# Adaptive Timeout Policies for Fast Fine-Grained Power Management

**Branislav Kveton**
Intel Research
Santa Clara, CA
*branislav.kveton@intel.com*

**Prashant Gandhi**
Intel Research
Santa Clara, CA
*prashant.gandhi@intel.com*

**Georgios Theocharous**
Intel Research
Santa Clara, CA
*georgios.theocharous@intel.com*

**Shie Mannor**
Department of Electrical and
Computer Engineering
McGill University
*shie@ece.mcgill.ca*

**Barbara Rosario**
Intel Research
Santa Clara, CA
*barbara.rosario@intel.com*

**Nilesh Shah**
Intel Research
Santa Clara, CA
*nilesh.n.shah@intel.com*

**Abstract**

Power management techniques for mobile appliances put the components of the systems into low power states to maximize battery life while minimizing the impact on the perceived performance of the devices. Static timeout policies are the state-of-the-art approach for solving power management problems. In this work, we propose adaptive timeout policies as a simple and efficient solution for fine-grained power management. As discussed in the paper, the policies reduce the latency of static timeout policies by nearly one half at the same power savings. This result can be also viewed as increasing the power savings of static timeout policies at the same latency target. The main objective of our work is to propose practical adaptive policies. Therefore, our adaptive solution is fast enough to be executed within less than one millisecond, and sufficiently simple to be deployed directly on a microcontroller. We validate our ideas on two recorded CPU activity traces, which involve more than 10 million entries each.

## Introduction

Recent advances in hardware and wireless technologies have made mobile computers ubiquitous in our daily life. Extending their battery life has naturally become a major challenge. A significant effort has been invested into developing laptop batteries that store more energy. Less attention has been paid to design better power management (PM) policies. The main purpose of a good PM policy is to decrease power consumption without affecting the perceived performance of devices. The ideal policy turns off the device when its services are not needed in the future. The device is turned on just before it is needed. Unfortunately, whether a particular device is needed in the future is typically unknown at the time of taking power management actions. This uncertainty about the future is the main problem of efficient PM. Thus, the better the prediction of the future, the more effective the resulting PM solution.

Timeout policies are state-of-the-art commercial solutions in PM. The policies turn off the controlled device when it has not been used for some predefined time. A familiar example of timeout policies are PM schemes in Microsoft Windows. Timeout policies are fairly simple and robust. However, they may react too fast or too slow to any change in user behavior. As a result, recent research tried to address the importance of *adaptive* decision making in PM.

This recent work can be roughly viewed as predictive, optimization, and online methods. Predictive methods (Srivastava, Chandrakasan, & Brodersen 1996; Hwang & Wu 1997) predict idleness periods from observed events. This idleness prediction establishes a basis for taking informative actions. Optimization approaches (Benini *et al.* 1999; Simunic 2002) usually formulate the original problem as a stochastic model, such as a Markov decision process. This model is used when searching for the optimal policy. Online learning algorithms (Helmbold *et al.* 2000; Dhiman & Simunic 2006) derive an adaptive PM policy from a set of expert policies. The expert policies are domain-specific heuristics.

In this paper, we focus on the domain of fine-grained PM. In particular, we discuss the power management of the complete processing unit comprised of multi-core processors, L1 and L2 caches, and associated circuitry. This PM problem is emerging as a domain for developing simple, intelligent, and efficient machine learning solutions. Solving the problem is important because the complete processing unit may account for as much as 40 percent of the power consumed by mobile computers. In the rest of this paper, we use the term *package* to refer to the complete processing unit.

The primary goal of package PM is to minimize the power consumption of the package without impacting its perceived performance. This performance objective can be restated as maximizing the *residency* of the package in low power states while minimizing the *latency* in serving hardware interrupts. The latency is a delay caused by waking up the package from low power states. To find a practical solution to this problem, we apply online learning. Furthermore, we demonstrate that resulting online policies yield both higher residency and less latency than state-of-the-art baselines in our domain.

