

# Prediction of CPU Idle-Busy Activity Pattern

Qian Diao

Justin Song

Intel Corp.

[qian.diao@intel.com](mailto:qian.diao@intel.com)

[justin.j.song@intel.com](mailto:justin.j.song@intel.com)

## Abstract

Real-world workloads rarely saturate multi-core processor. CPU C-states can be used to reduce power consumption during processor idle time. The key unsolved problem is: when and how to use which C-state. We propose a machine learning prediction method and usage model. We evaluate this model with idle traces collected on dual-core and quad-core processor, and find this method can well predict CPU's activity pattern at the error level not exceeding 4%. Compared with existing OS C-state policy, it results in 12% additional CPU power saving and 2% performance improvement. In industry, 12% power saving for any processor is very significant improvement. SPECWeb (which we used consists of 3 different benchmarks -- We consistently see double-digit power saving) is representative "front-end" server workload -- it takes >60% DP server market segment share.

## 1. Introduction

Since real-world workloads rarely saturate multi-core processor based servers, how to save power when CPU has nothing to do becomes increasingly important. ACPI defines variety of C-states [1], such as C0, C1, C2 and C3. C0 is for CPU being active (executing and retiring instructions). C1, C2 and C3 are all idle states (CPU execution stops). C2's power is lower than C1 and C3's power is lower than C2. Typically C2's entry and exit latency is higher than C1, and C3's higher than C2. Figure 1, extracted from ACPI spec [1], gives an example how C1, C2 and C3 power and latency look like. For example: C1's latency is 20us and power is 1000mW. Each processor vendor may have its own C-state optimizations, and map the optimizations to ACPI C-state, but in general the power and latency rule holds. In most modern OS, a core within multi-core package is mapped to a single logical CPU. An OSPM (OS Power Management) policy is responsible

for choosing a C-state when a core is about to idle. A break-event (typically an interrupt routed to idle core) brings the idle CPU to C0 state. If OSPM selects C3 and coming idle duration is very short (e.g. equal to C3 entry latency), the CPU will return to C0 once it completes entering C3, resulting in big latency to respond to break-event. Use latency numbers in figure 1, the added latency is  $60/2=30\mu s$ . On the other hand, if OSPM selects C1 and coming idle duration is very long (e.g. several milliseconds), the CPU has to stay in high-power C1 and waste power saving opportunities if it had chosen C3. Again use power numbers in figure 1, 500mW power saving opportunity is wasted.

Name (_CST, Package())	Latency in micro-second	Power in milli-watt
{		
3, // There are three C-states defined here with three semantics		
C1 → Package() {ResourceTemplate() {Register(FFixedHW, 0, 0, 0)}, 1, 20, 1000}	20	1000
C2 → Package() {ResourceTemplate() {Register(SystemIO, 8, 0, 0x161)}, 2, 40, 750}	40	750
C3 → Package() {ResourceTemplate() {Register(SystemIO, 8, 0, 0x162)}, 3, 60, 500}	60	500
}}		

Figure 1. Processor C-state power and latency example (data source ACPI spec)

So prediction for CPU's future activity is the key to use right C-state. Without a good prediction method, any potentially good C-state technologies cannot really benefit multi-core CPU's performance/watt. However, all today's methods are basically reactive. For example, latest Linux naively use CPU utilization and past idle duration on a CPU to speculate this CPU's next idle duration and accordingly choose desired C-state [2]. It overlooks random distribution of workloads' activity pattern. Furthermore, although a CPU can enter a desired C-state irrespective of other CPUs' states, CPU vendors usually do hardware based power

optimizations for a whole package – if all cores within the same package have entered a non-zero C-state, package voltage may be lowered and some units in the package may be clockgated or even power gated. Today’s C-state policy doesn’t take the neighboring CPUs’ C-state into consideration and hence hardware might not be able to take advantage of package power optimization.

Now that simple reactive method doesn’t work, we need machine learning methods for data analysis and model establishment. Dynamic Bayesian Networks (DBNs) provide a general framework to solve the inference tasks in computer vision [8][10], speech recognition[9], pattern recognition[10], etc. They are directed graphical models of stochastic processes. They generalize hidden Markov models (HMMs) and linear dynamical systems (LDSs), also called Kalman Filter Models (KFM), by representing the hidden (and observed) state in terms of state variables, which can have complex interdependencies. DBNs are attractive for the applications because they combine a natural mechanism for expressing domain knowledge with efficient algorithms for learning and inference. It provides two distinct benefits: Flexible modeling choices and schemes that can be tailored to fit the complexity of the applications, all of which can be conceptualized in a single framework with an intuitively-appealing graph notation. Second, there exist many effective and efficient inference and learning algorithms for DBNs that can be applied to different data and tasks. In this paper, we present a DBN and a corresponding inference method to predict CPU activity pattern. The DBN is a graphical model representation of KFM, which is also called linear dynamic system (LDS). The prediction algorithm is based on fixed-lag smoothing method [3].

Section 2 is about usage model of this prediction model. Section 3 is about CPU activity pattern definition and prediction framework. Section 4 describes the machine learning algorithm we use, as well as algorithm simplification. Section 5 is experimentation prediction results. In section 6, we evaluate how this prediction method benefit CPU power saving and performance improvement. And lastly section 7 summarizes this paper.

