Energy Conscious Online Architecture Adaptation for Varying Latency Constraints in Sensor Network Applications

Sankalp Kallakuri
ECE Department
Stony Brook University
New York, USA
elsanky@ece.sunysb.edu

Alex Doboli
ECE Department
Stony Brook University
New York, USA
adoboli@ece.sunysb.edu

ABSTRACT

Sensor network applications face continuously changing environments, which impose varying processing loads on the sensor node. This paper presents an online control method which adapts the architecture to minimize energy consumption while satisfying varying latency constraints. The method predicts processing load requirements over a finite time window and accordingly adapts the architecture. The behaviour of the hardware modules over time has been approximated with a Continuous Time Markov Process. Adaptive image processing for vehicle tracking was used as a case study for this approach.

Categories and Subject Descriptors

C.5.4 [**Computer System Implementation**]: vlsi systems; C.1.3 [**Processor Architectures**]: Other Architecture Styles, Adaptive Architectures

General Terms

Design

Keywords

Sensor Networks, Continuous Time Adaptation

1. INTRODUCTION

Sensor networks are emerging as a main technology for many applications in national security, health care, environmental monitoring, infrastructure security, food safety, manufacturing automation and many more [10] [13]. In fact, the vision is that sensor networks will offer ubiquitous interfacing between the physical environment and centralized databases and computing facilities [17]. Efficient interfacing has to be provided over long periods of time and for a variety of environment conditions, like moving objects, temperature, weather, available energy resources and so on. In this context, a key problem that ought to be tackled is that of devising embedded software and hardware architectures that can effectively operate in continuously changing, hard-to-predict conditions. In addition, architectures should be cheap and consume tiny amounts of energy - considering that their batteries are hard to replac or replenish.

For moving vehicle tracking, one of the main applications of sensor networks, the vehicle's velocity, trajectory and position defines the required sampling rate, hence the latency requirement for image processing [17]. In this case, architecture adaptation is challenging because static, off-line prediction of a vehicle's movement is quite inaccurate in real-life. Even if vehicle movement is predictable, the resulting off-line model is highly non-linear and discontinuous in many points. Therefor, it is quite inefficient to address this architecture adaptation problem by using typical embedded design methods [1] [3] [7] [12]. These consider static, quasi static or stationary scenarios, which all can be described through fixed, off-line models. As explained in the paper, vehicle tracking requires on-line model identification as well as continuous architecture adaptation to varying performance needs.

This paper presents a novel approach for online customization of embedded architectures that function in non-stationary environments. The crux of the approach is a synthesis technique for developing online controllers that adapt the data-path of an architecture to varying latency constraints while minimizing energy consumption. The approach includes three steps: *(i)* look ahead on performance parameters (like image sampling rate and system latency) by buffering input data coming in a given time window, *(ii)* dynamic processing requirements prediction using a linear estimator activated at the end of every window period, and *(iii)* on-line architecture adaptation. Since our design is for a non-stationary environments, the control policy varies with the environment but is stationary within a time window. Adaptive control policy design is based on expressing the operation over time of the data-path blocks as Continuous Time Markov Process (CTMP). A set of linear equations is set-up to reflect block utilization rates, buffer space constraints, and total energy consumption. Obtained utilization rates affect the adaptation thresholds of control policies. For systems with high utilization we could achieve upto 29% lesser power compared to greedy policy.

