Adaptive Computation Offloading for Energy Conservation on Battery-Powered Systems

Changjiu Xian
Department of Computer Science
Purdue University
West Lafayette, Indiana
cjx@cs.purdue.edu

Yung-Hsiang Lu
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana
yunglu@purdue.edu

Zhiyuan Li
Department of Computer Science
Purdue University
West Lafayette, Indiana
li@cs.purdue.edu

Abstract

This paper considers the problem of extending the battery lifetime for a portable computer by offloading its computation to a server. Depending on the inputs, computation time for different instances of a program can vary significantly and they are often difficult to predict. Different from previous studies on computation offloading, our approach does not require estimating the computation time before the execution. We execute the program initially on the portable client with a timeout. If the computation is not completed after the timeout, it is offloaded to the server. We first set the timeout to be the minimum computation time that can benefit from offloading. This method is proved to be 2-competitive. We further consider collecting online statistics of the computation time and find the statistically optimal timeout. Finally, we provide guidelines to construct programs with computation offloading. Experiments show that our methods can save up to 17% more energy than existing approaches.

1. Introduction

Extending the battery lifetime is one of the most critical problems in portable computing. Many power conservation techniques have been proposed from hardware level to system level [1]. Most techniques focus on predicting the idleness in the workloads such that the hardware components can be shut down or slowed down to save energy. On the other hand, since the idleness is ultimately determined by the application programs, energy-aware programs can adjust their workloads to create more opportunities for power management. Several studies [2,4,16] on energy-aware programs have considered rescheduling or reducing workloads as the tradeoff for energy savings. These studies assume the programs run on the portable device locally.

In a client-server distributed computing environment, the programs on the portable device can also offload their computation to the server to conserve energy [8, 9, 13–15]. The server is assumed to be grid-powered. This consumes the server's power to extend the battery life of the portable client. Computation offloading reduces the workload on the client's processor but it needs to use the client's network interface to send the needed data and receive the result. Consequently, computation offloading can save energy only if the communication energy is less than the computation energy. In this paper, we use the terms *portable*, *client*, and *local* interchangeably.

To determine whether to offload and which portion of the computation to offload, the energy consumption needs to be estimated before the execution. The communication energy depends on the size of the transmitted data and the network bandwidth; the computation energy depends on the computation time. Existing studies assume that both the data size and the computation time can be obtained before the execution. The data size is relatively easy to obtain because the input data are available before execution and the results are often in predefined sizes. The computation time is more difficult to determine because it can vary significantly for different instances of the computation [10, 17]. The size of the input data does not necessarily determine the execution time because it may also depend on the content of the data. Previous studies have used the worst-case or the average execution time of the past instances as the future execution time. The inaccuracy of these methods can be large. Studies in [6, 17] show that it is also ineffective to use the execution instance in the near past or a moving average to predict the future instances.

In this paper, we present an approach of offloading that does not require estimating the computation time before

978-1-4244-1890-9/07/$25.00 ©2007 IEEE

the execution. We first calculate the minimum computation time that can benefit from offloading. This is called the break-even time of offloading. We then propose an adaptive method that uses the break even time as the timeout for the client's computation. The computation is offloaded to the server only if it is not completed after the break-even time. The advantage of this method is that the instances with short computation time are executed at the client and the instances with large computation time are offloaded to the server. The overhead of this method is that the timeout wastes energy for the computation instances that are eventually offloaded. This method is proved to be 2-competitive. To find a better tradeoff between the overhead and the energy savings, we collect the computation time online to find the statistically optimal timeout. We also provide the guidelines to construct energy-aware programs with the adaptive offloading scheme. Experiments show that our methods can improve the energy savings significantly over existing approaches.

2. Related Work

There have been several studies on computation offloading. Rudenko et al. [14] present an automation framework for computation offloading and it records the average power consumption of a repetitive task for deciding whether to offload the task. Kremer et al. [8] propose an offloading scheme that uses check-pointing techniques to handle disconnection events for wireless connection. The cost model for local computation is based on the average computation time. Rong et al. [13] study offloading under real-time constraints. They use multiple synthetic tasks and each task has a known constant computation time. Li et al. [9] propose making offloading decisions at function level. The computation of each function is assumed to be a constant and obtained by profiling. Wang et al. [15] propose a method of parametric compiler analysis to determine the computation time. The method considers only simple parameters, such as the command line options. It cannot analyze more complex data such as an image. All these methods require estimating the computation time before execution in order to make offloading decisions. In contrast, we use a timeout method and do not require such estimation.