The paper is structured as follows. First, we introduce the

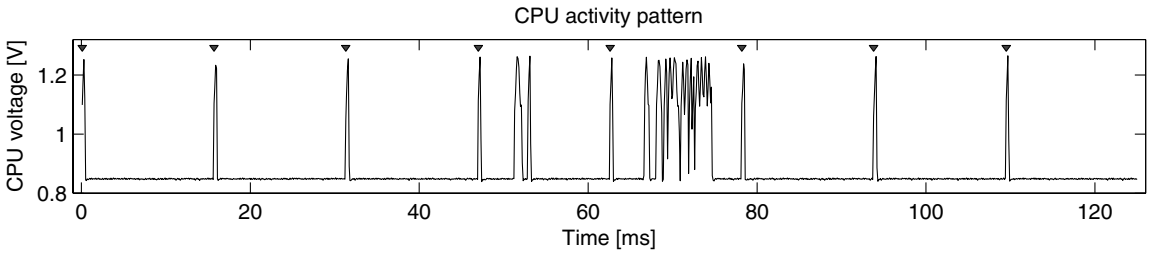


Figure 1: An example of a CPU activity pattern. The voltage is presented as a function of time (in milliseconds). Black triangles denote OS interrupts. Note that the distance between two consecutive OS interrupts is 15.6 ms. Due to this periodicity, software interrupts in Microsoft Windows can be easily predicted.

domain of package PM. Second, we discuss the state-of-the-art approach for solving the package PM problem. Third, we illustrate the benefit of using adaptive timeout policies in our domain. Fourth, we propose a simple adaptive policy, which is learned online using machine learning. Finally, the policy is evaluated on two CPU activity traces, which involve more than 10 million entries each. The first trace is synthetic. The second trace is generated by 30 minutes of human activity.

## Package power management

Package power management involves the complete processing unit including multi-core processors, L1 and L2 caches, and other associated circuitry. The central part of this system is a prediction module, which monitors recent CPU requests and predicts future idleness periods that are sufficiently long to power down the package. Our work demonstrates that this prediction can be done in both efficient and adaptive manner. In contrast to the existing adaptive power management work (Chung, Benini, & de Micheli 1999; Helmbold *et al.* 2000; Dhiman & Simunic 2006), we focus on package PM and not the power management of individual hardware devices, such as hard drives or network cards. Our prediction module must operate on the less than millisecond time scale, which limits the complexity of potential adaptive solutions.

To obtain high granularity CPU activity patterns, we monitor the core voltage of the Intel Core Duo CPU every 0.1 ms (Figure 1). After the voltage signal is recorded, we threshold it at the value 0.87 V. Voltages above this threshold are CPU states C0 (active), C1, and C2. Voltages below the threshold are CPU states C3 and C4. In short, active CPU states C0 are a result of hardware interrupts, which are caused by a variety of hardware activity, or software interrupts, which Microsoft Windows generates every 15.6 ms. Due to this repeating pattern, software interrupts in Microsoft Windows can be easily predicted. The challenging task is the prediction of hardware interrupts.

In the next section, we discuss a simple yet state-of-the-art solution to predicting idle CPU periods between two consecutive OS interrupts. This idleness time is sufficient to power down the package if we guarantee that no hardware interrupt occurs. In the rest of the paper, we generalize this solution to adapt to changing CPU workloads.

To make our PM study more realistic, we assume that the power-up time of the package is 1 ms. Moreover, we assume

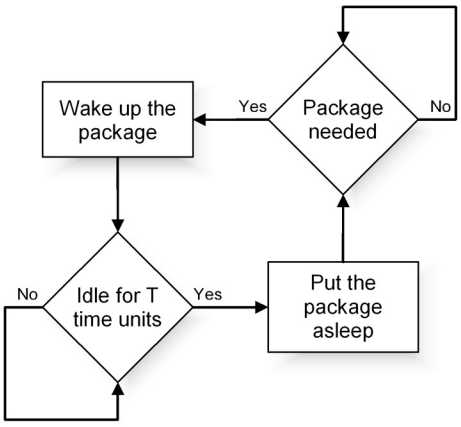


Figure 2: A flowchart representing a static timeout policy $T$ for package PM.