## 2. Usage model

As shown in figure 2, the proposed prediction model can use either in-band, e.g. in conjunction with an OS C-state policy, or out-of-band, e.g., on a co-processor or a mother-board firmware component. An activity monitor continuously gathers all cores’ activity

data, and sends to prediction model. Prediction model output next time slice’s activity pattern. An activity controller then uses this output to direct C-state selection. Possible usage of prediction output includes:

- If future time slice activity pattern has high percentage of aggregated idle, we can direct C-state policy to be prepared to use aggressive core C-state. This is because a single core will be idle long enough, and aggressive core C-state won’t hurt its performance.
- Another way to take advantage of high percentage of aggregated partial idle is to tell OS to change timing of tasks on each core, so that single cores’ idle can be overlapped. This way partial idle is transitioned to all cores idle, resulting in increased opportunity of using package-idle low power optimization.
- On a MP platform which has more than one physical processor packages, if each package’s future time slice has high percentage of all-core-idle, we can reschedule timing of tasks on each physical package to make sure both packages’ all-core-idle occurrences will be overlapping, so that platform level low power optimization like memory clockgating and low power link state can be used.
- Usually deep package C-state exit latency is long. Prediction for future multiple time slices can be used to direct package to proactively exit low package C-state to avoid performance penalty.

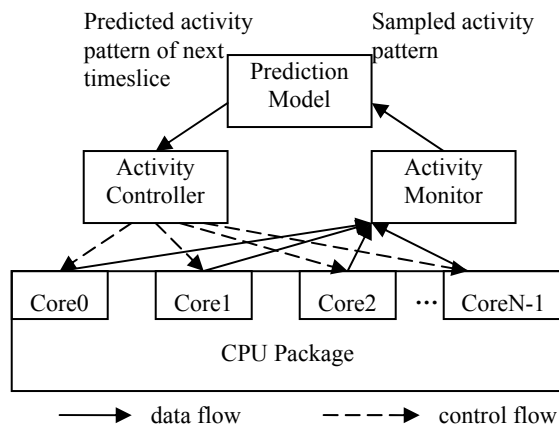


Figure 2. Usage model of CPU activity pattern prediction

## 3. CPU package idle pattern

### 3.1 CPU package activity state

We define activity state of multi-core CPU package as following:

$$\text{“CPU package activity state” } I = (S_0, S_1, \dots, S_{N-1})$$

where  $N$  is number of cores inside the same package. For dual-core processor,  $N=2$ ; for quad-core processor,  $N=4$ .  $S$  means state of a single core.  $S=0$  corresponds to C-state “0”, meaning core is busy;  $S=1$  corresponds to any core C-state “1” or deeper, meaning core is idle. Obviously, total number of CPU package idle states is  $2^N$ . We use  $I_i$  ( $0 \leq i \leq 2^N - 1$ ) to denote a package activity state. For example, for quad-core processor,  $I_{13}$  means core0, core1, and core3 are idle, and core2 is busy.

The  $2^N$  processor package activity states can be classified into 3 important categories:

- All cores are idle (1,1,..., 1). When all cores are idle, the whole processor package’s low power optimization can be used.
- All cores are busy (0,0,...,0). At this state, there is no possibility to put a core or package to a non-zero C-state.
- Partially idle state, with at least one core idle and at least one core busy. Although at this time it is impossible to use package power optimization, some deeper core C-state may be used on idle core(s). In later sections, for simplicity, we sum all partial idle states into one aggregated partial idle state.

Tick	0	100	1000	5000	5500	6500	7200	7800	9000	9500	10000
Core0	Idle	Busy	Busy	Busy	Busy	Busy	Busy	Idle	Idle	Idle	Idle
Core1	Idle	Idle	Busy	Busy	Busy	Idle	Idle	Idle	Idle	Idle	Idle
Core2	Idle	Idle	Idle	Busy	Busy	Busy	Idle	Idle	Idle	Busy	Busy
Core3	Idle	Idle	Idle	Idle	Busy	Busy	Busy	Busy	Idle	Idle	Busy
state	1111	0111	0011	0001	0000	0100	0110	1110	1111	1101	1100

Figure 3. Quad-core processor package activity state changes over time

Figure 3 gives an example how quad-core processor package activity state changes over time. The first row is CPU reference core clockticks. On 3GHz quad-core processor, one tick is 0.33 nano-second. The second through the fifth rows are each core’s busy/idle state. The reason why an idle core becomes busy is because some new task is scheduled on this core, or previously un-met conditions are met; and the reason why a busy idle becomes idle is because it has completed given task and OS scheduler doesn’t have no more tasks for

him, or its continuation of execution of current task needs some conditions which cannot be met. And the last row is the CPU package activity state.

### 3.2 CPU package activity pattern

We define multi-core CPU package activity pattern during time slice  $t$  as following:

“CPU package activity pattern”  $P_t = (\%R_{I_0}, \%R_{I_1}, \dots, \%R_{I_{2^N-1}})$ , Where  $t$  is time slice index,  $N$  is number of cores inside the same package,  $I_i$  ( $0 \leq i \leq 2^N - 1$ ) is an activity state,  $R_{I_i}$  is residency (how many reference clock ticks) of  $I_i$  during a time slice, and  $\%R_{I_i}$  is percentage that  $R_{I_i}$  occupies time slice  $t$ . Obviously the sum of  $\%R_{I_i}$  is

1. The length of time slice  $t$  is programmable. In our experimentation, we measure and count all cores’ residency every 500 micro-seconds, so a time slice is 500us.