Timeout is a popular power management policy and has been well studied [7,11]. However, all previous studies consider timeout for reducing only the energy consumption during idle periods. If the idle period is longer than the timeout, the hardware component is shut down to save energy. In our study, a timeout is set for computation instead of an idle period. If the computation is longer than the timeout, the computation is offloaded to a remote server to conserve the energy for the client.

Table 1. Parameters for offloading.

symbol	meaning
p_a, p_l	active power, idle power of processor
p_{tx}, p_{rx}	transmission power, receiving power (including network interface and processor)
p_n	idle power of the network interface
r_{tx}, r_{rx}	transmission and receiving speed
t_c, t_s	execution time on the client and on the server for the same computation
d_{in}, d_{out}	size of input and result

3. Adaptive Computation Offloading

This section presents our scheme of computation offloading. We present a 2-competitive timeout method for computation offloading in Section 3.1. Section 3.2 improves this method by considering online statistics of the computation time. In Section 3.3, we provide guidelines for constructing energy-aware applications with computation offloading.

3.1. Two-Competitive Timeout

We define the break-even time of offloading as the minimum execution time for the computation to benefit from offloading. The intuition is to let the short computation instances execute on the client and the others execute on the server. We will prove that this method is 2-competitive considering the timeout overhead.

In order to compute the break-even time, we formulate the energy consumption for executing the computation on the client and for offloading it to the server. The parameters used for the formulation are listed in Table 1. Without offloading, the program consumes energy $p_a t_c$ on the local processor. The network interface is not needed so it is turned off and consumes no energy. For offloading, the client's energy consumption consists of four parts: the energy for transmitting the input data $p_{tx}\frac{d_{in}}{r_{tx}}$, the energy for receiving the result $p_{rx}\frac{d_{out}}{r_{rx}}$, the processor's idle energy during remote computation $p_l t_s$, and the network's idle energy during remote computation $p_n t_s$. For offloading to break-even, we must have

$$p_a t_c = p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}} + p_l t_s + p_n t_s \qquad (1)$$

The relation between t_c and t_s depends on the difference of the computing capabilities of the client and the server. Previous study in [3] show that the relation can be expressed as $t_c = \alpha t_s$, where α is a constant for the given client and server. A larger α means a faster server. Equation (1) can

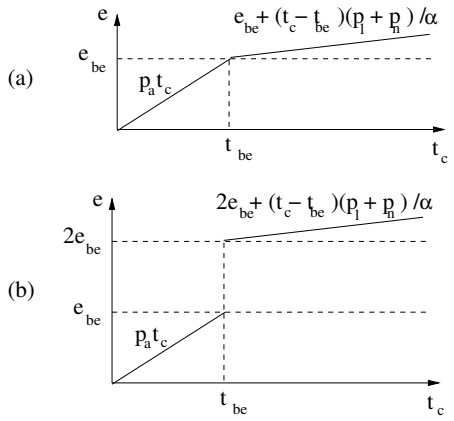


Figure 1. (a) The energy of the oracle method. (b) The energy of the timeout method with t_{be}.

be rewritten as the following.

$$p_a t_c = p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}} + p_l\frac{t_c}{\alpha} + p_n\frac{t_c}{\alpha} \quad (2)$$

We solve the equation for t_c and obtain the break-even time as follows.

$$t_{be} = \frac{p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}}}{p_a - \frac{p_l}{\alpha} - \frac{p_n}{\alpha}} \quad (3)$$

Equation (3) shows that the break-even time of offloading depends on the data sizes d_{in} and d_{out}. This is different from the traditional break-even time for shutting down an I/O device during idle periods. Since the shutdown occurs after the data have been processed and cleared from the device, the shutdown only needs the device driver to save the set of registers of the device and this takes a small amount of constant time. The traditional break-even time thus depends on solely hardware parameters. Equation (3) requires that $p_a - \frac{p_l}{\alpha} - \frac{p_n}{\alpha} > 0$ because the break-even time should be a finite and positive number. This constraint can be transformed as follows.

$$\alpha > \frac{p_l + p_n}{p_a} \quad (4)$$

This means that the server should have sufficient computing capability relative to the client; otherwise, computation offloading cannot save energy.