that when the package remains in its low power state for less than 10 ms, it wastes more power by transitioning in and out of this state than is saved by being in this state. Therefore, to save at least some power between consecutive OS interrupts, we must transit into the low power state within 5 ms from the latest OS interrupt. If the package is in a low power state and a hardware interrupt occurs, the package immediately wakes up to serve the interrupt. Due to the delay in performing this task, the package incurs a 1 ms latency penalty. These values are realistic and suggested by domain experts.

## Static timeout policies

A *static timeout policy* (Karlin *et al.* 1994) is a simple power management strategy, which is parameterized by the timeout parameter $T$ (Figure 2). When the controlled device remains idle for more than $T$ time units, the policy puts it asleep. The device is woken up when it is needed, which typically causes latency due to the power-up time of the device.

Static timeout policies are state-of-the-art solutions in various domains (Chung, Benini, & de Micheli 1999; Helmbold *et al.* 2000; Dhiman & Simunic 2006) because the past idleness of a device is usually the best univariate predictor of its future idleness. The parameter $T$ naturally trades off the residency of the policies for their latency. The same conclusion

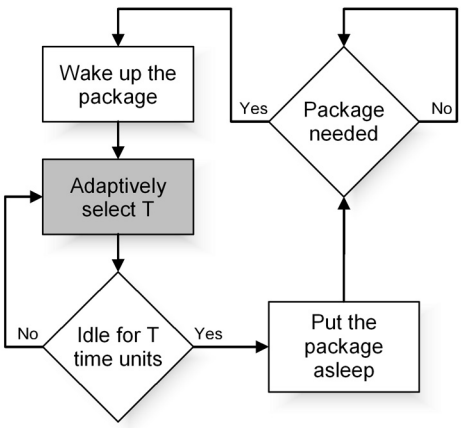


Figure 3: A flowchart representing an adaptive timeout policy for package PM.

holds in our domain (Figure 4). As shown by Figure 4, varying the timeout parameter $T$ indeed affects the residency and latency of policies. Note that the package wakes up ahead of the OS interrupts because these events are predictable.

## Adaptive timeout policies

An *adaptive timeout policy* is a timeout policy that modifies its timeout parameter $T$ (Golding, Bosch, & Wilkes 1996) to optimize some performance objective (Figure 3). In the PM domain, this objective function often trades off the residency of the policy for its latency. Existing evidence (Helmbold *et al.* 2000; Dhiman & Simunic 2006) suggests that even very simple adaptive policies may yield higher residency than the best performing static timeout policy at a given latency level.

The benefit of adapting timeout parameters in our domain is demonstrated in Figure 5. A simple adaptive policy $T_{\mathrm{A}}$:

$$T_{\mathrm{A}}^{(t)} = \begin{cases} 5 & \text{policy } T_{\mathrm{A}}^{(t-1)} \text{ caused latency between the OS interrupts } t-1 \text{ and } t \\ \frac{T_{\mathrm{A}}^{(t-1)}-1}{2}+1 & \text{otherwise,} \end{cases} \quad (1)$$

which increases its timeout parameter $T_{\mathrm{A}}^{(t)}$ to 5 ms whenever it incurs latency, and decreases $T_{\mathrm{A}}^{(t)}$ towards 1 ms otherwise, generates 12 percent higher residency than the best performing static timeout policy $T = 3.7$ ms (Figure 4) at the same latency. Figure 5 also shows an ideal adaptive timeout policy $T_{\mathrm{I}}$. This policy achieves the highest residency of all adaptive timeout policies within the latency limit of 1 ms. This upper bound illustrates the ultimate benefit of adaptiveness without being committed to a particular adaptive mechanism.

