Energy Modeling of Processors in Wireless Sensor Networks based on Petri Nets

Ali Shareef, Yifeng Zhu*
Department of Electrical and Computer Engineering
University of Maine, Orono, USA
Email: {ashareef, zhu*}@eece.maine.edu

Abstract

Power minimization is a serious issue in wireless sensor networks to extend the lifetime and minimize costs. However, in order to gain an accurate understanding of issues regarding power minimization, modeling techniques capable of accurately predicting energy consumption are needed. This paper demonstrates that Petri nets are a viable option of modeling a processor. In fact, this paper shows that the Petri nets' accuracy surpasses a Markov model utilizing supplementary variables to account for constant delays.

1 Motivations

Wireless sensor networks are becoming increasingly prevalent in a wide range of areas from surveillance [6] to monitoring temperature, humidity, and other environmental parameters [12, 10].

Wireless sensor network are usually comprised of nodes powered by batteries in locations where maintenance access may be difficult. Minimizing energy consumption in these networks would go a long ways toward extending the lifetime of the network and increasing the usability. However, in order to obtain a thorough understanding of the energy consumption characteristics in these networks, accurate modeling methods need to be developed.

Although the primary energy consumption in wireless sensor networks is for communication [2], the processor also serves a key role and there is a need to examine the energy characteristics of the embedded processors of these sensor nodes. A sound method of modeling provides a stable platform upon which the energy characteristics of innovative technology can be analyzed. One example of this is the capability of processors to power down to a low power mode after some time of no activity, which we will refer to as the *Power Down Threshold*. However, once powered down, the processor requires time *Power Up Delay* to reach operating mode again.

In this paper, we compare two different processor models: a Markov chain and a Petri net against software simulation. We show that the Petri net model is more accurate than the Markov chain. Section 2 introduces the use of both Markov models and Petri nets. Section 3 presents related work. Section 4 develops a model of a processor using a Markov model and a Petri net. Section 5 compares the results obtained by the simulation, Markov model, and Petri net, and Section 6 concludes this paper.

2 Introduction to the use of Markov Models, Petri Nets, and Software Simulation

Markov models have long been used to model systems dealing with exponentially distributed arrival and service rates. Markov models are composed of event chains where the likelihood of a system being in a particular state is independent of being in any other state. We can see how this restriction limits the usefulness of this method. Petri nets, on the other hand, are far more useful than Markov models. Initially Petri nets were based on Markov models, but since then, features of Petri nets' have been extended over the years that has blurred these similarities. Unlike Markov models, Petri nets are a simulation based approach. Petri nets can be thought of as a directed graph of nodes and arcs which are called places and transitions respectively. With Markov models, closed form equations can be derived that can be used to provide analysis about the modeled systems. Petri nets require that the modeled system be simulated for extended periods of time so that the *steady state probability* of the system is reached. Computer program such as TimeNet 4.0 [13] are available that can assist in creating and simulating Petri nets.

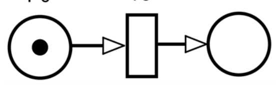

Figure 1. Example of Petri Net

Figure 1 depicts a simple Petri net that contains two places $P0$ and $P1$ and a transition $T0$. When a token exists in the input place of a transition as shown in $P0$, this enables the transition. When a transition is enabled, it fires according to user specified timing characteristics for that transition. When a transition fires, the token in the input place is removed and placed in the output place of the transition. In this case, the token in place $P0$ is moved to place $P1$. In this way, the operation of a system can be modeled and simulated. Statistical analysis of the average number of tokens in a particular place determines the steady state probability of the system being in that state. However, perhaps the most accurate method of modeling is through the use of software simulation where virtually anything can be programmatically modeled. Like Petri nets, software simulations are also repeated many times until the average behavior of the system can be achieved. A Markov chain can be used to construct analytical equations describing the average behavior of the system. However, mathematical difficulties that limit the usability of Markov models make simulation all the more appealing. On the other hand, the problem with simulation is that as the simulated system become more and more complicated, so does the software model of the system. This in turn results in increasing development, modification and testing time. Petri nets, on the other hand, can be constructed easily due to their graphical nature. The primary difference between the simulation, Markov, and Petri net model is that the Markov model utilizes a high level of abstraction. Whereas, the simulation and Petri net can be used to model at a much lower level closer to the actual hardware implementation. This enables the simulation and Petri net to account for the variations in the behavior of the processor [9].