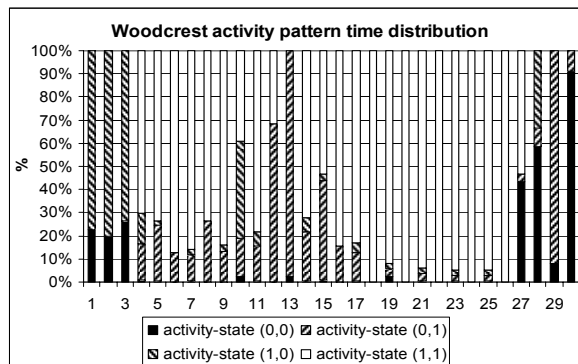


Figure 4. Dual-core processor package activity pattern distribution

An activity pattern roughly describes the core C-state and package power optimization power saving benefit’s upper and lower bound. For example, given dual-core processor (it has 4 activity states  $I_0, I_1, I_2, I_3$ ) activity pattern is (20%, 25%, 45%, 10%) at a time slice. This means all core busy’s total clockticks during that time slice is 20%; partial idle (core0 is busy core1 is idle, and core0 is idle and core 1 is busy)’s total clockticks is 25%+45%=70%; and all core idle’s total clockticks is 10%. So during that time slice, the maximum time to put the whole processor package in an package idle state cannot exceed 10%; opportunity to use core C-state cannot exceed 70%; and during

20% of that time slice, no core or package idle state can be used. Figure 4 gives an example of how CPU package activity pattern over time looks like. The workload is SPECWeb running on a two-way dual-core processor based platform. In this example, system load level (average CPU utilization) is roughly 50%.

## 4. Machine learning model

### 4.1 DBN model

In our machine learning model, the CPU activity pattern is considered the output of a discrete-time controlled process. It is a time series with multiple dimensions, which we will denote by  $y_{t_r} = (y_{1...t_r})$ . Our task is to predict values on the future time, given all the observations up to the present time. However, we are generally unsure about the future, and what we can do is to compute a best guess, as well as how confident we are of this guess, so we can hedge our bets appropriately. Hence we will try to compute a probability distribution over the possible future observations; we denote this by  $P(Y_{t+h} = y | y_{t_r})$ , where  $h > 0$  is the horizon, i.e., how far into the future we want to predict.

DBNs provide a good framework for time-series prediction. It generalizes the popular models, such as KFM and HMM, by allowing arbitrary probability distributions, not just (unimodal) linear-Gaussian. In addition, it outperforms some “classical” approaches, such as ARIMA, ARMAX, etc. (see e.g., [6]), neural networks and decision trees [7] by “better expressive power”. For example, it is much easier to incorporate prior knowledge into DBNs than the black-box models such as neural networks, which are notoriously hard to interpret, because DBNs has explicit probability definition based on Bayesian theorem and good network structure description base on graph theory. When the prior knowledge can not be expressed in terms of observable quantities, DBNs has more ways from both probability aspects, such as prior distribution for conditional probability distributions (CPDs), and graph theory, such as prior net, to incorporate the different knowledge.

As shown in figure 5, the DBN model we used is a state space model (SSM). The CPDs of the model can be defined as:

$$\begin{aligned} P(X_1 = x) &= N(x; x_{10}, V_{10}) \\ P(X_t = x_t | X_{t-1} = x_{t-1}) &= N(x_t; Ax_{t-1}, Q) \\ P(Y_t = y | X_t = x) &= N(y; Cx, R) \end{aligned} \quad (1)$$

where  $x_{10}, V_{10}$  are the initial mean and variance,  $A$  is the transition matrix,  $Q$  is the system covariance,  $R$  is the observation covariance, and  $C$  is the observation matrix. The transition and observation functions are the same for all time and the model is said to be time-invariant or homogeneous. (Without this assumption, we could not model infinitely long sequences.)

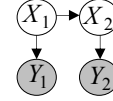


Figure 5. DBN model for CPU idle prediction

The model assumes the time series is a first-order Markov process, i.e.  $P(X_t | X_{t-1}) = P(X_t | X_{t-1})$ , so the subscripts denote two time slices since the current state depends only on the previous state. Oval nodes are continuous, shaded nodes are observed, and clear nodes are hidden. All the conditional probability distributions (CPDs) are linear-Gaussian.

Since we start our modeling from a simple assumption, in which all the variables in the DBN model are linear Gaussian and the dynamic is unimodal, the model in figure 5 become a DBN representation of KFM. It is also called linear dynamic system (LDS). It models a partially observed stochastic process with linear dynamics and linear observations, both subject to Gaussian noise. The linear stochastic difference equation in KFM is:

$$\begin{aligned} x(t) &= Ax(t-1) + w(t-1) \\ p(w) &\sim N(0, Q) \quad x(0) \sim N(x_{10}, V_{10}) \end{aligned} \quad (2)$$

and the measurement equation is:

$$y(t) = Cx(t) + V(t) \quad p(v) \sim N(0, R) \quad (3)$$

The  $n \times n$  matrix  $A$  in the difference equation (3) relates the state at the previous  $t-1$  time step to the state at the current step  $t$ , in the absence of either a driving function or process noise. Here  $n$  is the number of hidden states. In our particular task,  $n$  is the number of possible CPU activity status. Note that in practice  $A$  might change with each time step, but here we assume it is constant and the model is homogeneous. The  $m \times n$  matrix  $C$  in the measurement equation (3) relates the state to the measurement  $y(t)$ . Here  $m$  is the number of observed states. In our particular task,  $m = n$  is the number of possible CPU activity status. In practice,  $C$  might change with each time step or measurement, but here we also assume it is constant.