The proposed architecture adaptation method is different

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

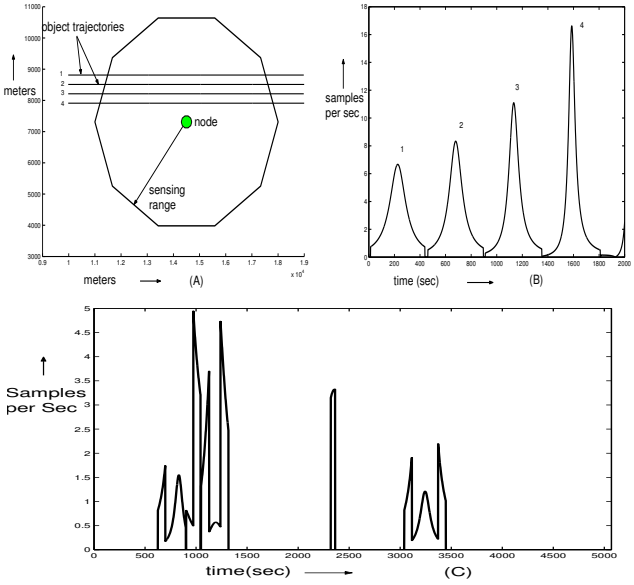

Figure 1: Trajectories and Sampling speeds

from other dynamic adaptation techniques including on-line software optimization and run-time hardware reconfiguration [5] [9] [16]. For example, dynamic partitioning for reconfigurable hardware selects the regions of the binary or source code to be sent to hardware modules [14] [15]. Other approaches are for reducing energy consumption by dynamic resource allocation [7] [11]. They predetermine the hardware configurations for certain static design requirements, and then search among them by using heuristics to improve energy consumption. We view the problem in a slightly different manner and feel there is a need to have a quick though sub-optimal control methodology for systems functioning in drastically varying dynamic environments, in which, optimal static policies do not exist, or exist only in the case in which under utilization of powered up hardware resources is ignored. In the latter case, the system suffers drastic overdesign to meet performance constraints under all possible load conditions.

The paper is organized as follows. Section 2 presents a motivating example and Section 3 defines the adaptive control problem. Section 4 details the mathematical modeling of the system and of the control policy. Section 5 is a discussion of results followed by conclusions in Section 6.

2. MOTIVATING EXAMPLE

In order to strengthen our case for the presence of highly varying process load environments, we present an example in which we show a relatively simple moving objects tracking scenario. We considered a camera based sensor [17], which is tracking a moving object, such as a person or vehicle. The tracking granularity requirement demands one image sample per meter of distance traveled by the object, in order to have a trace of the object's trajectory accurate to within one meter distance of the object's actual location at all times. If the object is traveling at a speed of 20 m/s this sampling speed would translate to having 20 samples/s for the camera.

Obviously, varying velocity of the object would require variation in the sampling speed, but the distance of the object from the sensor as well as the angle at which the object is traveling with respect to the focal plane could cause additional variation in the sampling speed requirements, as shown in Figure 1. In Figure 1(a), though trajectories are straight, the velocity component tangential to the camera's focal plane is varying, but varying smoothly. The sampling speed requirement changes faster for the trajectories nearer to the sensor, as observed in Figure 1(b). It may be possible in this case to create estimation models for the sampling rate, but this is a simple and non realistic scenario as compared to the one in Figure 1(c). Sampling rates in Figure 1(c) do not follow a smooth variation, and include points of discontinuity due to sudden changes in the vehicle's movement, like stopping, changes in direction, acceleration, and so on.

Through the mentioned example, we identified following attributes for sampling speed (thus also system latency) variation during vehicle tracking:

- Sampling speed and latency constraints are constantly changing without following any particular mathematical law. Performance variations might include several peak and bottom points and different convexities, concavities and discontinuities.
- Performance requirement magnitudes pertain to broad ranges of values. For example, the latency requirement for a busy sampling period can be 10× or even 100× higher than the sampling need for idle times.
- Performance variation gradients are in a wide range. Figure 1(a) shows that some variations are quite mild whereas others are very steep.

For this type of applications, it is difficult to formulate a static mathematical model that estimates performance needs without resulting in gross mispredictions. Such a "hypothetical" model would be highly nonlinear, discontinuous and partially defined. Most of the existing embedded design methods [3] [7] cannot be used in this case, as they need well defined, static description of performance constraints. It is intuitively understood that statically calculated optimizations are of little relevance in cases not covered by the estimation models. Stationary optimal control policies are quite unsuitable [4] because of discontinuities that are difficult to be handled by ordinary differential calculus. Hence, due to the discontinuous nature of the performance curves, these systems fit better into the framework of discrete events [2][8]. We modeled the system dynamics with discrete events formulated over a fixed window of time used to sample the future performance requirements of the system. As it is also difficult to predict the possible changes in processing load, a finite look ahead on this processing parameter is the best way of learning future processing loads.

