Ranking Servers based on Energy Savings for Computation Offloading

Karthik Kumar, Yamini Nimmagadda, and Yung-Hsiang Lu
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907
{kumar25,ynimmaga,yunglu}@purdue.edu

ABSTRACT

Offloading may save energy for battery-powered devices by migrating computation to grid-powered servers. Offloading can be provided as a service and the servers charge the devices' users based on the consumed resources. In this paper, we propose a scheme to rank the servers based on the amounts of energy savings. The ranking depends on two factors: (1) the energy saved due to offloading and (2) the energy consumed while waiting for the results. We instrument the offloaded programs to estimate the amounts of computation performed by the servers, and use this information to determine the amounts of saved energy. When the servers perform the offloaded computation, the battery-powered devices wait for the results and consume energy. The ratio of the two factors determines the rank of a server. If a server performs more computation within a shorter duration, the server is ranked higher. We implement our method on an HP iPAQ and demonstrate that our method can effectively rank servers based on energy savings.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; C.4 [**Performance of Systems**]: Design studies

General Terms

Algorithms, Design, Measurement, Performance

Keywords

computation offloading, ranking servers, energy savings

1. INTRODUCTION

Offloading can save energy in battery-powered devices (also called clients) by migrating computation from these devices to grid-powered servers. These servers may be orders of magnitude faster than the clients. The computation is performed on these servers and the results are returned to the devices. The servers have no knowledge about the computation and merely provide general platforms for offloading. The results of the computation are sent to the clients. The clients do not know the results; otherwise, offloading would be meaningless. The servers may charge the devices' users for the resources consumed, thus providing utility computing, or offloading as a service. Several papers [1, 2, 3, 4, 5] discuss infrastructures for offloading as a service.

Different servers may perform different amounts of computation within the same duration due to the servers' speeds, loads, scheduling constraints, etc. A server may charge a client based on the amounts of resources used. For example, a previous study [6] suggests using energy consumption to charge users based on a bidding procedure. For some applications, the offloaded computation has no single correct answer. Instead, the amounts of computation affect the degree of accuracy of the results. Such applications include text and multimedia retrieval [7, 8], Monte Carlo methods [9], and portfolio rebalancing [10]. To make offloading service practical, a client has to determine whether the server has indeed performed the computation as claimed. Otherwise, a server may overcharge a client for the computation that is not performed. In other words, the client must be able to estimate the amount of computation performed by the server *without* actually executing the program.

This paper presents a method to rank servers based on the amount of energy saved for the client. The rank is determined based on two factors: (1) the amount of energy saved by the client through offloading computation and (2) the energy consumed by the client while waiting for the results. The ratio of the two terms determines which server is more effective for saving the client's energy; a larger ratio is desirable. Even though the second term can be easily obtained by the client, acquiring the first term is more difficult because the client does not perform the computation. Instead, the client inserts additional code that can be used to determine how much work is performed by a server. The added code must meet the following requirements: (a) The overhead is low so that it does not substantially increase the execution time of the offloaded computation. (b) The code is independent of a server's architecture so that every server can execute. (c) The code provides information about the amount of work performed by a server. (d) The code is embedded inside the original program and difficult to detect by a server. If the code can be easily detected by a server, the server may overcharge the client by executing this code without running the rest of the program.

We insert radix-graph counters in offloaded programs in

ISLPED'09, August 19–21, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-684-7/09/08 ...$10.00.

our method. Radix graph counters increment numbers using pointers and meet all the four requirements mentioned earlier. The counters' values are returned to the client as part of the result. We implement this method on an HP PDA and execute an image-processing program on three different servers. Our experiments show that the method can accurately rank the service from these different servers.

2. RELATED WORK

Computation offloading may reduce the energy consumption of battery-powered systems. Previous studies can be classified into two categories: (1) computation offloading, and (2) software watermarking.

2.1 Computation Offloading

Offloading can be achieved using .NET remoting and Java RMI (remote method invocation). Many researchers [3, 4, 5] propose infrastructures for offloading as a service and mechanisms for selecting servers. Anagnoustou et al. [1] present a matching algorithm and an architecture for service discovery. Sivavakeesar et al. [2] propose a mechanism for discovering suitable service. Zhang et al. [11] propose a user-oriented Quality of Service model to measure whether services are suitable for being delivered to mobile users. Several studies consider the decision whether to offload computation [12, 13, 14]. This paper does not consider offloading decisions; instead, we assume that the client has decided to offload. We propose a method for ranking servers based on the amounts of energy savings on the client.