## 4.2 Prediction algorithm

As we mentioned before, given the sequence of observed values  $(y_1, \dots, y_t)$ , to predict the new observation value is to compute  $P(Y_{t+h} = y | y_{1:t})$  for some horizon  $h > 0$  into the future. Equation (4) is the computation of a prediction about the future observations by marginalizing out the prediction of the future hidden state.

$$\begin{aligned} & P(Y_{t+h} = y | y_{1:t}) \\ &= \sum_x P(Y_{t+h} = y | X_{t+h} = x) P(X_{t+h} = x | y_{1:t}) \end{aligned} \quad (4)$$

In the right part of the equation (4), our computation of  $P(X_{t+h} = x | y_{1:t})$  is based on the fixed-lag smoothing, i.e.  $P(X_{t-L} = x | y_{1:t})$ ,  $L > 0$ ,  $L$  is the lag. So before diving into the details of the algorithm, we would like to briefly introduce the fixed-lag smoothing in KFM at first.

```
function Smoothing( $y_{1:T}, x_{10}, V_{10}, A, C, Q, R$ )
   $x_{00} = x_{10}$ 
   $V_{00} = V_{10}$ 

  for  $t = 1:T$ 
    ( $x_{t|t}, V_{t|t}, L_t$ ) = Fwd( $y_t, x_{t-1|t-1}, V_{t-1|t-1}, A, C, Q, R$ )
  end

  for  $t = T-1:-1:1$ 
    ( $x_{t|T}, V_{t|T}, V_{t-1,t|T}$ ) = Back( $x_{t+1|T}, V_{t+1|T}, x_{t|t}, V_{t|t}, A, Q$ )
  end

end
```

Figure 6. KFM Smoothing

The fixed-lag Kalman smoother (FLKS) has been proposed by Cohn et al. [3] as an approach to perform retrospective data assimilation. It estimates the state of the past, given all the evidence up to the current time, i.e.  $P(X_{t-L} = x | y_{1:t})$ ,  $L > 0$ , where  $L$  is the lag, e.g., we might want to figure out whether a pipe broke  $L$  minutes ago given the current sensor readings. This is traditionally called “fixed-lag smoothing”, although the term “hindsight” might be more appropriate. In the offline case, this is called (fixed-interval) smoothing; this corresponds to computing  $P(X_{T-L} = x | y_{1:T})$ ,  $T \geq L \geq 1$  [3] [4]. Figure 6 is the pseudo code of algorithm. Fwd and Back are the abstract operators. For each Fwd (forwards pass) operation of the first loop (for  $t=1:T$ ), we firstly compute the inference mean and variance by

$x_{t|t-1} = Ax_{t-1|t-1}$  and  $V_{t|t-1} = AV_{t-1|t-1}A' + Q$ ; then we compute the error in our inference (the innovation), the variance of the error, the Kalman gain matrix, and the conditional log-likelihood of this observation by  $err_t = y_t - Cx_{t|t-1}$ ,  $S_t = CV_{t|t-1}C' + R$ ,  $K_t = V_{t|t-1}C'S_t^{-1}$ , and  $L_t = \log(N(err_t; 0, S_t))$  respectively; finally we update our estimates of the mean and variance by  $x_{t|t} = x_{t|t-1} + K_t err_t$  and  $V_{t|t} = V_{t|t-1} - K_t S_t K_t'$  [4].

For each Back (backwards pass) operation of the second loop (for  $t=T-1:-1:1$ ), at first we compute the inference quantities by  $x_{t+1|t} = Ax_{t|t}$  and  $V_{t+1|t} = AV_{t|t}A' + Q$ ; then we compute the smoother gain matrix by  $J_t = V_{t|t}A'V_{t+1|t}^{-1}$ ; finally we compute our estimates of the mean, variance, and cross variance by  $x_{t|T} = x_{t|t} + J_t(x_{t+1|T} - x_{t+1|t})$ ,  $V_{t|T} = V_{t|t} + J_t(V_{t+1|T} - V_{t+1|t})J_t'$ , and  $V_{t-1,t|T} = J_{t-1}V_{t|T}$  respectively. These equations are known as Rauch-Tung-Striebel (RTS) equations [4].

```
function Predicting( $y_{1:T}, x_{10}, V_{10}, A, C, Q, R$ )
   $x_{00} = x_{10}$ 
   $V_{00} = V_{10}$ 
   $y_{T+1} = (y_{T-1} + y_T) / 2$ 

  for  $t = 1:T+1$ 
    ( $x_{t|t}, V_{t|t}, L_t$ ) = Fwd( $y_t, x_{t-1|t-1}, V_{t-1|t-1}, A, C, Q, R$ )
  end

  for  $t = T:-1:1$ 
    ( $x_{t|T}, V_{t|T}, V_{t-1,t|T}$ ) = Back( $x_{t+1|T}, V_{t+1|T}, x_{t|t}, V_{t|t}, A, Q$ )
  end

end
```

Figure 7. Prediction algorithm

In our prediction process, there are  $h$  more forward and backward passes. The computation of the passes is same with that in the smoothing process. The only difference is that in the prediction step the new observation is null, which means  $y_{1:T+h} = [y_{1:T} \ y_{null}^1 \ \dots \ y_{null}^h]$ . In practice, we use the previous steps as the prior, for example, if  $h = 1$ , then  $y_{T+1} = (y_{T-1} + y_T) / 2$ , so the pseudo code of prediction algorithm is shown in Figure 7.

In the prediction, we don't predict future workloads, but predict patterns of workloads which have already been running for a predefined period of time. The statistical axioms indicate that the data pattern

conforms to a certain distribution, so given the history and the distribution prior, we use the statistics (Bayesian) theory to make the prediction. If the prediction result is not coincident with the future brand-new workload injected, the prediction mechanism will consider the new data automatically.

The prediction task is to predict values on the future time, given all the observations up to the present time. That is to say, given the observation of the previous CPU package activity state values, we predict the observation value on next time slice. So the prediction task can be mathematically formulated by computing a probability distribution over the possible future observations and equation 4 is the solution to it. To implement the solution, we need the following assumption:

- We assume the pattern of the status value changing is time-invariant or homogeneous, so the parameters of the DBN model,  $A, C, Q, R$ , are same at any time.
- We assume the distributions of the status value and observation value are linear Gaussian and the changing dynamics of the status is unimodal, which means the status value in the next time slice would be the previous status value plus a Gaussian noise and the observation is equal to the corresponding status value plus a Gaussian noise. The mathematical formulation to the assumption is equation 1, 2 and 3.
- We also assume the pattern of the status value changing is a first-order Markov process, which means the whole pattern can be denoted by two time slice model, shown as Figure 5. And the current state depends only on the previous state.