3 Related Work

There have been many methods that have been proposed for modeling embedded systems to study energy characteristics. For example, Lee et al in [9] use energy measurements of instruction execution with regression analysis to derive equation that model energy consumption. Although this method achieves a high level of accuracy, this method can only be applied to computing energy during instruction execution. Energy consumption due to hardware interrupts is not accounted for. For example, the power up energy consumption is not linked with any instructions and hence cannot be modeled using this method.

Coleri et al [3] have demonstrated the use of a Hybrid Automata to model TinyOS and hence the resulting power consumption of the nodes. Models based on finite state machines have been proposed in [8]. Although [11] discuss the use of Petri nets to model a single processor utilizing real-time scheduling, we have not found any literature that discusses the use of Petri nets to model energy consumption.

4 Evaluation of CPU with Markov and Petri Net

4.1 Modeling Energy Consumption of CPU using a Markov model

Intrinsically, embedded systems operating in a wireless sensor network offer great potential for power minimization. Generally the level of computation required is low, and usually interspersed with communication between other nodes in the network. The power consumption of the CPU can be minimized by moving to a low power mode and conserving energy when it is not directly involved in any computation.

There are two forms of workload generators that can be used to simulate workload for an application. One type is known as the *closed* workload model where a new task will not arrive until the current task has been completed. The other is known as the *open* workload model in which the tasks arrive independent of the state of the current task. Closed workload generators are best suited for modeling tasks that occur at set intervals. Open workload generators, on the other hand, are suited for tasks that are interrupt driven. Both workload generators are used depending upon the application. We will implement an open workload generator.

In Figure 2 the CPU "powers up" (p_u) from a low power sleep mode when jobs begin arriving. The Markov model depicts the various increasing states (p_{01}, p_{02}, p_{03}, etc) the CPU enters as the number of jobs increase as given by the arrival rate λ. The CPU services the jobs at rate μ and strives to move the CPU to lower states and eventually to the $idle$ state p_i. If the processor remains in the "idle" state for some time interval greater than some threshold (*Power Down Threshold*), the CPU moves back to the $standby$ state.

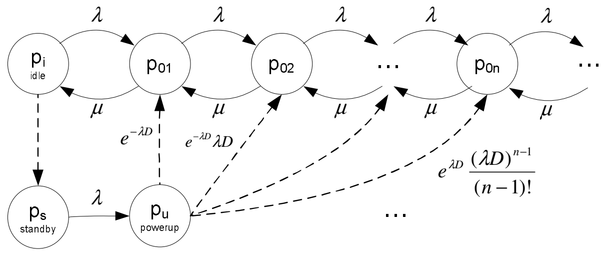

Figure 2. Birth-Death Process of CPU Jobs.

In this example, we make the following assumptions:

1. The request arrivals follow a Poisson process with mean rate λ.
2. The service time is exponentially distributed with mean $\frac{1}{\mu}$.
3. The CPU enters the $standby$ state (p_s) if there are no more jobs to be serviced for a time interval longer than T (*Power Down Threshold*).

4. The power up process takes a constant time D (*Power Up Delay*).

The CPU transition process consists of a mix of deterministic and exponential transitions. While all transitions shown as solid lines in Fig. 2 follow exponential time distributions, the transition from $idle$ to $standby$ is deterministic. The CPU enters the $standby$ state after idling for a constant time threshold T. This power down transition depends on its history and is not memoryless. Accordingly, the CPU transitions can not be modeled directly as a Markov process. Using the method of *supplementary variables* proposed in [4], we derive an alterative set of state equations to approximate the effects of constant delays in stationary analysis. Let $X = [x_1, x_2]$ denote two age variables, representing how long a deterministic transition has become enabled [5]. And let $\mathbf{P}_i(x_1)$ and $\mathbf{P}_u(x_2)$ be the age density functions with respect to x_1 in the idle state and with respect to x_2 in the power up state, respectively.

The state equations for this mixed transition process can be derived by the inclusion of the supplementary variable X. The deterministic transition from the $idle$ state to the $standby$ state can be modeled as below.

$$p_i = \int_0^T \mathbf{P}_i(x_1)dx_1 \quad (1)$$

$$\frac{d}{dx_1}\mathbf{P}_i(x_1) = -\lambda \mathbf{P}_i(x_1) \quad (2)$$

$$\mathbf{P}_i(0) = \mu p_{01} \quad (3)$$

$$\mathbf{P}_i(T) = \lambda p_s \quad (4)$$

where $\mathbf{P}_i$ is an exponential function with coefficient λ.