The *ideal adaptive timeout policy* for a given latency limit is derived greedily as follows. First, note that the policy must put the package asleep between all consecutive OS interrupts where no hardware interrupt occurs. The power-down action is performed immediately after the first of the OS interrupts, which yields the highest possible residency for 0 ms latency. No additional residency can be achieved without exchanging it for some latency. To achieve the highest residency for $\ell$ ms of latency, the policy must put the package asleep during the $\ell$ longest idleness periods that are terminated by a hardware event.

### Online learning

In the rest of the section, we discuss online learning of adaptive timeout policies. In particular, we apply the *share* algorithm (Herbster & Warmuth 1995) to learn how to adapt their timeout parameters. The algorithm is a well-known machine learning method based on voting of experts. The method can be implemented in any domain with temporal aspects and its theoretical properties are well studied (Herbster & Warmuth 1995). It often produces good results (Helmbold *et al.* 2000; Gramacy *et al.* 2003) with almost no tuning. We believe that the approach can take advantage of the temporal correlations between hardware interrupts in our CPU activity traces (Figure 1). As a result, we learn contexts for applying aggressive and conservative PM policies.

In our domain, the experts are represented by power management strategies. We assume that these strategies are static timeout policies, which are specified by their timeout parameters $T_i$. The share algorithm uses the timeout parameters $T_i$ to compute the global adaptive timeout parameter $T_{\mathrm{G}}$, which is given by the linear combination:

$$T_{\mathrm{G}}^{(t)} = \frac{\sum_i w_i^{(t)} T_i}{\sum_i w_i^{(t)}}, \quad (2)$$

where $w_i^{(t)}$ is a weight corresponding to the $i$-th expert at the time $t$. At every OS interrupt, all weights $w_i^{(t-1)}$ are updated based on the recent performance of the experts as:

$$\begin{aligned} \omega_i &= w_i^{(t-1)} \exp[-\eta \mathrm{Loss}\,(t, T_i)] \\ \mathrm{pool} &= \sum_i \omega_i \left(1-(1-\alpha)^{\mathrm{Loss}(t,T_i)}\right) \\ w_i^{(t)} &= (1-\alpha)^{\mathrm{Loss}(t,T_i)}\omega_i + \frac{1}{n}\mathrm{pool}, \end{aligned} \quad (3)$$

where $\eta > 0$ is a learning rate, $0 < \alpha < 1$ is a share parameter, $\mathrm{Loss}\,(t, T_i)$ quantifies the loss of the $i$-th expert between the OS interrupts $t-1$ and $t$, and $n$ denotes the total number of the experts.

The learning rate $\eta$ and the share parameter $\alpha$ control how fast the weights of poorly predicting experts are reduced and recovered, respectively. The loss function reflects the quality of the timeout policies $T_i$ and is given by:

$$\begin{aligned} \mathrm{Loss}\,(t, T_i) = {} & c_{\text{wasting}} \\ & [\text{total idle CPU time} \\ & \text{when the package is not asleep} \\ & \text{under the policy } T_i] \\ & + \\ & c_{\text{latency}} \\ & [\text{total unpredicted power-up time} \\ & \text{under the policy } T_i], \end{aligned} \quad (4)$$