Based on above assumptions and Equation 4, and according to the existing fixed-lag Kalman smoother (FLKS) method (shown as Figure 6), we use our proposed prediction method (as shown in Figure 7) to make the prediction of the observation value on next time slice.

### 4.3 Simplified implementation

In section 4.2, the computation looks complicated, e.g. there are matrix inversions in the  $T+1$  step loop, when computing Kalman gain matrix in Fwd operator and the smoother gain matrix in Back operator. And the computational complexity will be  $O(TN^3)$ , where  $T$  is the number of history observations;  $N$  is the number of activity states, because for a general  $N*N$  matrix, Gaussian Elimination for solving the matrix inverse leads to  $O(N^3)$  complexity.

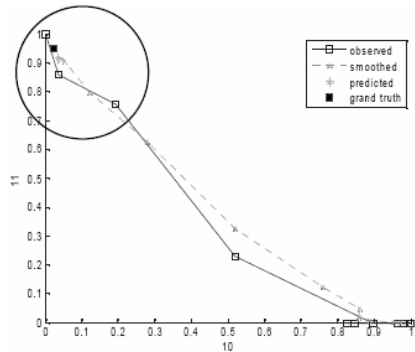
However, for our particular task, the algorithm implementation can be simplified. As shown in figure 7, we can cache the previous  $T$  step intermediate result of  $x$  and  $V$ , and just make one step update of Fwd for the new coming time slice  $T+1$ . Similarly, for the backwards pass, we can just compute one step Back operator for the  $T+1$  time slice, and using the cached previous  $T$  step intermediate results. Hence after the simplification, the computational complexity would be  $O(N^3)$ . Furthermore, The  $N = 2^{N'}$  ( $N'$  is the number of cores inside the same package) processor package activity states can be classified into 3 important categories: all idle, all busy and partial idle, and we can use these three categories to describe the states, so the  $N$  would become only 3. On the other hand, we can simplify the DBN model and set  $A, C, Q, R$  and initial  $V$  as diagonal matrix with the element value being 0 or 1, then the operation complexity of the algorithm will become much simple, as shown in Table 1. The total estimated compute time needed is 6.5us for next 500us time slice. This can be translated to 1.3% overhead on a 500MHz co-processor.

**Table 1. Complexity of the Simplified Prediction**

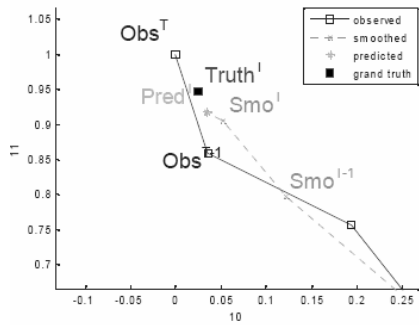
Operation	Addition	Subtraction	Multiplication	Floating point arithmetic in matrix inverse	Load +store	Operations and compute time estimate ( $N=3$ )
One step Fwd Complexity	$2N^2+3N$	$2N$	$2N^2+2N^3$	$N^3$	$2N^2$	148 (~3.0us @ 500MHz co-processor)
One step Back Complexity	$3N$	$N+N^2$	$4N^3+N^2$	$N^3$	$2N^2$	177 (~3.5us @ 500MHz co-processor)
Total Algorithm	$2N^2+6N$	$3N+N^2$	$3N^2+6N^3$	$2N^3$	$2N^2$	325 (~6.5us @ 500MHz co-processor)

## 5. Result

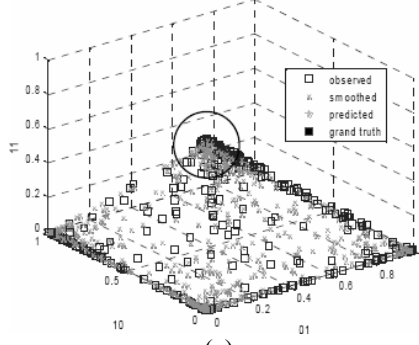
We implemented our prediction method with a published Kalman filter toolkit [5], and test the prediction method on several CPU activity data. Usually it is hard to train a DBN model precisely. We use the prior knowledge to set the parameters of DBN model,  $A, C, Q, R$ , so there is no training step in our system. And the parameters are not changed when we make online prediction/inference. The CPU activity data were sampled at ~50% system load level for both dual-core and quad-core processor.



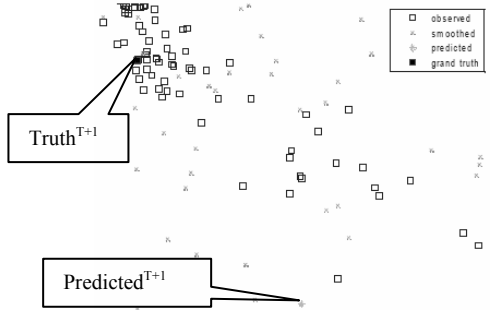
(a)



(b) Zoomed in



(c)