2.2 Software Watermarking

The inserted code must be difficult for a server to identify. A similar problem has been studied for software watermarking. Software watermarking hides information among code in order to identify copyrighted software. The information may use pointers to hide graphs since pointer analysis is NP-complete [15]. Collberg et al. [16] use a radix graph to hide a watermark in programs. When a program is executed with a certain input, the hidden watermark is revealed to identify the software as copyrighted. This paper uses a similar technique by embedding radix-graph counters inside offloaded programs. These counters are difficult for a server to detect. When the offloaded programs complete, the counters' values are returned as parts of the results. The client extracts the counters' values to determine the amounts of work performed by the server.

Our previous work [17] proposes a protocol for a client to detect whether a server actually performs offloaded computation. This paper solves a different problem by comparing the amounts of energy saved by offloading. This paper has the following contributions: (1) We propose to rank servers based on the amounts of energy saved for clients. (2) We present graph-based counters to quantify the amount of computation performed by a server. (3) We describe how to accurately obtain the amount of computation when handling multi-level code segments. (4) We implement our method on an HP PDA and demonstrate that our approach is effective in ranking servers.

3. ESTIMATING THE ENERGY SAVINGS

The amount of saved energy depends on the amount of computation performed by the server. The client can only observe the result of the computation and the time taken to return the result. We provide a method for the client to estimate the amount of computation done by the server. Section 3.1 provides a motivating example for our method. Sections 3.2 and 3.3 provide the details of how we obtain the amount of computation. Sections 3.4 and 3.5 describe how we use the amount of computation to estimate the energy savings and to rank servers.

3.1 Motivating Example

We show a simple example to illustrate how we obtain the amount of computation. The program shown below takes two numbers x and y as inputs, and finds which occurs more frequently in a linked list a. Each element in the list has two fields: *value* containing an item and *next* pointing to the following element in the list. The program uses c_x and c_y to count the occurrences. This program is offloaded to a server for running t seconds.

```
MORE-FREQUENT($list\ a, int\ x, int\ y, int\ t$)
$c_x, c_y, counter \leftarrow 0$
**while** $time < t$ **and** $a.next$ != $null$
▷ loop for $t$ seconds **or** till the end of the list
    **do**
        **if** ($a.value == x$)
            $\{c_x \leftarrow c_x + 1\}$
        **if** ($a.value == y$)
            $\{c_y \leftarrow c_y + 1\}$
        $a \leftarrow a.next$
        $counter \leftarrow counter + 1$
**if** $c_x \geq c_y$
return $(x, counter)$
    **else**
return $(y, counter)$
```

The result of the program is either x or y; however, the server may not search the entire list within t seconds. In such a case, the number of the elements searched depends on the speed of the server. Searching more elements increases the likelihood of returning a correct answer. When *counter* is the list's length ($counter = |a|$), the answer is correct. In this example, the client needs to know how many of a's elements have been checked and inserted the counter in the code. When the program returns, the counter's value is appended to the answer and sent back to the client. The code to update *counter* is inserted for the purpose of quantifying the amount of work performed by the server. The code can be inserted anywhere inside the `while` loop. Unfortunately, the code is easy for a server to detect. Sections 3.2 and 3.3 explain how to determine the locations for inserted code and their operations that are more difficult to be detected by a server.

3.2 Locations of Inserted Code

In a complex program, code may be inserted in multiple locations to determine the amount of computation performed by a server. We call these locations *checkpoints*. We have to determine (1) what operations to perform at checkpoints and (2) where to insert checkpoints. At each checkpoint, a counter is used to quantify the amount of computation performed by a server. Checkpoints should be inserted inside the segments of code when the amount of computation may vary in different executions. These locations are determined based on the program's structure. We classify the structure into three cases:

(1) *Simple Loops.* This is the case shown in the previous example. In a simple loop, a checkpoint is inserted *inside* the loop to count the number of iterations. The example in Section 3.1 shows a checkpoint placed inside the `while` loop.

(2) *Branches.* A program may have several branches with different amounts of computation. The control flow of the branching condition is not known before execution. Placing a checkpoint in each of the branches can determine which branch is executed and consequently the amount of computation that is performed by executing that branch.