3. PROBLEM DESCRIPTION

Problem Definition: For a given hardware architecture, device an adaptive, on-line control policy for each hardware resource such that (a) fixed buffer space and (b) varying latency constraints are met and (c) energy consumption is minimized.

To address the specifics of moving vehicle tracking problems, we propose a processing approach based on following three defining points:

1. There is a look ahead on performance parameters like sampling rate and latency.
2. Based on the look ahead there is a dynamic processing requirement prediction.

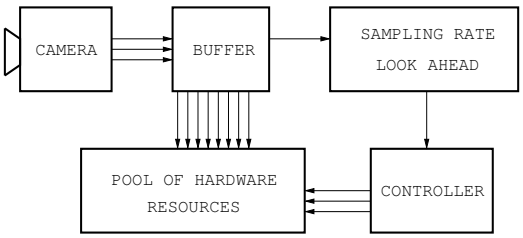

Figure 2: Architecture for proposed methodology

3. Online architectural adaptation takes place to reduce energy consumption and meet buffer space and varying latency constraints.

Look ahead and dynamic performance prediction is conducted within a fixed *time window* over which the data collected is to be processed.

Figure 2 presents the architecture that we used for implementing dynamic adaptation. During look ahead for the next window, the incoming data is buffered. The controller uses the inputs from the sampling rate look ahead to update its control policy. The controller makes changes to the pool of hardware with the updated policy after each window.

The time *window length* (WL)is a design parameter, and will have to be decided by the designer based on empirical data obtained from simulations of the particular application. Intuitvely, large time windows allow superior prediction but lengthen the adaptive response time and need more storage space. We ran several experiments to view the possible trends in the processing load variation as well as looked at the application's specification to gauge what window size would serve us. Although smaller window sizes would track the variations in requirements better, we have a lower limit to the window length. Specifically, we selected the limit to be the time required to process one sample if all resources of each type were used.

We defined *data load* (DL) as the data amount in terms of number of samples, which must be procesed in a time window. In the tracking example this amount of data may vary, depending on the rate the object moves. If DL is high then the number of hardware resources turned on is larger in order to meet the tighter timing constraint. There is an upper limit to DL based on the the hardware processing blocks and memory space being made available for it by an architecture.

4. MATHEMATICAL MODELING

We characterized the dynamics of the system architecture in the following manner. The state of the sytem is defined in terms of the resources that it is made up of. For example, a system with L, M and N number of resources of types R^1, R^2 and R^3 would have a state space vector S of the form $S = \{R_1^1, R_2^1 ... R_L^1, R_1^2, R_2^2 ... R_M^2, R^3, R_2^3 ... R_N^3\}$. Hence, if there are 10 elements for each of three types of resources, which are, ALUs, shifters and multipliers, the possible values that could be taken by each element of the state space is a cost which depends on the status of that element. In addition, each resource type R^i can be in one of its Z different *modes*. Resource modes depend on the processing activity of a resource and its present power mode status. Every element of each resource type has following four modes: (1) powered up and processing, (2) powered up and idle, (3) powered down, and (4) powered down and being requested. Please note that there are only two control actions asociated with each element, power up and power down. The obtained control policy is used to implement the controller block in Figure 2.

We encountered two major decision making steps while doing the mathematical modeling of this problem. The first being what mathematical framework would best model the variation of the system state over time, and the second being what control policy could quickly adapt with the system. The traditional methods [1] [7] [12] have been designed for static or stationary environments where there isn't a need for adaptation in the control policies. These issues were presented next.

A. System state variation modeling. As motivated in Section 2, we decided to express the system state variation using difference equations. The difference equation that characterises the system is given by

$$S_{k+1} = f\{S_k, U_k, E_k\} \tag{1}$$

which states that the next state is a function of the current state, the control vector U_k and certain look ahead E_k. The control vector U_k is a set of control actions that were taken at the k_{th} time instant. The sampling rate variation has been obtained by monitoring the incoming data rate, which works at the granularity of the window length. The effect of this look ahead is modeled by the term E_k.

Buffering data over the time window allows the control method to obtain knowledge of the latency requirements of the system for the window period, hence it can incorporate this knowledge into the decisons it makes.