A brief derivation of the competitiveness of our timeout method is described as follows. Figure 1 (a) shows the energy consumption of the oracle method as a function of execution time t_c. When $t_c \leq t_{be}$, the energy consumption is equal to the left-hand side of Equation (2). When $t_c > t_{be}$, the energy consumption is equal to the right-hand side of Equation (2). Let e_{be} be the break-even energy, we have

$$e_{be} = p_a t_{be} = p_l\frac{t_{be}}{\alpha} + p_n\frac{t_{be}}{\alpha} + p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}} \quad (5)$$

Based on equations (2) and (5), the energy consumption of the oracle method for $t_c > t_{be}$ can be expressed as $e_{be} + (t_c - t_{be})(p_l + p_n)/\alpha$.

Figure 1 (b) shows the the energy consumption of the timeout method. The only difference of this method from the oracle method is that the execution instance with $t_c > t_{be}$ always consume extra energy $p_a t_{be}$ for the timeout. The energy consumption is the same as the oracle method when $t_c \leq t_{be}$. When $t_c > t_{be}$, the energy consumption is $2e_{be} + (t_c - t_{be})(p_l + p_n)/\alpha$.

The energy ratio between the timeout method and the oracle method equals to 1 when $t_c \leq t_{be}$, as explained earlier. When $t_c > t_{be}$, the ratio is a function of t_c as follows.

$$\frac{2e_{be} + (t_c - t_{be})(p_l + p_n)/\alpha}{e_{be} + (t_c - t_{be})(p_l + p_n)/\alpha} \quad (6)$$

When $t_c \to t_{be}$, the ratio is 2. When $t_c \to +\infty$, the ratio approaches to 1. The maximum ratio is 2, so we have proved that the timeout method is 2-competitive.

3.2. Statistically Optimal Time-Out

The execution time for different instances of computation depends on the input data and can vary significantly. The instantaneous execution times are difficult to predict but their statistical distribution are often more stable and change slowly, as has been observed in previous studies [10, 17]. In this section, we use the statistical information to find the optimal timeout to minimize the expected energy of the client.

The statistical information we collect is a discrete cumulative distributed function (CDF) of the execution time for the past execution instances. The range between 0 and the worst-case execution time is divided into a finite number of intervals, as shown in Figure 2. The total number of intervals is n and the t_0, t_1, ..., t_n are the interval boundaries. For simplicity, we use equal-length intervals in this paper and the length of each interval is δ_t. If an execution instance takes time t, it is mapped to the i^{th} interval with $i = \lceil t/\delta_t \rceil$. We use Ψ to denote the discrete CDF. The probability that the computation consumes the i^{th} interval is $1 - \Psi(i-1)$. Note that $\Psi(0) = 0$.

With the statistical information, we can find a better timeout to minimize the expected total energy of the client. Let x denote the number of intervals for the optimal timeout. The expected total energy consumption consists of three parts. The first part is the expected energy for the timeout. Since the first x intervals are executed in the client, the expected energy is $\sum_{i=1}^{x}(1 - \Psi(i-1))\delta_t(p_a + p_n)$. The second part is the expected energy of sending data and receiving results for offloading. The probability that the offloading occur is $1 - \Psi(x)$, so the expected communication energy is $(1 - \Psi(x))(p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}})$. The third

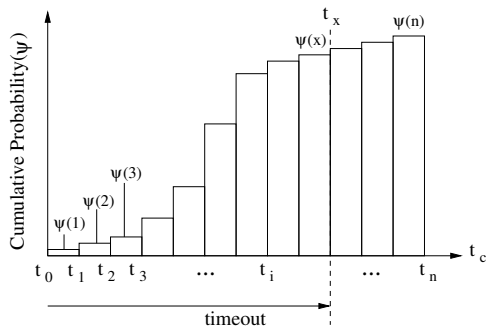


Figure 2. The discrete cumulative probability for execution time. Each interval is equal to δ_t. With a timeout of t_x, the probability that the offloading occurs is $1 - \Psi(x)$.