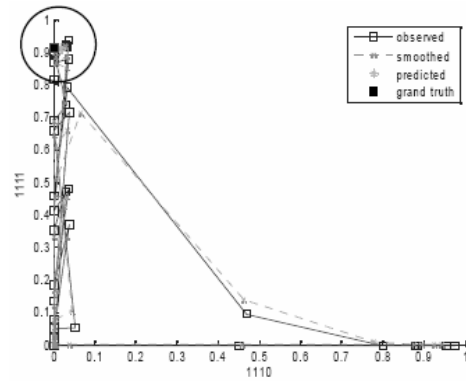
(d) Zoomed in

**Figure 8. Prediction performance, dual-core processor platform, T=1000**

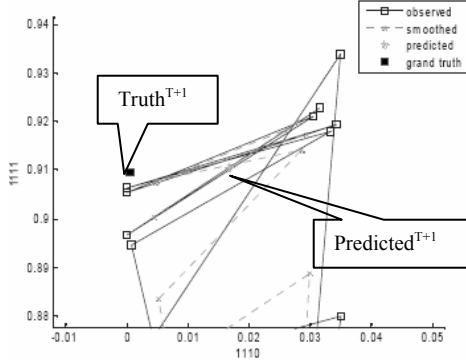
Figure 8 illustrates the result on dual-core processor based platform with 500 us sampling rate. In the

experiment, we set  $T=1000$ ,  $h=1$ , which means given 1000 observations, we predict the No. 1001 value. The final prediction and smoothing result consists of  $T+1$  vectors and each vector has 4 elements, whose values are the percentage of CPU 4 activity states time.

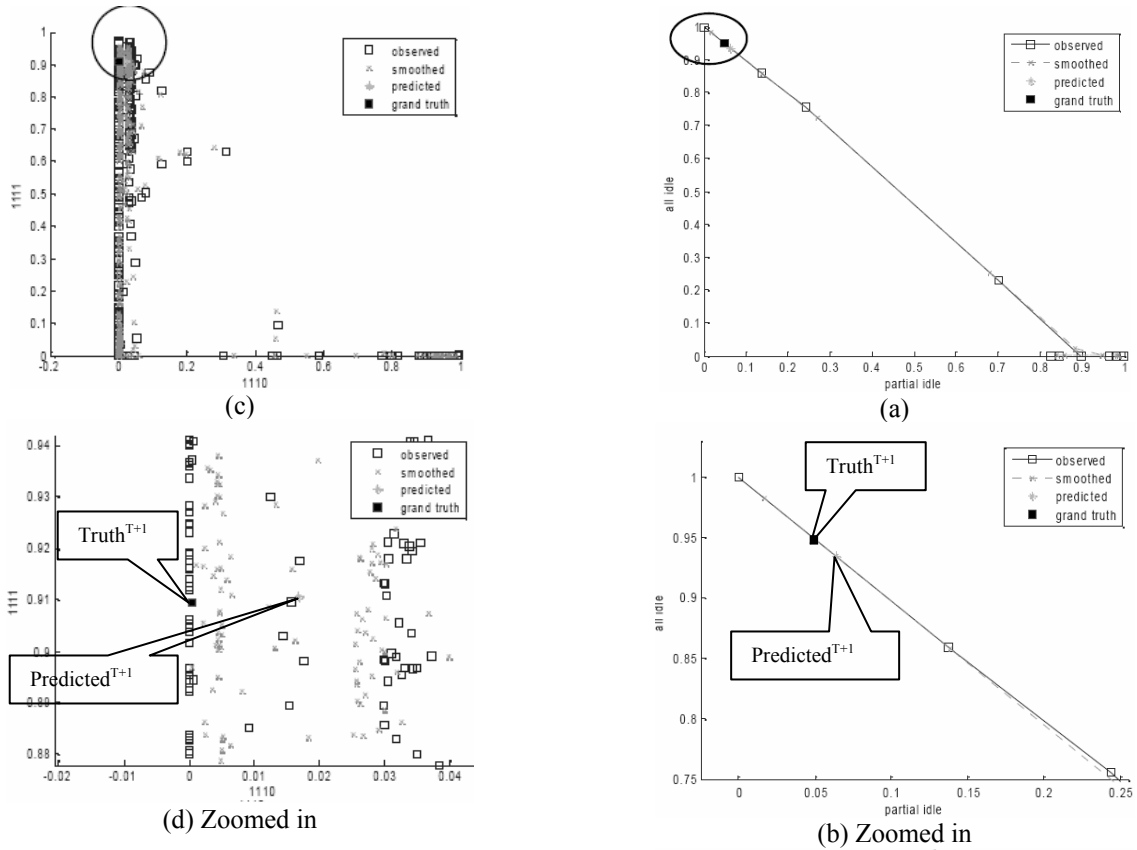
In order to show the details, we select two CPU activity states “10” and “11” as the x axis and y axis, and plot the last 10 time slices ( $t=T+1-10:T+1$ ) results, as shown in Figure 8(a). In the figure, we can see that the smoothing results capture the input observation pattern well, and the prediction for the final time slice ( $T=1001$ ), which is depicted with star point in the figure, is quite close to the grand truth (depicted with the black square point). We also zoom in the prediction point and show the further details in Figure 8(b). We map all the 1001 time slices to a 3-D space. The  $x, y, z$  axis are state “01”, “10” and “11” respectively, as shown in Figure 8(c) and Figure 8(d) (zoom in of the prediction point). The figure indicates that there are no outliers in the smoothing and prediction result, which means current DBN model has little system error and can effectively estimate the stochastic process in the input data.