B. Control policy. The controller implements the following control policy, which consists of certain state transitions under certain conditions. The decisions taken in the control policy are which hardware elements and how many of them to turn on or turn off. The control policy scales in the following manner for all types of elements, though the actual policy for each will differ in numbers.

$$U_{(w)} = \begin{cases} N_4^T, N_2^T, N_3^T \longrightarrow N_1 & if\ SR_w < SR_{w+1}, \\ N_2^T \longrightarrow N_3\ and\ N_4^T \longrightarrow N_1 & if\ SR_w = SR_{w+1}, \\ N_2^T, N_4^T, N_1^T \longrightarrow N_3 & if\ SR_w > SR_{w+1}, \end{cases} \tag{2}$$

SR_w and SR_{w+1} are the sampling requirements of the current window and the next window respectively. SR_{MAX} is the maximum sampling speed the system can tackle. Beyond this sampling speed there will not be enough hardware resources to speed up the processing. The number of elements that make the transition from current state i are given by N_i^T. N_j is the number of devices that need to be in next state j during next time window. The elements that transit to state j follow the priority with which they are listed. This has been formulated in the following manner for the first case of the control policy shown above.

$if\ SR_w > SR_{w+1}$,

$$N_1 = \begin{cases} \sum_{u \in U_w} \lambda_{1,u}\ N(1 + \Delta SR_w) & if\ SR_{w+1} < SR_{MAX}, \\ N & otherwise, \end{cases} \tag{3}$$

$$N_2^T = \begin{cases} N_2 & if\ N_1 \geq N_2 \\ N_1 - N_2 & otherwise \end{cases} \tag{4}$$

$$N_4^T = \begin{cases} 0 & if\ N_1 < N_2 \\ N_4 & if\ N_1 - N_2^T \geq N_4 \\ N_1 - N_2 - N_4 & if\ N_1 - N_2^T < N_4 \end{cases} \tag{5}$$

$$N_3^T = \begin{cases} 0 & if\ N_1 - N_2 < N_4 \\ N_3 & if\ N_1 - N_2^T - N_4^T \geq N_3 \\ N_1 - N_2 - N_4 - N_3 & if\ N_1 - N_2^T - N_4^T < N_3 \end{cases} \tag{6}$$

Equations (3)-(6) basically state the optimal control policy according to which any element, which is idle and powered

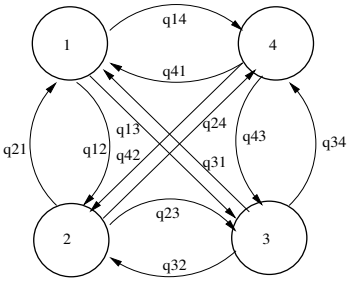

Figure 3: State transitions for power modes

*

up should be either shut down or put to use depending on the next windows latency requirements. The prediction of the powered up elements in the next state N_1 was done by taking into consideration the change in sampling rate $\Delta SR = \frac{SR_w - SR_{w+1}}{SR_w}$, as well as the *likelihood* λ of an hardware element being in state "1" (power up and processing) which we have approximated by the steady state probability of an element being in state "1" $\alpha 1 = \sum_{u \in U_w} \alpha_{1,u}$. Linear equations (7)-(11) are solved to find the steady state probabilities$\alpha_{i,u}$. Likelihood captures the global influence of a certain hardware resource on system performance, thus it jointly reflects the criticality of the block with respect to timing, buffer size needs and energy consumption, as well as the amount of resources of that kind in an architecture.

The policy states that if any element is powered down and being requested it should be turned on or shut down again depending on sampling rates. The power down or power up has to be done from different states depending on the severity of the gradient. Thus, elements in state "4" should move to state "1" first, then if N_1 is not satisfied, elements from state "2" should make transitions, and so on. This is a set of constraints for the elements of type R^1 which are N_1 in number. Similar constraints exist for the other elements and for the two other cases of the control policy.