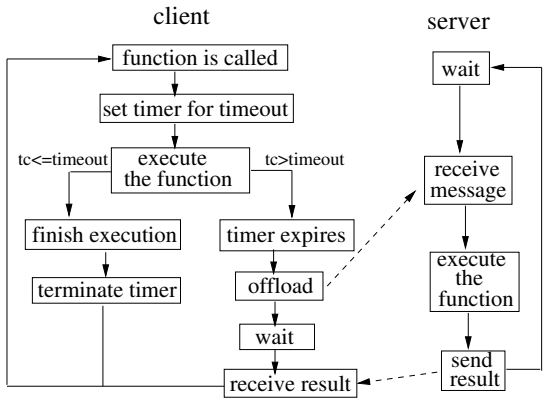


Figure 3. Adaptive computation offloading.

part is the idle energy consumption of the processor and the network interface during the remote execution. Since x is the timeout and the offloading occurs with a probability of $1 - \Psi(x)$, the probability for the intervals with $i \leq x$ to execute in the server is $1 - \Psi(x)$. For intervals with $i > x$, the probability for them to execute in the server is $1 - \Psi(i-1)$. Hence, the third part energy is $\frac{((1-\Psi(x))x\delta_t + \sum_{i=x+1}^{n} \delta_t(1-\Psi(i-1)))}{\alpha}(p_l + p_n)$

Overall, to minimize the total energy of the client, the problem can be formulated as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=1}^{x}(1-\Psi(i-1))\delta_t(p_a+p_n) \\ & +(1-\Psi(x))\left(p_{tx}\frac{d_{in}}{r_{tx}} + p_{rx}\frac{d_{out}}{r_{rx}}\right) \\ & +\left((1-\Psi(x))x\delta_t + \sum_{i=x+1}^{n}\delta_t(1-\Psi(i-1))\right)\frac{p_l+p_n}{\alpha} \end{aligned} \tag{7}$$

The possible values of x are finite and within the set $\{0, 1, 2, ..., n\}$. We can perform a simple scan to find the optimal x for the minimization problem. We assign each of the values in the set to x and calculate the total expected energy using Equation (7). The x's value that results in the least energy consumption is the optimal timeout. The complexity of this method is $O(n^2)$ because the calculation of Equation (7) for each given value of x takes time $O(n)$. If n is large, solving Equation (7) may consume significant time and energy. However, our sensitivity analysis (Section 4.2) shows that $n = 20$ can capture 99% of the energy savings. The time to solve the above equation is about 700 microseconds and it is small compared with typical programs' computation time (from hundreds of milliseconds to seconds). This allows us to calculate the timeout at runtime to adapt to the change of the statistics.

If $x = 0$, we should offload the computation from the beginning. If $x = n$, no offloading is necessary. We should otherwise offload the computation after a timeout of $x\delta_t$.

3.3. Application Construction

In this section, we provide guidelines to construct energy-aware applications with computation offloading. Applying computation offloading technique to a program requires restructuring the program to support the functionalities needed by offloading. We assume the computation to offload is a function of the program, repeatedly called during execution of the program. Figure 3 illustrates the major steps of our adaptive offloading technique. When the function is called, a timeout is set before executing the function. If the actual execution time t_c is shorter than the timeout, the function completes before the timer expires. The timer should be terminated since no offloading is needed. If t_c is longer than the timeout, the timer will expire during the execution. When this occurs, the computation is offloaded to the server. The client aborts the local computation and sends the input data to the server. The server then executes the same function and sends the result back to the client.

Figure 4 illustrates a more detailed program structure for the client side. Figure 4 (a) shows the original function `foo` and Figure 4 (b) shows the restructured `foo`. It first computes the timeout based on the data sizes d_{in}, d_{out} and the CDF of execution time, using the method explained in Section 3.2. The programmer should obtain the data sizes through either manual analysis or any automated program analysis tool. If the computed timeout is longer than the maximum execution time, we execute the computation locally. If the timeout is shorter than the minimum execution time, the computation is always offloaded. We otherwise

```
(a)  foo (input) {
       execute the computation
       return output
     }

(b)  foo (input) {
       /* from Section 3.2 */
       timeout = calc_timeout (d_in, d_out, cdf)
       if (timeout >= max(tc) ) {
          output = local (input)
       }else if (timeout <= min(tc) {
          output = offload (input)
       }else{
          tid = start_thread (local, input)
          start_timer (timer_handler, timeout, tid)
          output = join_thread (tid)
          if (thread tid is cancelled)
             output = offload (input)
       }
       return output
     }
     timer_handler( ) {
       if (thread tid not joined yet)
          cancel_thread (tid)
     }
```

Figure 4. (a) The original function. (b) The re-structured function.

need to perform timeout. This can be realized by using a timer and two threads, as supported in many programming languages. A separated thread is launched to execute the computation such that we can timeout the computation by simply canceling the thread.