The deterministic power up process can be modeled as below.

$$p_u = \int_0^D \mathbf{P}_u(x_2)dx_2 \quad (5)$$

$$\frac{d}{dx_2}\mathbf{P}_u(x_2) = -\lambda \mathbf{P}_u(x_2) \quad (6)$$

$$\mathbf{P}_u(0) = \lambda p_s \quad (7)$$

In addition, when the system is stable, we have

$$(\lambda+\mu)p_{01} = \lambda p_i + \mu p_{02} + e^{-\lambda D}\mathbf{P}_u(0) \quad (8)$$

$$(\lambda+\mu)p_{0n} = \lambda p_{0,n-1} + \mu p_{0,n+1} + e^{-\lambda D}\frac{(\lambda D)^{n-1}}{(n-1)!}\mathbf{P}_u(0) \quad \text{for } n \geq 2 \quad (9)$$

$$1 = \sum_{n=1}^{\infty} p_{0,n} + p_i + p_s + p_u. \quad (10)$$

From equations (1), (2), (3), and (4), we can get

$$p_{01} = \frac{\lambda}{\mu}e^{\lambda T}p_s \quad (11)$$

$$p_i = (e^{\lambda T}-1)p_s. \quad (12)$$

From equations (5), (6), and (7), we have

$$p_u = (1-e^{-\lambda D})p_s. \quad (13)$$

We define the generating function of $p_{0,n}$ $(n = 1, 2, \ldots)$ as

$$G_0(z) = \sum_{n=1}^{\infty} p_{0,n}z^n. \quad (14)$$

We multiply equation (8) by z and equation (9) by z^n, add from $n = 1$ to ∞, use equations (7), (11), (12) (14), and get

$$G_0(z) = \frac{\lambda z p_s}{\mu - \lambda z}\left(e^{\lambda T} + \frac{e^{\lambda D(z-1)}-1}{z-1}z\right). \quad (15)$$

Thus we get

$$G_0(1) = \frac{\lambda p_s}{\mu-\lambda}\left(e^{\lambda T} + \lambda D\right). \quad (16)$$

Substituting equations (11), (12), (13), (14) and (16) into the normalization equation (10) gives

$$p_s = \frac{1-\rho}{e^{\lambda T} + (1-\rho)(1-e^{-\lambda D}) + \rho\lambda D} \quad (17)$$

where $\rho = \frac{\lambda}{\mu}$.

Consequently,

$$p_u = \frac{(1-\rho)(1-e^{-\lambda D})}{e^{\lambda T} + (1-\rho)(1-e^{-\lambda D}) + \rho\lambda D}. \quad (18)$$

And the utilization is

$$G_0(1) = \frac{\rho(e^{\lambda T} + \lambda D)}{e^{\lambda T} + (1-\rho)(1-e^{-\lambda D}) + \rho\lambda D}. \quad (19)$$

Let $L(z) = \sum_{n=1}^{\infty} np_{0n}\, z^n$, then $L(1)$ is the total number of jobs in the system.

$$L(z) = \sum_{n=1}^{\infty} np_{0n}\, z^n = z\frac{d}{dz}\left(\sum_{n=1}^{\infty} P_{0n}\, z^n\right) = z\frac{d}{dz}G_0(z) \quad (20)$$

Incorporating equation 15 into 20, we can get the total number of jobs in the system as follows.

$$L(1) = \frac{\rho}{1-\rho}\frac{e^{\lambda T} + \frac{1}{2}(1-\rho)\lambda^2 D^2 + (2-\rho)\lambda D}{e^{\lambda T} + (1-\rho)(1-e^{-\lambda D}) + \rho\lambda D} \quad (21)$$

According to Little's Law, the average latency for each job is

$$\tau = \frac{L(1)}{\lambda}. \quad (22)$$

The total running time is

$$T = \frac{N}{\lambda} + L(1)\tau = \frac{N + L(1)^2}{\lambda} \quad (23)$$

where N is the total number of jobs.

Thus the total energy consumption is

$$\begin{aligned} E &= (p_i P_{idle} + p_s P_{standby} + p_u P_{powerup} \\ &\quad + G_0(1) P_{active}) \frac{N + L(1)^2}{\lambda} \end{aligned} \tag{24}$$

where P_{idle}, $P_{standby}$, $P_{powerup}$ and P_{active} are the power consumption rate in the $idle$, $standby$, $powerup$ and active states (p_{01}, p_{02}, etc), respectively.