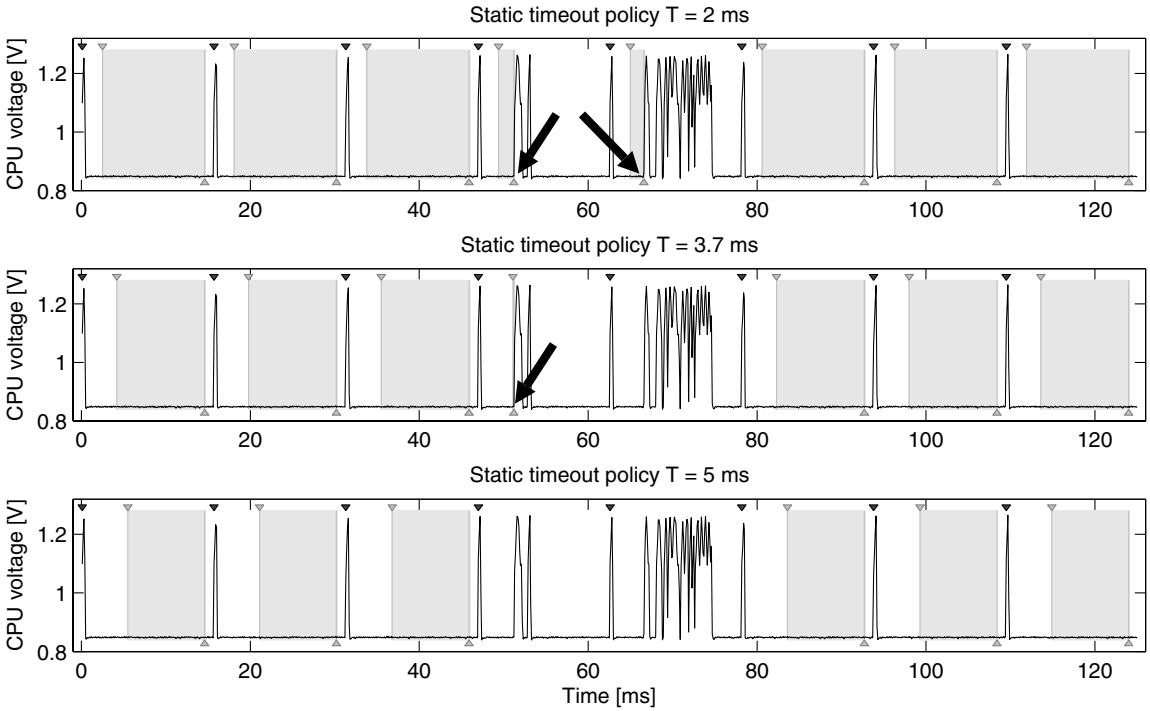


Figure 4: Power management actions generated by three static timeout policies. The policies put the package into its low power state 61.44, 50.56, and 44.16 percent of time. The policies produce 2, 1, and 0 ms latency. Black triangles denote OS interrupts. Gray triangles at the top and bottom correspond to the time when the package was put asleep and woken up. The time at which latency occurs is marked by arrows.