C. Finding the likelihood factors. Finding the precise likelihood value of a resource is an NP-complete problem, as it requires finding the optimal architecture for given constraints. Instead, we approximated likelihood with the steady state probability of a single hardware element modeled as Continuous Time Markov Chain (CTMC) [4], as shown in Figure 3. CTMC were previously used for control policy design, including power mode controllers [1] [12] and bus arbiters [6]. Another advantage of this modeling is that it offers - for each hardware resource, a figure of merit that cumulatively expresses its time criticality, usage, energy consumption and impact on buffer size. Defining likelihood using an heuristic cost function would have been an alternative. However, having no rigorous mathematical support, we avoided this possibility.

Steady state probabilities $\alpha_{1,u}$ were calculated by modeling the mode changes $q(j,u)$ are the rates for choosing certain transition u while in a certain mode j. The rates $q(i,j,u)$ and $q(j,u)$ have been obtained by using a simple scheduling algorithm. The scheduling algorithm works on a threshold of time after which the hardware element powers down. The threshold depends on the energy consumed in powering down $E_{turnoff} + E_{turnon}$ and the energy saved by being in the power down mode over a certain period of time $P_{dn}t$. The time threshold is given by T_{th}, $T_{th} = \frac{E_{turnoff}+E_{turnon}}{P_{dn}}$ Thus for all idle times $t_{idle} \geq T_{th}$ the hardware element will be powered down.

The rates $q(i,j,u)$ are the rate for selecting a certain action while being in state i and going to state j, as shown in [12][4]. $Bprop_k i, a$ are the buffer occupancy rates for the hardware elements, and we treat them as costs which cannot exceed a certain amount of available buffer space.

$$\sum_{u \in U(j)} q(j,u)\alpha_{j,u} - \sum_{i \in I} \sum_{u \in U(i)} q(i,j,u)\alpha_{i,u} = 0, j \in S, \quad (7)$$

$$\sum_{i \in S} \sum_{u \in U(i)} Bprop_k(i,u)\alpha_{i,u} \leq C_k, k = x, y, z, \quad (8)$$

$$\sum_{i \in S} \sum_{u \in U(i)} \alpha_{i,u} = 1, \quad (9)$$

$$\alpha_{i,u} \geq 0, i \in S, u \in U(i), \quad (10)$$

The goal of minimizing the energy consumption of the system while meeting buffer size constraints is expressed by the following set of equations.

$$minimize \;\; C_{window} = \sum_{i=1}^{N} \sum_{u=1}^{U(i)} \sum_{k=1}^{D} C_k \alpha_{i,u} \quad (11)$$

C_{window} is the total cost over the time window, and is the sum of the cost incurred at each clock cycle.

$$C_k = \sum_{a=1}^{L} Cx_{a,k} + \sum_{b=1}^{M} Cy_{b,k} + \sum_{c=1}^{N} Cz_{c,k} \quad (12)$$

The cost at k_{th} clock cycle is given by a summation of the costs incurred by all hardware elements. We would like to state at this point that this summation of costs is simply in the mathematical modeling, and the controller will not be performing such summing operations, and will not be looking at the cost function. These equations will be solved offline and the solution shall be encoded into the controllers. If each hardware processing block has the four modes presented in Section 3, the cost functions of the elements follow the following pattern.

$$Cx_{i,k} = \begin{cases} P_{up}\Delta t + B_{prop} & if\ x_{i,k} = 1, \\ P_{up}\Delta t + B_{prop} & if\ x_{i,k} = 2, \\ P_{dn}\Delta t + B_{prop} & if\ x_{i,k} = 3, \\ P_{dn}\Delta t + B_{prop} & if\ x_{i,k} = 4, \end{cases} \quad (13)$$

The cost of an element in the powered up states (state "1" and state "2") is the costs associated with being powered up P_{up}. In state "3" and state "4" the cost is simply that of powering down, and in each state there is an added penalty based on the buffer space that got occupied while the element was being requested B_{prop}.