4.2 CPU Energy Modeling using a Petri Net

As was shown in the last section, the development of a Markov model for even a simple CPU is mathematically rigorous and cumbersome especially when dealing with deterministic transitions. Any slight modifications to the model will entail that the equations be re-derived again. Petri nets on the other hand offer a more flexible approach.

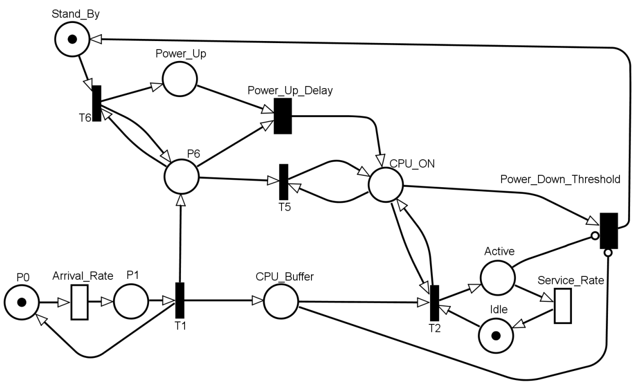

Figure 3. Petri net model of CPU

Transition	Firing Distribution	Delay	Priority
AR	Exponential	Arrivals	NA
$T1$	Instantaneous	–	4
$T2$	Instantaneous	–	1
SR	Exponential	ServiceRate	NA
PDT	Deterministic	PDD	NA
$T5$	Instantaneous	–	2
$T6$	Instantaneous	–	3
PUT	Deterministic	PUD	NA

Table 1. CPU Jobs Petri Net Transition Parameters

Fig. 3 shows an *Extended Deterministic and Stochastic Petri Net* (EDSPN) [1] modeling a CPU as the Markov model described earlier. The Petri net models a CPU that starts from some "stand by" (*Stand_By*) state and moves to an "ON" state (*CPU_ON*) when jobs are generated. The CPU remains in the "ON" state so long as there are jobs in the CPU buffer. If there are no jobs in the CPU buffer, after some time interval the CPU moves to the "stand by" state to conserve power.

The model uses an open workload generator because when transition $T1$ fires to deposit a task in the CPU_Buffer, a token is moved back to place $P0$ which enables transition $Arrival\_Rate$ and allows another task to be generated. Table 1 lists the parameters of all the transitions in the Petri net.

The CPU is simulated by executing the Petri net using the following steps:

1) Jobs are generated in place $P1$, when transition $Arrival\_Rate$ fires randomly in the interval $[0, 1]$ using an exponential distribution. The token in the place $P0$ is moved to $P1$ and enables $T1$.

2) Transition $T1$ is an immediate transition and fires as soon as it is enabled. Also since, $T1$ has the highest priority, it will fire before any other immediate transition if multiple immediate transitions are enabled at once. When $T1$ fires, the token is removed from $P1$ and three tokens are generated. One is deposited in place $P0$, one is deposited in place $P6$, and one is deposited in the place CPU_Buffer. Initially, the CPU is in the $Stand\_By$ mode. However, when a job arrives and a token is deposited in place $P6$, transition $T6$ is enabled.

3) When $T6$ fires, the two tokens from $Stand\_By$ and $P6$ are removed and two tokens are then generated. One is placed in place $Power\_Up$ and one is placed in $P6$. The CPU has now moved to a powering up state. Transition $Power\_Up\_Delay$ is now enabled with a token in $Power\_Up$ and in $P6$.

4) Since transition $Power\_Up\_Delay$ has a deterministic delay, the transition fires after a fixed interval, and a token is deposited in place CPU_ON. The CPU is now ON and ready to process events.

5) Remember that when $T1$ fired, a token was placed in the CPU_Buffer, this token, the token in CPU_ON, and the token in place $Idle$ enables transition $T2$. When it fires, the system is now in the processing state. With a token in $Active$, the transition $Service\_Rate$ is enabled.

6) After $Service\_Rate$ fires, the token is removed from $Active$ and placed in $Idle$.

7) In the event that another task arrives while the system is still ON and processing other tasks, a token will be deposited in $P6$ and CPU_Buffer. When the CPU is already ON, having a token in $P6$ will enable $T5$ which will fire immediately, and a single token will be placed in CPU_ON. This is necessary because tokens cannot be allowed to accumulate infinitely in any place.

8) The token that was added to the CPU_Buffer will remain there until the CPU is idle as determined by a token in $Idle$. All jobs that arrive while the CPU is ON will cause the Petri net to cycle through steps 7 and 8.