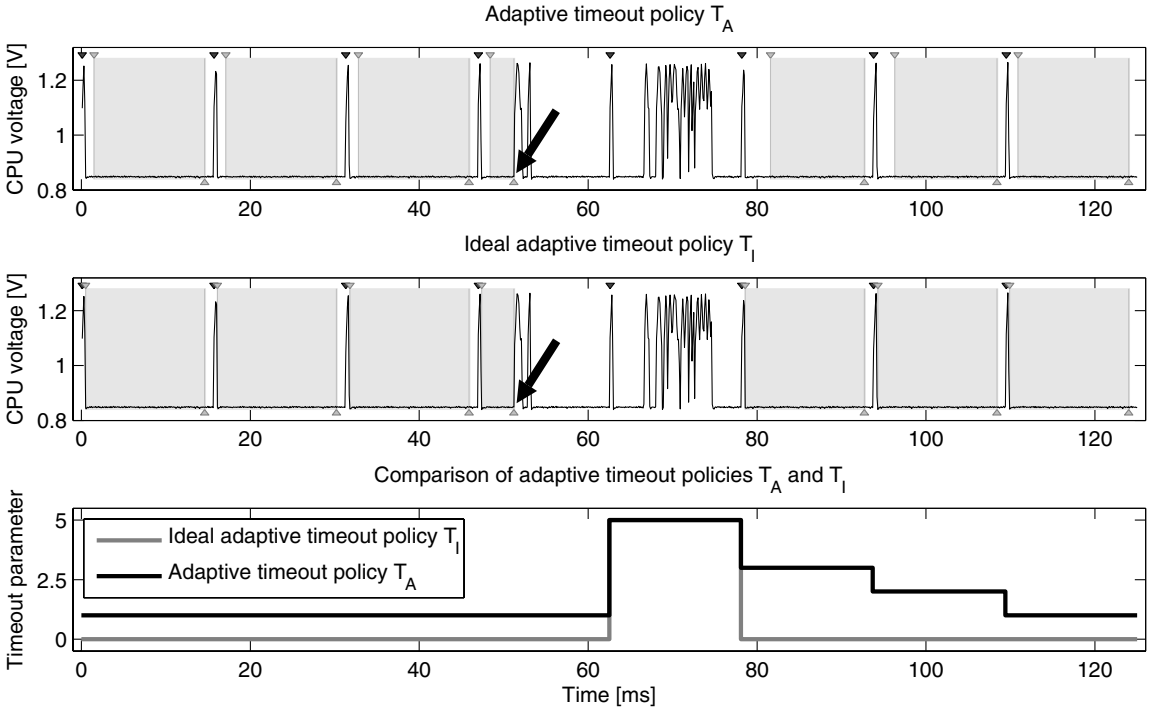


Figure 5: Power management actions generated by an adaptive timeout policy $T_{\mathrm{A}}$ and the ideal adaptive timeout policy $T_{\mathrm{I}}$ at the same latency. The policies put the package into its low power state 63.28 and 71.28 percent of time. Both policies cause only 1 ms latency. Black triangles denote OS interrupts. Gray triangles at the top and bottom correspond to the time when the package was put asleep and woken up. The time at which latency occurs is marked by arrows.

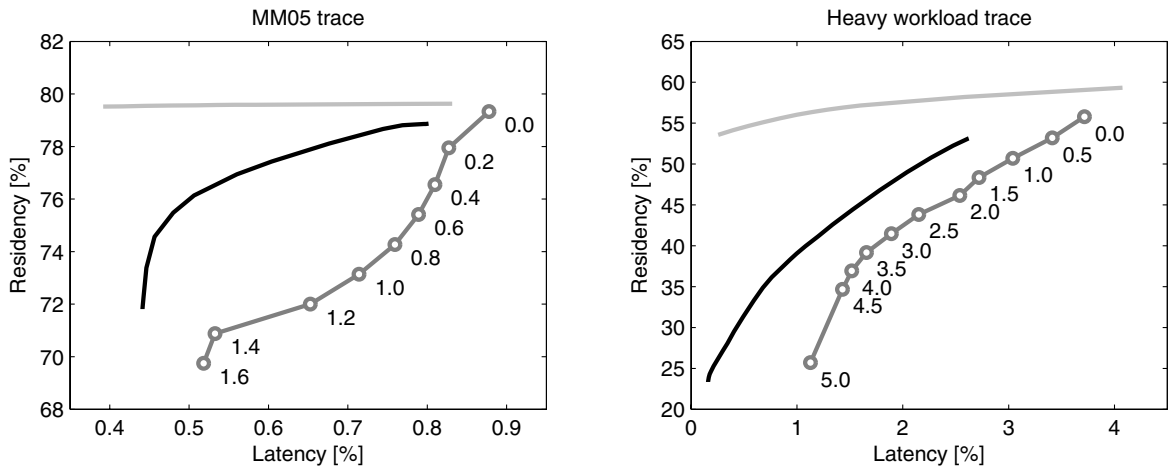


Figure 6: Comparison of static (dark gray lines) and ideal adaptive (light gray lines) timeout policies to learned adaptive policies (black lines) on two CPU activity traces: MM05 and a heavy workload trace. Numbers at the circles correspond to the timeout parameters of static timeout policies.

where the parameters $c_{\text{wasting}}$ and $c_{\text{latency}}$ quantify our preference between causing latency and being in low power states. In the experimental section, the parameters $\alpha$ and $\eta$ are set at the values of 0.0001 and 1, respectively. Setting of the share parameter $\alpha$ close to zero corresponds to ignoring the shared pool of experts. As a result, the learning rate $\eta$ is a redundant parameter. Its value can be compensated by simply adjusting the weights $c_{\text{wasting}}$ and $c_{\text{latency}}$.