We record the execution time to build CDF at runtime to adapt to the change of statistics. When the computation is executed on the client, we simply insert codes before and after the computation to measure the execution time and add it to the CDF. We also count the offloaded computation instances for the CDF. To count an offloaded instance, the server side measures the execution time t_s and sends it back to client along with the computation result. The client then converts t_s to t_c ($t_c = \alpha t_s$) and add it to the CDF.

The overhead of using the timer and threads, building CDF, and computing the timeout is about 3 milliseconds based on our setup and measurement (Section 4.2). Any computation shorter than 3 milliseconds should not be considered to use this offloading technique.

4. Experiments

We have implemented our offloading scheme in Linux. In this section, we evaluate the energy savings, the performance impact, and the overhead of the scheme by comparing with existing approaches.

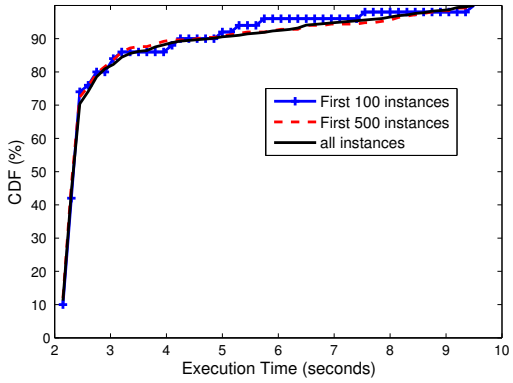


Figure 5. The CDF for the first 100, 500, and all execution instances.

4.1 Experimental Setup

Our experiments are performed in the Integrated Development Platform (IDP) by Accelent Systems running Linux 2.4.18. The IDP uses Intel PXA250 as the processor and provides probing points to measure the power consumption of different hardware components. We install an Orinoco wireless card for network communication, and it supports the protocol 802.11. To measure the power, we use another computer with a data acquisition card (DAQ) from National Instruments. The processor and the wireless card on IDP are connected to the DAQ for energy measurement. The server used for offloading is a 3GHz Pentium 4 machine with 4 GB RAM. The measured maximum available bandwidth for the client to transmit to and receive from the server is 810 KB/s and 379 KB/s, respectively.

Our evaluation uses image processing as the workload. Image processing and understanding will be one of the key applications for low-power mobile devices [8]. Image processing can be used in the context of robot control and navigation, guidance systems for blind people, autonomic vehicles, target acquisition and classification. Stereo image correspondence is an important technology for several of these applications. A stereo algorithm processes a pair of stereo images to produce a disparity map and it uses the gray-scale to indicate the depth of the pixels. We use the stereo algorithm in [12] and the stereo-vision images are taken from the image database in [5].

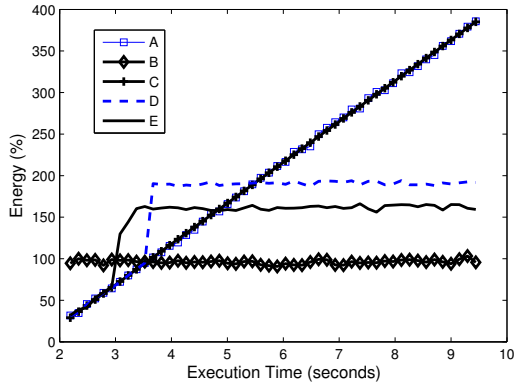
The processing for each pair of stereo images is one computation instance for the image correspondence program. Figure 5 shows the CDF of execution time for the first 100, 500, and all 1000 instances. The three lines are close to each other and suggest that the CDF is stable and changes slowly. This is consistent with the observation in previous

study [17]. We also measured the ratio α between the client (the IDP) and the server using the image correspondence program with 500 pairs of images. The results show that the ratio α has an average of 71.52 and the standard deviation is 1.67. Since the variation is small, we use the average for α in our method.

