p_i^\alpha k_{ij}^{\alpha\beta} = \sum_{i \notin B,\, j \in B} p_i^\alpha k_{ij}. \tag{3}$$

In practice, the indexes i and j go from 0 to $2N-1$ in our $2N \times 2N$ matrix, where we store in the even and odd rows the $\alpha \to \nu$ and $\beta \to \nu$ rates respectively. Similarly, in the even and odd columns we have the $\mu \to \alpha$ and $\mu \to \beta$ rates respectively.

In python code, we can evaluate Eq. 1 as follows:

```
#We are going to consider here we that already have the
#Markovian transition matrix (MarkovTM)
import numpy as np

#Creating the pseudo-colored transition matrix
K  = np.zeros((2*N,2*N))
for i in xrange(2*N):
      for j in  xrange(2*N):
            K[i,j] = MarkovTM[int(i/2),int(j/2)]

#There are forbidden transitions, so some elements of K
# must be zero by construction.
for i in xrange(N):
      for j in xrange(N):
               # the variables macroStateA and macroStateB are lists that
            # contain the states that define them.
            if (i in macroStateB) or (j in macroStateB):
                 K[2*i,2*j] = 0.0
            if (i in macroStateA) or (j in macroStateA):
                 K[2*i+1,2*j+1] = 0.0
            if (not (j in macroStateA)) or (i in macroStateA):
                 K[2*i+1,2*j] = 0.0
            if (not (j in macroStateB)) or (i in macroStateB):
                 K[2*i,2*j+1] = 0.0

#Obtaining the SS labeled populations from the 2N x 2N matrix K
Pr=solveSteadyState(K) #Solving K^T Pr = Pr

#Total population of the alpha steady state.
popAlpha = 0.0
for i in xrange(0,2*N,2):
        popAlpha += Pr[i]

#Computing the Flux from A to B
fluxAB = 0.0
for i in xrange(0,2*N,2):
     for j in xrange(2*N):
         # int(j/2) extracts the physical (uncolored) state index
         if int(j/2) in macroStateB:
               fluxAB += Pr[i]*K[i,j]

#Computing the MFPT from A to B
MFPTAB = popAlpha/fluxAB
```

In the previous code N is the number of physical states, K is $\mathcal{K}$ and Pr is p^{μ}, the steady state solution of Eq. 2. The macro states A and B are denoted by macroStateA and macroStateB respectively and are essentially python lists that contain the indexes of the states that define them.

2.2 Markov + color

Here, the MFPT is calculated in the same way as the previous section (Eqs. 1 and 3), except that we use a modified version of $\mathcal{K}$ that has partial color information in it ($\tilde{\mathcal{K}}$). We emphasize that $\mathcal{K}$ has color history information built in, even though it is manipulated like a Markovian matrix.