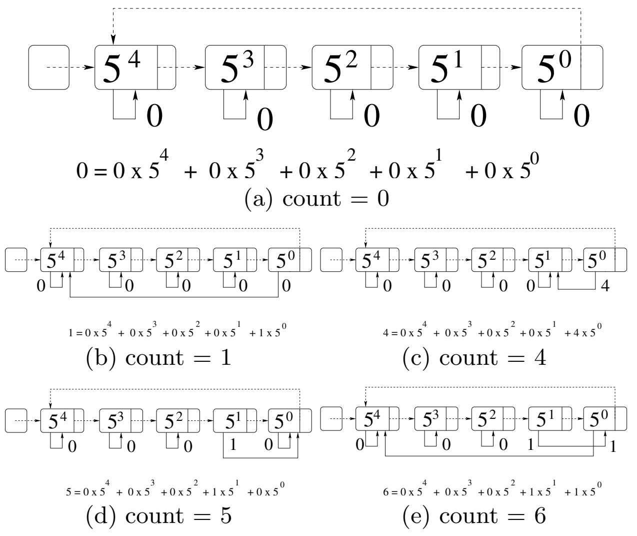
(3) *Nested Segments.* Loops and branches may form nested segments: (i) a loop within a loop, (ii) a condition within a loop, (iii) a loop within a condition, and (iv) a condition within another condition. A checkpoint may be placed outside the outer segment, inside the outer segment, or inside the inner segment. These locations provide different degrees of accuracy and overhead. We suggest adding two checkpoints, one in the outer segment and the other in the inner segment. The following example explains the necessity of two checkpoints.

```
NESTED-LOOP(int $x$, int $y$)
for $i_x = 1$ to $x$
    do
        ▷ checkpoint at the outer segment
        ▷ perform computation I
        for $i_y = 1$ to $y$
            do
                ▷ checkpoint at the inner segment
                ▷ perform computation II
```

A checkpoint in the outside segment counts the value of x; a checkpoint in the inner segment counts the value of $x \cdot y$. If only the first checkpoint is used, the client has no information about y. If only the second checkpoint is used, the client obtains the product but has no information about the individual values of x or y. If $x \cdot y$ is 100, this may occur for different pairs of (x, y), such as $(10, 10)$, $(20, 5)$, $(5, 20)$, or $(100, 1)$. If the amounts of computation in I and II are significantly different, knowing $x \cdot y$ is insufficient. Instead, two checkpoints should be inserted to obtain the values of x and y individually.

3.3 Counters at Checkpoints

The example in Section 3.1 uses a simple way to increment a counter. This can be easily detected by a server; as a result, a server may manipulate the counter without executing the rest of the program. Hence we propose radix-graph counters; these counters use pointer operations and they are more difficult to detect by program analysis. Figure 1 shows a radix graph with five nodes, and a head. This graph is a base-five counter. Each node i has two pointers: p_i and q_i. The set $\{p_i | 1 \le i \le 5\}$ forms a circular linked list and these pointers never change. The subscript i is counted from the tail of the list; thus, the node immediately after the head is the fifth node, i.e. $i = 5$. The location of the element i determines the index of the element, with the element near the head with the highest index (i.e. the most significant digit). Each q_i points to a node j; the distance of the node j from i gives the coefficient at each i. Distances are counted in the direction of the p_i. The count works by following three steps:



$0 = 0 \times 5^4 + 0 \times 5^3 + 0 \times 5^2 + 0 \times 5^1 + 0 \times 5^0$
(a) count = 0
$1 = 0 \times 5^4 + 0 \times 5^3 + 0 \times 5^2 + 0 \times 5^1 + 1 \times 5^0$
(b) count = 1
$4 = 0 \times 5^4 + 0 \times 5^3 + 0 \times 5^2 + 0 \times 5^1 + 4 \times 5^0$
(c) count = 4
$5 = 0 \times 5^4 + 0 \times 5^3 + 0 \times 5^2 + 1 \times 5^1 + 0 \times 5^0$
(d) count = 5
$6 = 0 \times 5^4 + 0 \times 5^3 + 0 \times 5^2 + 1 \times 5^1 + 1 \times 5^0$
(e) count = 6

Figure 1: Radix graph counter showing different values. The dotted lines show the p pointers to form the linked list. The solid lines show the q pointers to encode the value of the counter. 1(a) The initial condition with the value of zero. Counting is performed by pointer displacements. 1(b) shows that q_1 (the tail node) points to the fifth node (immediately after the head). This encodes a distance of 1 in the direction of p_1. Figures 1(c), 1(d) and 1(e) show the radix counter for values of 4,5, and 6 respectively.