The equations could be solved for the state space as a whole but this would lead to too many equations due to the large state space we are dealing with. We have made an assumption that the steady state transitions probabilities of an individual element would be the same for all elements of its type. Another assumption we made is that it is safe to consider the steady state transition probabilities of each element independent of the probabilities of the other, since the timing delays of the elements are different and obey the following relationship $Tmull >> Talu = Tshift$ and the occurrence of instructions that utilise these elements in an algorithm is different too. The occurrence rates of intructions which could use these resources in the RGB to CMY colour conversion algorithm that we considered had a the highest

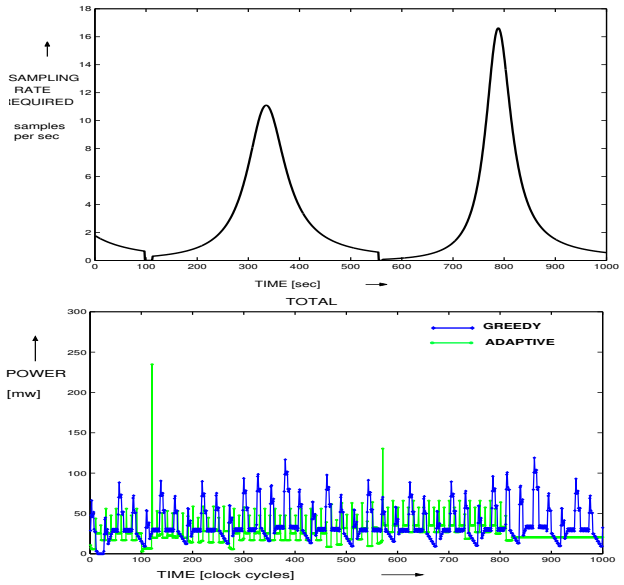

Figure 4: Sampling Requirements and Total Power

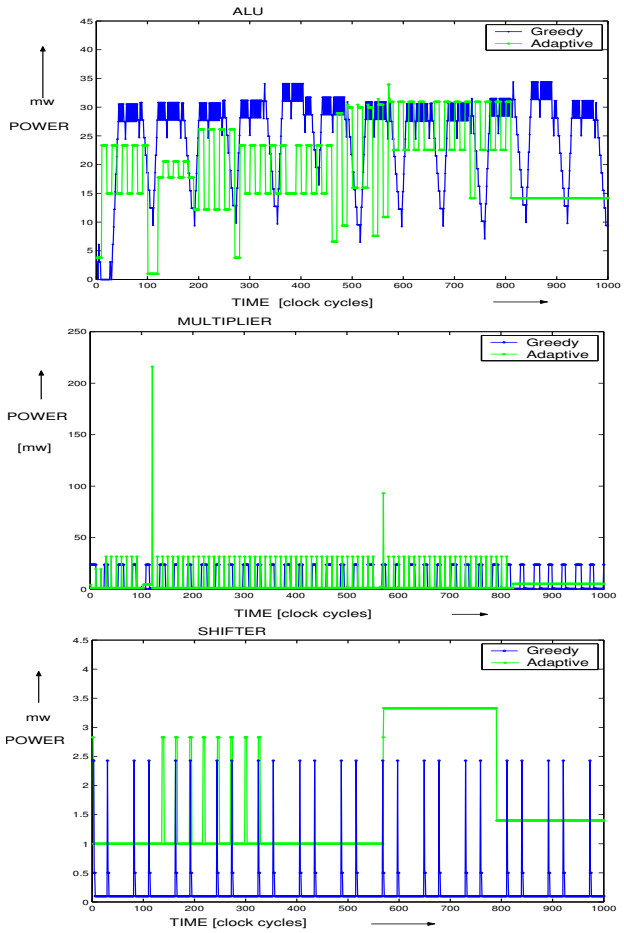

Figure 5: Power trends for different resources

rate for intructions which could use ALUs followed by multipliers and shifters. In other words the number of adders powered up has no correlation with the number of multipliers powered up as they wouldn't make much difference in satisfaction of overall timing constraints.

D. Controller circuit. The output of the set of equations is the steady state probabilities for the state and control action pairs given by $\alpha_{i,a}$ as shown in [4]. These steady state probabilities $\alpha_{i,a}$s with the sampling rate are used in the controller. They provide the basic structure for the control policy and these are then scaled by the sampling rate as in equation (3).

5. EXPERIMENTS

The task graphs used belong to the R,G,B to Y,Cb,Cr colour space conversion algorithm, which is composed of three task graphs executed in parallel [17]. Several instances of thistask graphs may also be run on parallel on separte pixels of the image. Though the graphs have similar structure their execution times will depend on the incoming data due to the several data dependent branching operations. The powering up and powering down of Hardware resources consequently has differing rates even if the processing load or samples per sec. remains the same.