In order to compute $\tilde{\mathcal{K}}$ we first compute a non-Markovian $2N \times 2N$ count matrix $\mathcal{C}$, in which the color information is used when available. The following code can be used to compute $\mathcal{C}$, from each transition. Note that we use the Markovian solution (p^{μ}, Pr in the code) of Eq. 2 to infer the color when it is not available.

```
import numpy as np
countMatrix = np.zeros((2*N,2*N)) #Initiating as zero all the elements
prevColor = "U" #previous color is set as unknown

for i in xrange(numSnapShots): #numSnapShots is the number of time points
                                                 #we have in the simulation
      currentState = stateIndex[i] #funtion that gets the i-snapshot

      # Macro state determination.
      # The variables macroStateA and macroStateB are lists that
      # contain the states that define them.
      if currentState in macroStateA:
            macroState = "A"
      elif currentState in macroStateB:
            macroState = "B"
      else:
            macroState = "U"

      #Color determination
      if macroState == "A":
            color = "A"
      elif macroState == "B":
            color = "B"
      else:
            color = prevColor

      #Constructing the Markov + Color count matrix "countMatrix"
      if i>1:
            if prevColor == "A" and color == "B":
                  # alpha -> beta transition
                  countMatrix[2*prevState, 2*currentState+1] += 1.0
            elif prevColor == "B" and color == "A":
                  # beta -> alpha transition
                  countMatrix[2*prevState+1, 2*currentState] += 1.0
            elif prevColor == "A" and color == "A":
                  # alpha -> alpha transition
                  countMatrix[2*prevState, 2*currentState] += 1.0
            elif prevColor == "B" and color == "B":
                  # beta -> beta transition
                  countMatrix[2*prevState+1, 2*currentState+1] += 1.0
            elif prevColor == "U" and color == "B":
                  # unknown-color -> beta transition
                  countMatrix[2*prevState, 2*currentState+1]   += \
                  Pr[2*prevState] / ( Pr[2*prevState] + Pr[2*prevState+1] )
                  countMatrix[2*prevState+1, 2*currentState+1] += \
                  Pr[2*prevState+1] / ( Pr[2*prevState] + Pr[2*prevState+1] )
            elif prevColor == "U" and color == "A":
                  # unknown-color -> alpha transition
                  countMatrix[2*prevState, 2*currentState]   += \
                  Pr[2*prevState]  / (Pr[2*prevState] + Pr[2*prevState+1])
                  countMatrix[2*prevState+1, 2*currentState] += \
                  Pr[2*prevState+1]/ (Pr[2*prevState] + Pr[2*prevState+1])
            elif prevColor == "U" and color == "U":
                  # unknown-color -> unknow-color transition
                  tempSum =2 * (Pr[2*prevState] + Pr[2*prevState+1])
                  countMatrix[2*prevState,2*currentState+1]   += \
                  Pr[2*prevState]/tempSum
                  countMatrix[2*prevState+1, 2*currentState+1] += \
                  Pr[2*prevState+1]/tempSum
                  countMatrix[2*prevState,2*currentState]      += \
                  Pr[2*prevState]/tempSum
                  countMatrix[2*prevState+1, 2*currentState]   += \
                  Pr[2*prevState+1]/tempSum
            else:
                  print "Error while constructing the Markov + Color matrix"
                  sys.exit()
      prevColor = color
      prevState = currentState
```

After the count matrix is built we just need to normalize each row to obtain the Markov + Color rate matrix $\tilde{\mathcal{K}}$, i.e., divide the elements of the row by the total sum of the counts in the given row. Then we can compute the fluxes and the MFPT from $\tilde{\mathcal{K}}$ the same way we shown in the previous section from $\mathcal{K}$.

2.3 2nd-order Markov

The 2nd-order Markov model extends the regular Markov model by including in the analysis another time-point in the history at time $t - n\tau$, where $n = 1, 2, ...$ (see the main manuscript). Then, all the properties can be estimated from a numerical simulation of the model from the transition probabilities. In order to initiate the simulation, we take randomly one of trajectory segments used to construct the matrix and extract a small fragment to be used as a "seed". The number of snapshots in the seed should be greater than the history length by at least one snapshot. In every system, the number of iterations used for the simulation was 10 times the length in snapshots of the original simulation.

2.4 2nd-order Markov + color

In the 2nd-order Markov+Color estimator, we want to use color information when it is available, but the strategy we follow is different from (first-order) Markov+Color. Here, for each transition we examine the trajectory back in time up to a given history length ($n\tau$). If a macro state A or B is found, the third index m (see Eq.5 in the main manuscript) takes the label of the first bin/state which defines the macro state - which is just a convenient way to encode the α and β labels. The following code shows we assign the third index to construct the matrix.

```
for j in xrange(histLength+1,numSnapShots):
        secondIndex = stateIndex[j-1] #the state from which the transition starts
    for m in xrange(j-2,j-2-histLength,-1):
        thirdIndex = stateIndex[m] #the time-point we take more backward in time
        if (secondIndex in macroStateA) or (thirdIndex in macroStateA):
                thirdIndex = macroStateA[0]
            break
        elif f (secondIndex in macroStateB) or (thirdIndex in macroStateB):
                thirdIndex = macroStateB[0]
            break
```

Again, once we have the transition matrix, all the properties can be estimated from a numerical simulation of the model from the transition probabilities as described in the previous section.

3 Non-Markovian estimates vs the lag-time

Any model has to "learn" from the real dynamics, or at least from the evolution of the selected collective variables at a time resolution no higher than the one used to store the trajectories. Even a carefully designed model cannot make unambiguous predictions about the dynamics at a higher time resolution. That is a limitation that any model will have.