## Experiments

The goal of the experimental section is to compare static and adaptive timeout policies in a real-world power management scenario. This comparison is performed on two CPU activity traces collected from the Intel Core Duo CPU.

### Experimental setup

The first trace is recorded during running MobileMark 2005 (MM05). MM05 is a performance benchmark that simulates the activity of an average Microsoft Windows user. A corresponding CPU activity trace is 90 minutes long and contains more than 50 million records. The second trace is generated by running Adobe Photoshop, Microsoft Windows Explorer, Microsoft WordPad, and Microsoft Media Player. This trace reflects 30 minutes of human activity and contains more than 14 million records. In the rest of the section, we refer to it as a heavy workload trace. Note that this trace is not synthetic.

Both static and adaptive timeout policies are evaluated for various latency and residency levels. This allows us to study the overall benefit of adapting timeout parameters. To obtain static timeout policies of different aggressiveness, we simply vary the timeout parameter $T$. Every adaptive timeout policy is built from four timeout experts: 0.0, 0.4, 1.6, and 25.6 ms. The experts are chosen to cover all possible idleness periods between consecutive OS interrupts. To get adaptive timeout policies of different aggressiveness, we modify their latency weight $c_{\text{latency}}$. The residency weight $c_{\text{wasting}}$ is kept constant at the value of 0.01. Our results are summarized in Figure 6.

In the rest of the paper, the latency of policies is measured in percentage as:

$$\text{latency} = 100\frac{\text{total unpredicted power-up time}}{\text{total active CPU time}}. \quad (5)$$

This adjustment allows for correcting the length and idleness of different CPU activity traces.

### Experimental results

Based on our results, we can draw the following conclusion. Adaptive timeout policies constantly yield higher residency than static timeout policies at the same latency. For instance, on the heavy workload trace, they yield more than 15 percent higher residency for lower latency levels. These policies can be also viewed as lowering the latency of static timeout policies at the same residency. For example, on the MM05 trace, adaptive timeout policies reduce the latency to almost a half of its original amount. Even if this improvement is certainly impressive, our upper bounds (Figure 6) clearly indicate that there is still a lot of room for improvement.

Encouraged by our results, we decided to examine learned adaptive policies. At the highest level, the policies represent the following rule: apply the aggressive static timeout policy $T = 0$ ms unless the amount of recent latency exceeds some threshold. When it happens, prevent all package PM actions until the recent latency drops below the threshold. Note that this rule is learned automatically and has not been suggested by our domain experts.

The computational cost of policies plays an important role in the power management domain. Our adaptive policies can be computed naively in five times more time than any regular static timeout policy. This computational overhead is caused by the simulation of the four timeout experts, which is done in addition to executing the adaptive policy. Fortunately, the experts can be evaluated on compressed CPU activity traces. The compression is similar to run-length encoding and it can be performed while executing the adaptive policy (Figure 7).

Time period between 0.0 and 15.6 ms:

| | | | |
|---|---|---|---|
| 4 | 1 | 152 | 0 |

Time period between 15.6 and 31.2 ms:

| | | | |
|---|---|---|---|
| 4 | 1 | 152 | 0 |

Time period between 31.2 and 46.9 ms:

| | | | |
|---|---|---|---|
| 5 | 1 | 152 | 0 |

Time period between 46.9 and 62.5 ms:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 39 | 0 | 10 | 1 | 6 | 0 | 4 | 1 | 93 | 0 |

Time period between 62.5 and 78.1 ms:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 37 | 0 | 6 | 1 | 8 | 0 | 7 | 1 | 2 | 0 |
| 19 | 1 | 1 | 0 | 9 | 1 | 1 | 0 | 27 | 1 | 35 | 0 |

Time period between 78.1 and 93.7 ms:

| | | | |
|---|---|---|---|
| 4 | 1 | 152 | 0 |

Time period between 93.7 and 109.4 ms:

| | | | |
|---|---|---|---|
| 5 | 1 | 152 | 0 |

Time period between 109.4 and 125.0 ms:

| | | | |
|---|---|---|---|
| 4 | 1 | 152 | 0 |

Figure 7: Run-length encoding of the CPU pattern from Figure 1. The encoded symbols 0 and 1 represent idle and active CPU states, respectively. The original trace is 1,250 samples long. The compressed trace is expressed by 68 integers. This represents almost a 20-fold compression ratio.