We implemented a SystemC model of an architecture which has a bank of 10 ALUs, 10 Multipliers and 10 Shifters. Each of the resources has a controller which implements the control policy discussed in section 4. The elements are connected to a data bus, which carries the sampled data. The controllers are sensitive to the sampling rate and perform a buffering operation depending on *Data Load* and a look ahead on the sampling rate before powering up or powering down resources during a window. The equations (7)-(11) were solved using MATLAB6p1 to obtain steady state probabilities $\alpha_{i,u}$. The steady state probabilities for each type of resource were then embedded into the controllers.

The sampling rate variation has been shown in Figure 4. The reason we chose such variation is we wanted to test how well the control policy adapts to the varying required processing rate. The power consumption trend follows the required sampling rate as more resources get turned on to meet the smaller latency constraints for higher sampling rates. The "Greedy" policy which turns on every requested resource and turns them off based on a time threshold T_{th}, to which we compared our "Adaptive" policy, tends turn on more resources hence has a larger power consumption. To its advantage the Greedy policy manages to track the variation in processing requirements faster. For a window length of 10 clock cycles and a *data load* of 480 we found the total power consumption in the adaptive policy was 29% lesser than that of the greedy policy. The *Data Load* of 480 signifies that a maximum of 480 samples could be buffered, though the actual maximum buffer space used was close to 200 samples hence we could have applied tighter buffer space constraints on the equations (7)-(11) and used a smaller buffer space.

The individual types of resources had their own controllers, hence had different trends in terms of their power consumption. The ALU had a high utilization rate due to the large number of ALU operations in the task graphs. Due to the large number ALU operations the ALU bank had fraction of ALUs powered on all the time and extra ALUs were turned on/off depending upon the requirements as shown in Figure 4. The greedy policy tracked the variation better but overdesigned in terms of turning on more ALUs which became idle over a certain fraction of the *window length*.

The shifters had a low utilization as there are only 2 shift operations among the 81 total operations in the three task graphs. The greedy policy turned on the shifters exactly when needed and performed better than the adaptive policy in terms of power consumption. The adaptive policy performed poorly as it tried to adapt when there really wasn't need to adapt. The adaptive policy kept some shifters powered up even though they were idle as seen in Figure 5. It also unnecessarily powered up extra shifters while trying to adapt to the second peak in processing requirements shown in Figure 4.

The multipliers had a low utilization rate similar to the shifters and the greedy policy performed better in terms of lower power consumption. We see a large spike in the multipliers graph in Figure 5. This is because the adaptive policy tried to turn off all multipliers when the sampling rate went very low as shown in Figure 5. This created a backlog of "multiply" operations which were then serviced at once by turning on extra multipliers.

The window length has to be carefully selected in order to allow the adaptive policy to track the variation in the sampling requirements yet conserve power. This can be observed for the ALU bank from Figure 6, in which the adaptive policy tends to look more like the greedy policy and consumes more power while trying to track varying processing requirements, for example, when window length (WL)=2. With longer window lengths (WL)=10, 50 and 100 the policy reduces power consumption while being insensitive to the varying latency requirements, thus the adaptive policy will require larger buffer space for larger windows. In case the window length is large its possible that the hardware resource will become idle over a fraction of the window length thus wasting power. Since the ALU had a high utilization rate this did not occur. In resources with low utilization the power trend with varying window size was different.

In the experiments the power savings from the ALU dominate the poor performance of the control policy in the case of shifters and multipliers. In case of low utilisation rates of resources or low occurrence rates of instructions which could utilise a particular resource the greedy policy, which turns on all requested resources, may perform comparably with the adaptive policy. In future attempts we shall apply the adaptive policy to a commercial core which has power down modes and selectively controllable resources.