9) However, in the event that the job arrival rate is very slow, the CPU will power down and move to the $Stand\_By$ state. Note that when there is a token in CPU_ON and no tokens in $Active$ and CPU_Buffer, transition $Power\_Down\_Threshold$ becomes enabled.

The small circle at the ends of the arcs from $Active$ and CPU_Buffer specify this inverse logic. Since $Power\_Down\_Threshold$ is a transition with deterministic delay, it will fire after a specified period as given by the value PDD, and the token from CPU_ON will be removed and transferred to $Stand\_By$.

By computing the average number of tokens in places during the simulation time results in the "steady state percentage" of time the CPU spends in the corresponding state. For example, the average number of tokens in CPU_ON will indicate the percentage of time the CPU was on. The average number of tokens in $P7$ will indicate the steady state percentage of time that the CPU was "powering up." The percentages are determined by the system parameters $Arrival\_Rate$, $Service\_Rate$, $Power\_Up\_Threshold$, and $Power\_Down\_Threshold$ delays. Once the percentages are obtained, they can be used to compute the total energy consumption of the system over time as given in Equation 25.

$$\begin{aligned} TotalEnergy \quad = \quad & (P_{Standby} \times PW_{Standby} \\ & + P_{TimePoweringUp} \times PW_{PoweringUp} \\ & + P_{Idle} \times PW_{Idle} \\ & + P_{TimeActive} \times PW_{Active}) \times Time \end{aligned} \tag{25}$$

5 Comparison between Simulation, Markov Models and Petri Net performance

In order to determine the feasibility of Markov models and Petri nets for modeling the behavior of a CPU, this section will compare them together. An event simulator written in Matlab was used to simulate the CPU repeatedly over multiple iterations until the steady state percentages for the desired states was obtained. Equation 25 was used to compute the total energy. The simulation will be used as a benchmark against which the performance of the Markov model and Petri net will be compared. The power parameters of the $PXA271$ Intel Processor as given in [7] were used to provide realistic analysis.

Total Simulated Time	1000 sec
Arrival Rate	1 per sec
Service Rate	.1 per sec

Table 2. Simulation Parameters

Figure 4 compares the behavior of the CPU as predicted by the simulator, the Markov model, and the Petri net when the *Power Up Delay* is fixed to 0.001 sec while the *Power Down Threshold* is varied. The solid line represents the simulator, the dashes with squares the Markov model, and the dashes with circles the Petri net.

Since an open workload generator is being used, jobs arrive randomly in an interval. If the time between jobs exceeds the *Power Down Threshold*, then the CPU moves to the $Stand\_By$ state or "sleeps" to conserve power. When this happens, when the next job arrives, the CPU must transition to the $Power\_Up$ state where it spends time "waking up" (in the process consuming more energy) before it is able to service another job.

State	Power Rate (mW)
Standby	17
Idle	88
Powering Up	192.442
Active	193

Table 3. Power Rate Parameters for the PXA271 CPU (mW)

Figure 4 indicates that both the Markov model and Petri net predict steady state percentages that are close to the simulation results. In fact, as Table 4 indicates, the difference between the Markov model and simulation is less than the difference between the Petri net and simulation. However, as we can see from Table 4 that as the *Power Up Delay* becomes larger, the performance of the Markov model begins to suffer.

Power Up Delay (Sec)	Avg. Sim-Markov	Avg. Sim-PN	Avg. Markov-PN
0.001	0.338	0.351	0.076
0.3	4.182	1.677	3.338
10.0	116.788	16.046	103.077

Table 4. Δ Steady State Percentages (%) Estimates for Varying *Power Up Delay*

Figure 5 reveals the energy consumption characteristics of the CPU as the *Power Down Threshold* increases. It is obvious that as the CPU spends more time in the $Idle$ state which has a higher power consumption rate than the $Stand\_By$ mode, will result in increasing power consumed.

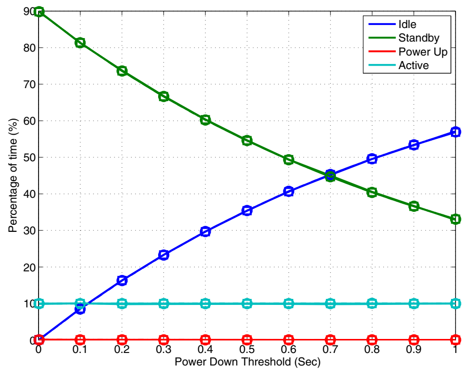

Figure 4. Comparison between performance of Simulation, Markov Models, and Petri Nets for Power Up Delay of 0.001 sec.