Figure S7 illustrates with an example how our estimates "Markov + Color" and "2nd-Markov + Color" lie within the confidence interval of the direct method (Long MD) for a history length of 200ns. However, all the methods, including the direct, are influenced by the discretization error and they all grow artificially with the lag time.

Our Non-Markovian models are not an exception and they will inherit the same discretization error of the available data. However, in this case we can reduce the lag-time as much as we can and overcome that limitation. On the other hand, in a regular Markov model that is not possible since the model has to be as Markovian as possible.

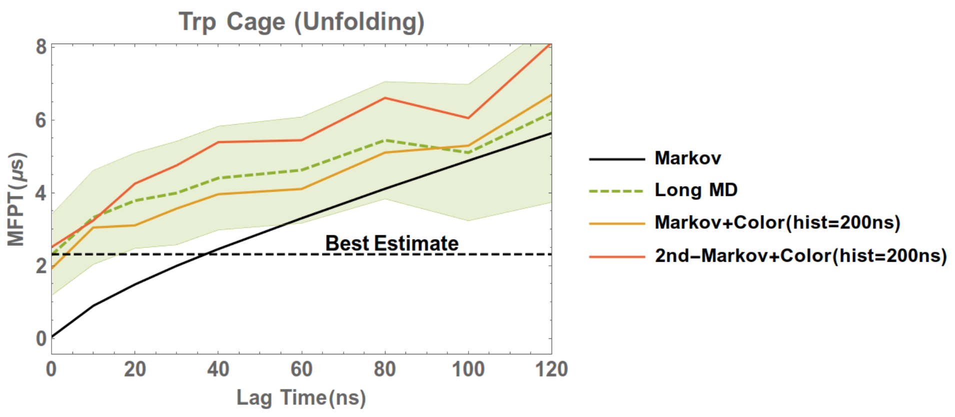

Figure S7: Unfolding MFPT estimates vs lag-time in Trp-cage. The horizontal black dashed line is the direct estimation at the highest time resolution (0.2ns). The green region represents a the confidence interval, with 95% of confidence, of the direct estimation (Long MD). The best estimate is the direct estimation for the shortest lag-time.

References

[1] Kresten Lindorff-Larsen, Stefano Piana, Ron O Dror, and David E Shaw. How fast-folding proteins fold. *Science*, 334(6055):517–520, Oct 2011.

[2] Ernesto Suarez, Steven Lettieri, Mattew C. Zwier, Carsen A. Stringer, Sundar Raman Subramanian, Lillian T. Chong, and Daniel M. Zuckerman. Simultaneous computation of dynamical and equilibrium information using a weighted ensemble of trajectories. *J Chem Theory Comput*, 10(7):2658–2667, 2014.

[3] Guillermo Pérez-Hernández, Fabian Paul, Toni Giorgino, Gianni De Fabritiis, and Frank Noé. Identification of slow molecular order parameters for markov model construction. *J Chem Phys*, 139(1):015102, Jul 2013.

[4] Christian R Schwantes and Vijay S Pande. Improvements in markov state model construction reveal many non-native interactions in the folding of ntl9. *J Chem Theory Comput*, 9(4):2000–2009, Apr 2013.

[5] Kyle A Beauchamp, Gregory R Bowman, Thomas J Lane, Lutz Maibaum, Imran S Haque, and Vijay S Pande. MSMBuilder2: Modeling conformational dynamics at the picosecond to millisecond scale. *J Chem Theory Comput*, 7(10):3412–3419, Oct 2011.

[6] Robert T McGibbon, Kyle A Beauchamp, Christian R Schwantes, Lee-Ping Wang, Carlos X Hernández, Matthew P Harrigan, Thomas J Lane, Jason M Swails, and Vijay S Pande. MDTraj: a modern, open library for the analysis of molecular dynamics trajectories,. *Biophys J*, 109:1528–1532, 2015.

[7] Stefano Piana, Kresten Lindorff-Larsen, and David E Shaw. How robust are protein folding simulations with respect to force field parameterization? *Biophys J*, 100(9):L47–9, May 2011.

[8] Ernesto Suarez, Adam J. Pratt, Lillian T. Chong, and Daniel M. Zuckerman. Estimating first-passage time distributions from weighted ensemble simulations and non-markovian analyses. (in press) doi:10.1002/pro.2738. *Protein Science*, 25:67–78, 2015.