(a)

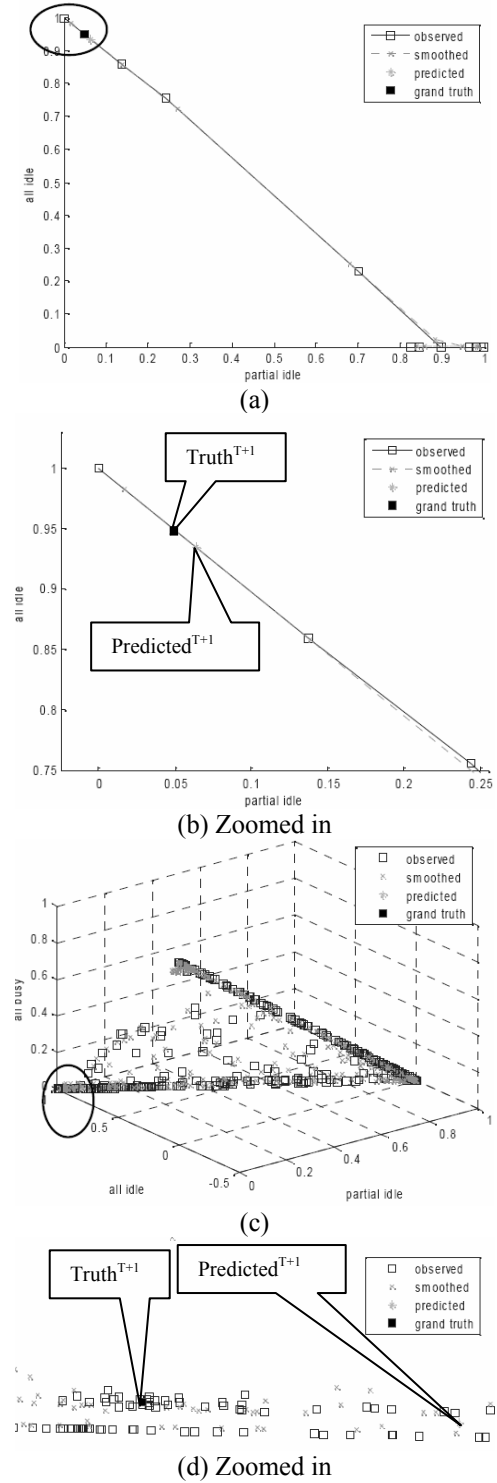


(b) Zoomed in

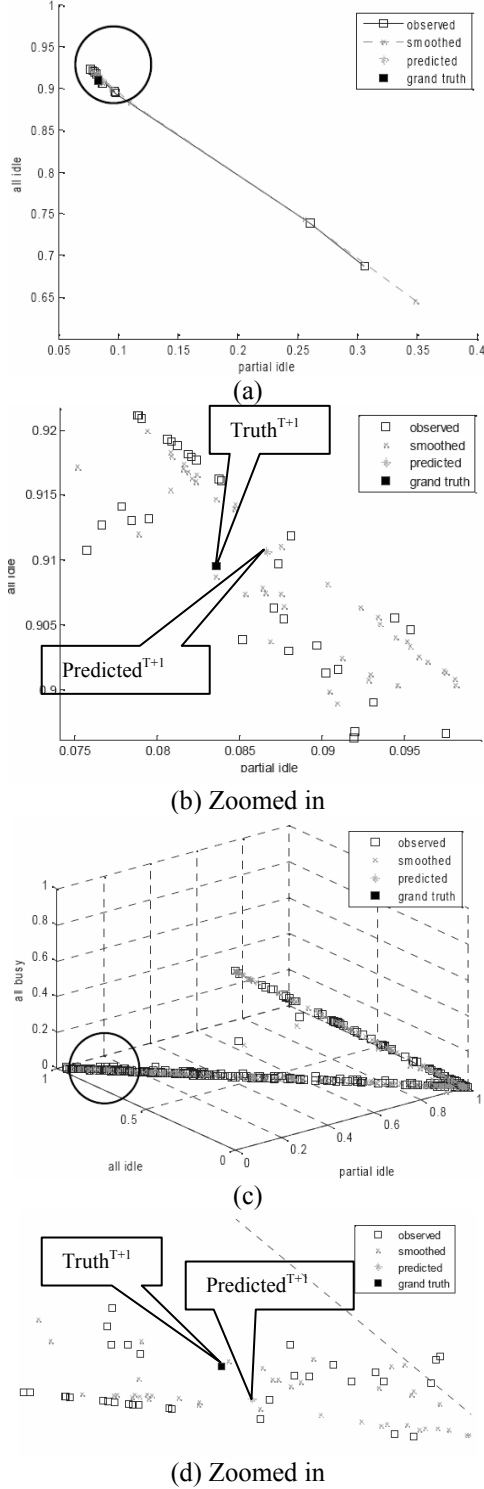


**Figure 9. Prediction result, quad-core processor based platforms,  $T=1000$**

Figure 9 describes the result on a quad-core processor based platform with 500 us sampling rate. The final prediction and smoothing result consists of  $T+1$  vectors and each vector has 16 elements, whose values are the percentage of CPU 16 activity states' time. In order to show the details, we select two CPU activity states "1110" and "1111" as the  $x$  axis and  $y$  axis, and plot the last 100 time slices ( $t=T+1-100:T+1$ ) result, as shown in Figure 9(a). In the figure, we can see that the smoothing results still capture the input observation pattern very well, and the prediction for the final time slice ( $T=1001$ ), which is depicted with star point in the figure, is very close to the grand truth (depicted with the black square point). We also zoom in the prediction point and show the further details in Figure 9(b). We map all the 1001 time slices to a 2-D space. The  $x$ ,  $y$  axis are state "1110", and "1111" respectively, as shown in Figure 9(c) and Figure 9(d) (zoom in of the prediction point). The figure also indicates that there are no outliers in the smoothing and prediction result, which means current DBN model has little system error and can effectively estimate the stochastic process in the input data.