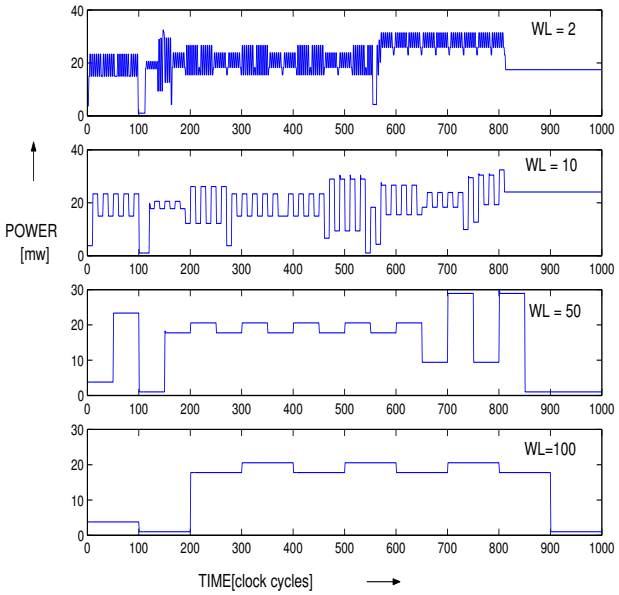

Figure 6: Varying Window Lengths

6. CONCLUSIONS

We have shown that the adaptive policy performs well for resources with high utilization rates under varying latency constraints. We could obtain up to 29% lesser power consumption compared to a greedy policy for certain design constraints. With well chosen window length and data load parameters we can get a control policy which is obtained offline, yet can be near optimal for online adaptation of embedded systems suffering varying latency constraints.

7. REFERENCES

[1] L. Benini, A.Bogliolo, and G. Micheli. A survey of design techniques for system level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, June 2000.

[2] C. G. Cassandras and S. LaFortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[3] P. Eles, Z. Peng, K. Kuchinski, and A. Doboli. *System Level Hardware/Software Partitioning using Simulated Annealing and Tabu Search*. Kluwer Academic Publishers, 1997.

[4] E. Feinberg. Optimal control of average reward constrained continuous time finite markov decision processes. *Proceedings of the IEEE Conference on Decision and Control*, pages 3805–3810, 2002.

[5] S. Hauck, M. H. T.W. Fry, and J. Kao. The chimaera reconfigurable functional unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[6] S. Kallakuri, A. Doboli, and S. Doboli. Stochastic modeling based environment for synthesis and comparison of bus arbitration policies. *Proceedings of IEEE Annual Symposium on VLSI*, pages 199–206, 2004.

[7] J. Khan and R. Vemuri. An iterative algorithm for battery aware task scheduling on portable computing platforms. *Proceedings of Design Automation and Test in Europe*, pages 622–627, 2005.

[8] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 2000.

[9] R. Lysecky and F. Vahid. A study of speedups and comptetiveness of fpga soft processors cores using dynamic hardware/software partioning. *Proceedings of Design Automation and Test in Europe*, pages 18–23, 2005.

[10] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of ACM International Worshop on Wireless Sensor Network Applications*, 2002.

[11] B. Miramond and J. Delsome. Design space expolration of dynamically reconfigurable architectures. *Proceedings of Design Automation and Test in Europe*, pages 366–371, 2005.

[12] Q. Qiu, Q.Wu, and M.Pedram. Stochastic modeling of a power-managed system: Construction and optimisation. *IEEE Transactions on Computer Aided Design*, 20(9):1200–1217, October 2001.

[13] M. Rahimi, R. Pon, W. Kaiser, G. Sukhatme, D. Estrin, and M. Srivastava. Adaptive sampling for environmental robots. *Proc. of International Conference on Robotics and Automation*, 2004.

[14] G. Stitt and F. Vahid. Hardware/software partitioning of software binaries. *Proceedings of the International Conference on Computer Aided Design*, pages 164–170, 2002.

[15] G. Stitt and F. Vahid. Dynamic hardware/software partioning: A first approach. *Proceedings of the Design Atomation Conference*, pages 250–255, 2003.

[16] M. Writhlin and H. B. Disc: The dynamic instruction set computer. fpgas for fast board developement and reconfigurable computing. *Proceedings of the SPIE 2607*, pages 92–103, 1995.

[17] Y.Weng and A. Doboli. Smart sensor architecture customized for image processing applications. *Proceedings of IEEE Real-Time and Embedded Technology and Embedded Applications*, pages 396–403, 2004.