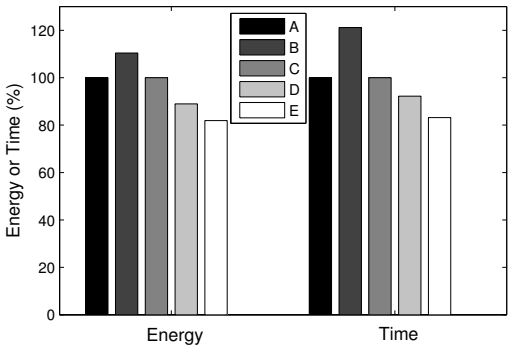
We consider five methods: A- always execute the computation locally, B- always offload the computation from its beginning, C- perform offloading if the average computation time is longer than the break-even time, D- execute the computation up to the break-even time and, if the computation hasn't completed yet, offload it to the server, E- use the statistics of computation time to find a better time-out. Method C represents the common aspect of the previous studies [8, 9, 13, 14] because all of them assume the computation time of a function or a task is a known constant or use the average computation time based on offline profiling. These studies perform offloading with different constraints, such as the events of network disconnection [8], the data dependencies among different tasks [9], the scheduling of multiple real-time tasks [13], and the operating system's ability to automate offloading [14]. We do not consider these constraints and focus on varied computation time. We do not compare with the study in [15] because its parametric compiler analysis cannot analyze complex data such as the images used in our workload. Methods D and E are our methods and explained in Section 3.1 and Section 3.2, respectively.

4.2 Energy and Latency

Figure 6 (a) shows the average energy consumption of the computation instances as a function of the execution time. The size of the input data is 600KB and the size of the result is 300KB. The average energy of each method is normalized to method *A* at the break-even point, 3.63 second. The figure shows that the energy of method *A* is proportional to the execution time because there is no offloading. Method *B* is almost a constant because it always offload the computation from the beginning and the major energy consumption is for communication. The local idle energy during remote processing is small because the remote processing time is less than 150 ms with $\alpha = 71.52$. The average execution time on the client is 3.2 second and this is shorter than the break-even time, so method *C* executes in the client and its energy consumption is the same as method *A*. method *D* uses the break-even time as timeout, so its energy consumption before 3.63 second is the same as method *A*. After 3.63 second, method *D* offloads the computation to the server and the energy is the break-even energy plus the energy of method *B*. Method *E* finds the optimal timeout is about 3 seconds. This timeout is shorter than the break-even time and pays less timeout overhead, so *E* consumes less



(a)



(b)

Figure 6. (a) The average energy consumption of the instances as a function of execution time, normalized to the break-even energy. (b) The total energy consumption of all instances, normalized to method *A*.

energy for the instances longer than the break-even time.

Figure 6 (b) shows the total energy and time of all instances for the five methods. Method *B* consumes 12% more energy than *A*. Compared to *A* or *C*, our methods *D* and *E* save 11% and 17%, respecitvely. The normalized time consumption of the five methods is similar to the energy consumption so we omit the description.

The above results are for input images of dimension 640x480 in pixels. To evaluate the methods over different data sizes, we also tested dimensions 480x320 and 320x240. Figure 7 (a) shows that the break-even time are decreasing for smaller data sizes. The statistically optimal timeout found by method *E* is different from the break-even time and also decreases. Figure 7 (b) shows that our method still saves significant amounts of energy for smaller data sizes. The reason is that the both execution time and communication time decrease as the data size decreases and the

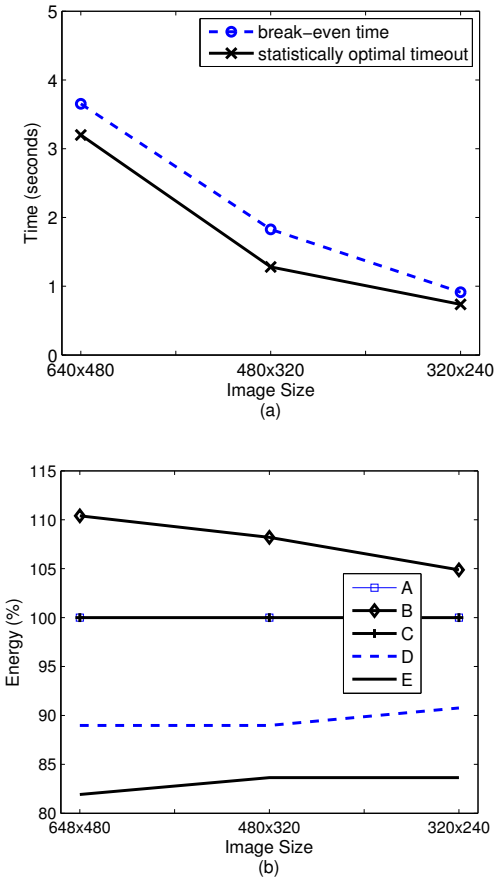


Figure 7. (a) The break-even time and statistically optimal timeout. (b) The energy of all methods normalized to method A.