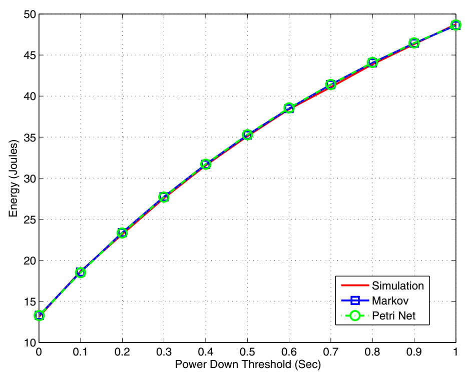

Figure 5. Energy Consumption Comparison between Simulation, Markov Model, and Petri Net for *Power Up Delay* **of 0.001 sec.**

Power Up Delay (Sec)	Avg. Sim-Markov	Avg. Sim-PN	Avg. Markov-PN
0.001	0.154	0.166	0.037
0.3	1.558	0.298	1.401
10.0	24.866	1.285	25.411

Table 5. Δ Energy Consumption (Joules) Estimates for Varying *Power Up Delay*

Table 5 indicates that the difference in energy prediction between the simulation, Markov model and Petri net is almost the same. However, as the *Power Up Delay* begins to increase, we see that the effects of the deviating steady state percentages negatively effect the energy predictions of the Markov model.

The approximation made for the effects of the constant delays in the Markov model is unable to account for the increasing constant delays and results in erroneous prediction for the Markov model.

6 Conclusion

The experimental results indicate that the Petri net is a better method of modeling processors than using a Markov model. This is due to the fact that a Markov model can only account for arrival and service rates that follow an exponential distribution. Markov models cannot handle fixed deterministic rates. Petri nets are also very easy to implement, and the complicated derivations of Markov models can be avoided. Any changes to the model can be made easily to a Petri net, while the Markov model will require re-derivation of the equations.

However, the drawbacks to Petri nets is their long simulation time that is required before the percentages stabilize. Evaluating a Markov models means just evaluating an analytical expression.

An interesting point to note is that when the constant delays are small such as when the *Power Down Delays* is 0.001 sec in Table 4 and 5, the performance of the Markov model is better than the Petri net. If an effective method of modeling constant delays in Markov chains can be derived, the Markov model may very well become the modeling method of choice.

References

[1] *TimeNET 4.0 A Software Tool for the Performability Evaluation with Stochastic and Colored Petri Nets, User Manual.*

[2] U. Cetintemel, A. Flinders, and Y. Sun. Power-efficient data dissemination in wireless sensor networks. In *MobiDE*, September 2003.

[3] S. Coleri, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modeling. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104, New York, NY, USA, 2002. ACM.

[4] D. R. Cox. The analysis of non-markovian stochastic processes by the inclusion of supplementary variables. *Proceedings Cambridge Philosophical Society*, 51(3):433–441, 1955.

[5] R. German. Transient analysis of deterministic and stochastic petri nets by the method of supplementary variables. In *MASCOTS '95: Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 394–398, Washington, DC, USA, 1995. IEEE Computer Society.

[6] T. He, P. Vicaire, T. Yan, Q. Cao, G. Zhou, L. Gu, L. Luo, R. Stoleru, J. A. Stankovic, and T. F. Abdelzaher. Achieving long-term surveillance in vigilnet. In *INFOCOM*. IEEE, 2006.

[7] D. Jung, T. Teixeira, A. Barton-Sweeney, and A. Savvides. Model-based design exploration of wireless sensor node lifetimes. In *Proceedings of the Fourth European Conference on Wireless Sensor Networks*. EWSN 2007, Jan. 2007.

[8] S. Kellner, M.Pink, D. Meier, and E.-O. Blass. Towards a realistic energy model for wireless sensor networks. In *Proceedings of IEEE Fifth Annual Conference on Wireless On demand Network Systems and Services*, pages 97–100, January 2008.

[9] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded risc processors. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–10, June 2001.

[10] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[11] M. Naedele. Petri net models for single processor real-time scheduling.

[12] J. Polastre, R. Szewczyk, A. Mainwaring, D. Culler, and J. Anderson. Analysis of wireless sensor networks for habitat monitoring. In *Wireless sensor networks*, pages 399–423, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[13] A. Zimmermann, M. Knoke, A. Huck, and G. Hommel. Towards version 4.0 of TimeNET. In *13th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems (MMB)*, pages 477–480, Mar. 2006.