**Figure 10. Prediction result, dual-core based platform,  $T=1000$ , all busy, aggregated partial idle and all idle**



**Figure 11. Prediction result, quad-core based platforms, T=1000, all busy, aggregated partial idle and all idle**

As described in section 3.1, all activity states can be abstracted into 3 states: all cores idle (1111), all cores busy (0000) and partially idle. Aggregated partially

idle state is the sum of all states except 1111 and 0000. So CPU activity state number becomes 3. The prediction result on the new data from dual-core processor based platform and quad-core processor based platform are shown in figure 10 and 11 respectively.

From the figures, we can see that both prediction and smoothing method has good performance. As stated before, prediction accuracy is the key to use core/package C-state for best power saving, so we compared prediction error levels as shown in Table 2. The absolute mean square error (expressed in absolute difference value from grand-true) and relative error of prediction (expressed in percentage) on the four data sets is about 0.03 and 4%, respectively. However, the sum data on quad-core processor based platform has much better performance, about 0.0033 mse and 0.36% relative error. We also notice that all the 1001 points in Figure 11(c) are distributed in two segmented lines, which means almost all the summed data on quad-core processor based platform can be formulated by simple linear equations. E.g. many points are distributed on the line of  $x+y=1$  on “partial idle” and “all idle” surface. The physical meaning behind is that the CPU cores almost do not have “all busy” state. This phenomenon also explains why our prediction method has much better performance on this data, because it has explicit simple pattern and in many circumstance the pattern can be well fit even with some simple linear equations.

**Table 2. The absolute mean square error and relative error of prediction on the four data sets**

Datasets	dual-core	quad-core	aggregated partial-idle on dual-core	aggregated partial-idle on quad-core
State number of CPU package activity states	4	16	3	3
Means square error of prediction	0.0387	0.0395	0.0204	0.0033
Relative error	4.08%	4.34%	2.15%	0.36%

## 6. Benefit analysis

We developed a power projection model which applies 1) Linux C-state policy [2] and 2) our prediction model on collected CPU idle-busy traces, and get average power for both cases. We assume the processor is quad-core, and each core’s C-state latency

and power numbers are based on ACPI spec example shown in figure 1. The results are shown in table 3.

**Table 3. Projected benefit by prediction**

	Linux C-state policy	Our prediction	Improvement
% of wrong decisions	26%	4.3%	Wrong decisions reduced by 83%
CPU Power saving by using C-state	3492mW*	3064mW*	Additional power saving 428mW*
Performance impact	2.1%**	0.09%**	Performance impact improved by 2%**

\*: based on power numbers in figure 1 (source: ACPI spec [1]). It doesn't represent real power of our experimentation processor.

\*\* : based on latency numbers in figure 1 (source: ACPI spec [1]). It doesn't represent real latency of our experimentation processor.

## 7. Summary

In this paper, we define CPU activity pattern in terms of core idle or busy status, and also use a statistical model based method for CPU activity pattern prediction. The CPU activity pattern is highly important to CPU power reduction by using core C-states or package low-power optimization technologies. The prediction has good result on real dual-core and quad-core platforms.

Current KFM (a special case of DBN) is based on linear Gaussian and unimodal dynamics assumption. For the different workload, we would like to try other DBN models, such as switching KFM, and other approximate inference methods to make the prediction under more general assumption (Gaussian mixture or multi-modal dynamics).

We would like to make further experiments to observe the average precision of method with longer time slices data. Also for different workloads, we need to verify if current model assumption (linear Gaussian and unimodal dynamics) is still good enough to capture all the stochastic patterns behind the data. We'll also need to continue the pathfinding efforts in how to use this prediction model for power reduction.

## References

- [1] Advanced Configuration and Power Interface Specification, <http://acpi.info>.
- [2] C-state policy in Linux kernel 2.6.21, <http://kernel.org>.
- [3] Todling, R., and S.E. Cohn, 1996: Some strategies for Kalman filtering and smoothing. In *Proc. ECMWF Seminar on Data Assimilation*, 91--111.
- [4] Kevin P. Murthy, Dynamic Bayesian Networks: Representation, Inference and Learning, PhD thesis, UC Berkeley, Computer Science Division, July 2002.
- [5] Kevin P. Murthy, Kalman filter toolbox for Matlab, June 2004. <http://www.cs.ubc.ca/~murphyk/Software/Kalman/kalman.html>
- [6] J. Hamilton. *Time Series Analysis*. Wiley, 1994.
- [7] Christopher Meek, David Maxwell Chickering, and David Heckerman. Autoregressive tree models for time-series analysis. In *Proceedings of the Second International SIAM Conference on Data Mining*, pages 229--244, Arlington, VA, April 2002. SIAM.
- [8] Qian Diao, Jianye Lu, Wei Hu, Yimin Zhang and Gary Bradski. Book Chapter: DBN MODELS FOR VISUAL TRACKING AND PREDICTION, *Bayesian Network Technologies: Applications and Graphical Models*, edited by Ankush Mittal, Ashraf Kassim and Tele Tan. IGI Global, USA, In Press, Aug. 2006.
- [9] Yimin Zhang, Qian Diao, Shan Huang, Wei Hu, Chris Bartels, Jeff Bilmes, DBN Based Multi-stream Models for Speech, In *Proceedings. (ICASSP '03), IEEE Computer Society*, pp. 836-839 vol. 1. Hong Kong, China, April 6-10, 2003.
- [10] Tao Wang, Qian Diao, Yimin Zhang, Gang Song, Chunrong Lai, Gray Bradski, A Dynamic Bayesian Network Approach to Multi-cue based Visual Tracking, in *Proc. of 17th International Conference on Pattern Recognition (ICPR'04)*, Vol. 2, pp. 167-170, Cambridge, UK, Aug., 2004.