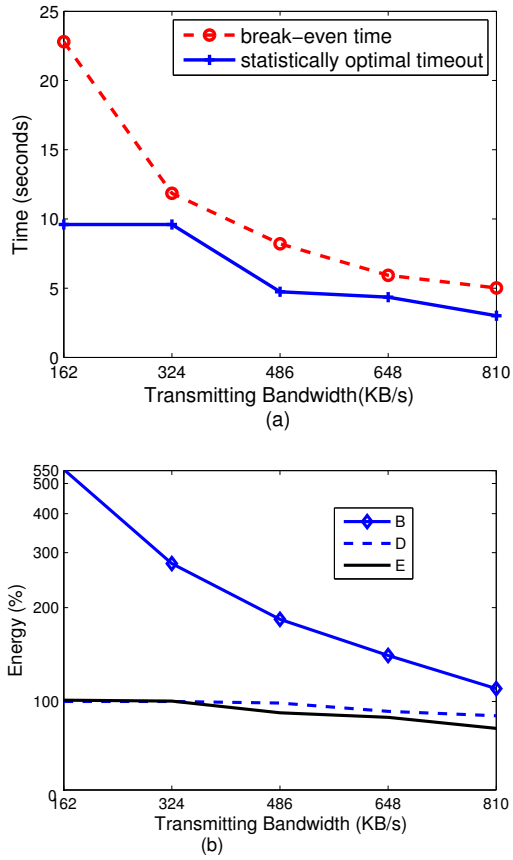


Figure 8. (a) The break-even time and statistically optimal timeout. (b) The energy of all methods normalized to method A whose energy is independent of bandwidth and omitted in the figure. Method C consumes the same energy as A and is also omitted.

tradeoff still exists between the communication energy and computation energy.

Another important factor affecting the offloading decision is the bandwidth of the network channel. The bandwidth can change in different environments. In order to evaluate the methods under different bandwidth, we intentionally adjust the bandwidth for the client. Since we are unable to increase the bandwidth, we evaluate only the cases where the bandwidth is lower. To reduce the bandwidth, we simply slow down receiving and sending in the server side. We decrease the bandwidth to 80%, 60%, 40%, and 20% of the maximum available bandwidth. Figure 8 (a) shows that the break-even time and the optimal timeout decreases as the bandwidth increases. When the transmitting bandwidth is less than 324 KB/s, the break-even time is longer than the worst-case execution time. This suggests that offloading has no benefit at all. With such a low bandwidth, the statistically optimal timeout found by method E is equal to the worst-case execution time and thus method E executes all instances on the client. Figure 8 (b) shows the energy of the four methods over the bandwidth. Method A is omitted in the figure because it executes computation on the client and its energy is not affected by bandwidth. Our method E always consumes the least energy among all the methods.

Method E uses discrete CDF and the granularity of the CDF is determined by the number of intervals used to represent the CDF. Figure 9 shows that method E can reduce more energy with more fine-grained CDF. Three images sizes (640x480, 480x320, and 320x240) are tested and they all show that when the number of intervals are more than 20, the energy consumption does not further decease. Based on this, we use 20 intervals in our implementation.

Finally, we evaluate the overhead of our adaptive offloading method. The measurements show that recording of the

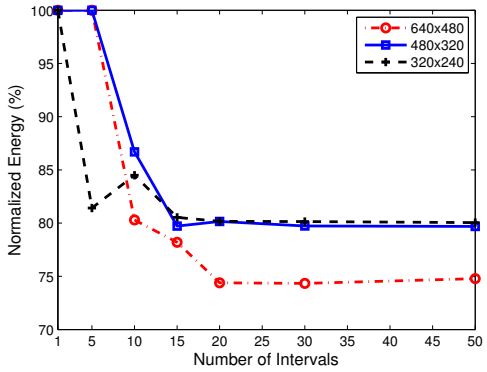


Figure 9. The energy consumption of method E varies with the number of intervals used for the CDF. The energy is normalized to the case where a single interval is used.