Since most parts of our CPU activity traces are primarily idle or active, run-length encoding reduces their length to almost 2 percent of the original size. Therefore, we believe that the overhead of simulating the four timeout experts in practice is much less than the initial estimate of 400 percent. Moreover, note that all computational steps in Equations 2, 3, and 4 can be implemented using integer arithmetic (Press *et al.* 1992).

## Conclusions

Recent advances in hardware and wireless technologies have made mobile computers ubiquitous in our daily life. Extending their battery life has naturally become a major challenge. This paper discussed the domain of fine-grained power management and illustrated how machine learning methodology can applied in this field. In addition, we proposed a practical adaptive solution for a real-world PM problem. The solution is conceptually simple and yields better results than state-of-the-art baselines in our domain.

Although our results are Intel Core Duo CPU specific, we believe that they generalize to other mobile and desktop processors. The proposed adaptive solution is simple enough to run on a microcontroller, and sufficiently fast to be executed within less than a millisecond. Our future goal is to validate its benefits by implementing it on the hardware level.

Finally, since our domain allows for a greedy computation of ideal adaptive timeout policies, we know that our current solutions are far from being optimal. In our future work, we plan to investigate other machine learning methods that may further improve the residency and latency of these solutions.

## Acknowledgment

We thank anonymous reviewers for comments that led to the improvement of this paper. We also thank Paul Diefenbaugh of Intel Corporation for his valuable advice on package performance and power issues.

## References

Benini, L.; Bogliolo, A.; Paleologo, G.; and Micheli, G. D. 1999. Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18(6):813–833.

Chung, E.-Y.; Benini, L.; and de Micheli, G. 1999. Dynamic power management using adaptive learning tree. In *Proceedings of the 1999 IEEE / ACM International Conference on Computer-Aided Design*, 274–279.

Dhiman, G., and Simunic, T. 2006. Dynamic power management using machine learning. In *Proceedings of the 2006 IEEE / ACM International Conference on Computer-Aided Design*.

Golding, R.; Bosch, P.; and Wilkes, J. 1996. Idleness is not sloth. Technical Report HPL-96-140, Hewlett-Packard Laboratories.

Gramacy, R.; Warmuth, M.; Brandt, S.; and Ari, I. 2003. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems 15*, 1465–1472.

Helmbold, D.; Long, D.; Sconyers, T.; and Sherrod, B. 2000. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications* 5(4):285–297.

Herbster, M., and Warmuth, M. 1995. Tracking the best expert. In *Proceedings of the 12th International Conference on Machine Learning*, 286–294.

Hwang, C.-H., and Wu, A. 1997. A predictive system shutdown method for energy saving of event-driven computation. In *Proceedings of the 1997 IEEE / ACM International Conference on Computer-Aided Design*, 28–32.

Karlin, A.; Manasse, M.; McGeoch, L.; and Owicki, S. 1994. Competitive randomized algorithms for nonuniform problems. *Algorithmica* 11(6):542–571.

Press, W.; Teukolsky, S.; Vetterling, W.; and Flannery, B. 1992. *Numerical Recipes in C*. Cambridge, MA: Cambridge University Press.

Simunic, T. 2002. Dynamic management of power consumption. In *Power Aware Computing*. New York, NY: Kluwer Academic Publishers.

Srivastava, M.; Chandrakasan, A.; and Brodersen, R. 1996. Predictive system shutdown and other architectural techniques for energy effcient programmable computation. *IEEE Transactions on Very Large Scale Integration Systems* 4(1):42–55.