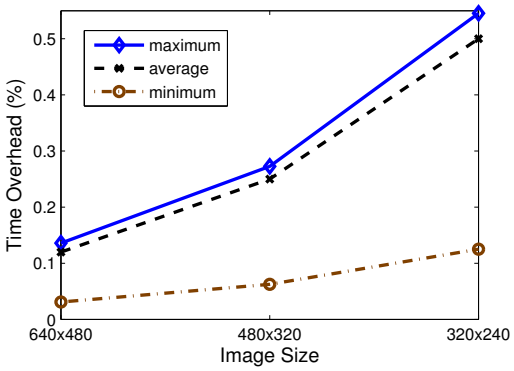


Figure 10. The percentage of time overhead for adaptive offloading.

execution time takes about 50 microseconds, computing the timeout takes about 700 microseconds, launching the thread takes about 1900 microseconds, and executing the timer handler takes about 300 microseconds. Overall, the overhead is about 3 milliseconds. Figure 10 shows the maximum, minimum, and average percentage of the time overhead with respect to the execution time without offloading. Three different images sizes are examined. The overhead is no more than 0.55%. The figure for the energy overhead is similar and omitted.

5. Conclusions

This paper presents an adaptive method for computation offloading to save energy on the client. The method uses timeout and does not require estimating the execution time for each computation instance. Online statistics of computation time is used to compute optimal timeout. The experiments show that our method can improve the energy savings considerably with small overhead.

6 Acknowledgments

This work is supported in part by National Science Foundation CNS-0347466 and CCF-0541267. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] L. Benini and G. D. Micheli. System-Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, April 2000.

[2] L. Cai and Y.-H. Lu. Dynamic Power Management Using Data Buffers. In *Design Automation and Test in Europe*, pages 526–531, 2004.

[3] E.-Y. Chung, G. D. Micheli, and L. Benini. Contents Provider-Assisted Dynamic Voltage Scaling for Low Energy Multimedia Applications. In *International Symposium on Low Power Electronics and Design*, pages 42–47, 2002.

[4] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *ACM Symposium on Operating Systems Principles*, pages 48–63, 1999.

[5] S. Gautam, G. Sarkis, E. Tjandranegara, E. Zelkowitz, Y.-H. Lu, and E. J. Delp. Multimedia for Mobile Users: Image Enhanced Navigation. In *Multimedia Content Analysis, Management, and Retrieval, IST/SPIE Symposium on Electronic Imaging*, 2006.

[6] D. Grunwald, P. Levis, C. B. M. III, M. Neufeld, and K. I. Farkas. Policies for Dynamic Clock Scheduling. In *Symposium on Operating System Design and Implementation*, pages 73–86, 2000.

[7] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Nonuniform Problems. *Algorithmica*, 11(6):542–571, June 1994.

[8] U. Kremer, J. Hicks, and J. Rehg. A Compilation Framework for Power and Energy Management on Mobile Computers. In *Interational Workshop on Languages and Compilers for Parallel Computing*, pages 1–12, 2001.

[9] Z. Li, C. Wang, and R. Xu. Computation Offloading to Save Energy on Handheld Devices: a Partition Scheme. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 238–246, 2001.

[10] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 50–61, 2001.

[11] Y.-H. Lu, E.-Y. Chung, T. Simunic, L. Benini, and G. D. Micheli. Quantitative Comparison of Power Management Algorithms. In *Design Automation and Test in Europe*, pages 20–26, 2000.

[12] A. Redert, E. Hendriks, and J. Biemond. Correspondence Estimation in Image Pairs. *IEEE Signal Processing Magazine*, 16(3):29–46, May 1999.

[13] P. Rong and M. Pedram. Extending The Lifetime of A Network of Battery-powered Mobile Devices by Remote Processing: A Markovian Decision-based Approach. In *Design Automation Conference*, pages 906–911, 2003.

[14] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning. The Remote Processing Framework for Portable Computer Power Saving. In *ACM Symposium on Applied Computing*, pages 365–372, 1999.

[15] C. Wang and Z. Li. Parametric Analysis for Adaptive Computation Offloading. In *Conference on Programming Language Design and Implementation*, pages 119–130, 2004.

[16] C. Xian and Y.-H. Lu. Energy Reduction by Workload Adaptation in a Multi-Process Environment. In *Design Automation and Test in Europe*, pages 514–519, 2006.

[17] W. Yuan and K. Nahrstedt. Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems. In *ACM Symposium on Operating Systems Principles*, pages 149–163, 2003.