(1) Initialization: When a radix graph of m nodes is created, $\{p_i \mid 1 \le i \le m\}$ forms a circular linked list and each q_i points to itself. This corresponds to zero and is shown in Figure 1(a).

(2) Counting: Every time the graph is accessed, the value encoded by the graph increments by one. Figure 1(a) shows the initial state, where all q_i's are self pointers. The first access modifies q_1 so that it points to the fifth node by assigning p_1 to q_1. This represents one step from the first node and encodes the value of one. The next increment modifies q_1 again. Since q_1 points to the fifth node now, $q_1.next$ points to the fourth node; $q_1 \leftarrow q_1.next$ moves the pointer. This process continues until $q_1.next$ is the first node, as shown in Figure 1(c). The next increment resets q_1 and modifies q_2, as shown in Figure 1(d) and encodes the value of five. Figure 1(e) shows the encoding of six. A counter with m nodes can reach a maximum value of $(m-1) \times \sum_{i=0}^{m-1} m^i$.

(3) Traversal: Before the results are sent from the server, the radix graph is traversed to determine the value by counting the displacements of the q_i's. This value is embedded in the result and hidden in text or images using steganographic techniques [18]. The client uses the value to determine how much computation has been performed by the server.

3.4 Energy Reduction and Consumption

The offloaded program has multiple checkpoints. When the result is returned to the client, it adds these counters to determine the total amount of computation performed by the server. Suppose c is this total amount of computation.

We can estimate the amount of energy savings as follows. Suppose the client does not offload the program. Let k be the client's performance without offloading. The same amount of computation would take $\frac{c}{k}$ seconds to finish at the client. Let p_c be the client's power consumption for performing this computation. The client would have consumed energy

$$p_c \frac{c}{k}. \tag{1}$$

When the client offloads the program, the client has to wait for the result from the server. The client consumes idle power p_i and waits for t seconds. Thus, the client consumes

$$p_i t \tag{2}$$

if the computation is offloaded. We assume the idle power is a constant in this paper. In reality, the idle power may change due to frequency scaling [19]. We also need to add the overhead of offloading. This overhead is dominated by the energy consumption for sending the program to the server and receiving the result from the server. Let e_o be this energy overhead.

3.5 Server Ranking

A server's rank is determined by the ratio of the saved energy and the consumed energy:

$$\frac{p_c \frac{c}{k}}{p_i t + e_o}. \tag{3}$$

A higher-rank (i.e. more desirable) server produces a larger ratio. This formula can be used to rank servers in different scenarios.

(1) If e_o is small enough and can be ignored, the ratio is simplified to

$$\frac{p_c \frac{c}{k}}{p_i t + e_o} = \frac{p_c}{p_i} \frac{\frac{c}{k}}{t}. \tag{4}$$

The first fraction is the ratio of the computation power and the idle power of the client. The second fraction is the ratio of the execution time at the client and the client's waiting time. Since the server is faster than the client, the waiting time t is smaller than $\frac{c}{k}$. A faster server can finish the computation more quickly; as a result, the ratio is larger.

(2) Even if a server intends to cheat by returning random results quickly, t is small but c is zero. The server is ranked lower.

(3) Offloading actually saves energy if the ratio is larger than one. If the communication overhead e_o is too large and the amount of computation c is too small, the ratio is smaller than one. This indicates that offloading is not beneficial. If the idle power and the communication overhead are negligible ($p_i \approx 0$ and $e_o \approx 0$), offloading is always beneficial because the ratio is always larger than one.

(4) For a given client, we assume its computation power p_c, idle power p_i, and performance k are constants. The servers' ranking is determined by the relative values of the ratio. This ratio can be measured by using two terms: c and t. The former is obtained by the inserted counters and the latter is the time waiting for the result. *Neither needs special hardware supports.* Consequently, our method can be easily deployed to real systems.

4. EXPERIMENTS

4.1 Workload

We use content-based image retrieval (CBIR) as the workload in our experiments. CBIR has been shown as a useful application for mobile users and for studying computation offloading [7, 8, 20]. In CBIR, an image is represented by a set of numbers called its *features*. Image matching is performed by comparing their features. The number of features compared depends on the images; this can vary based on the optimization proposed in [21].

We implement a CBIR program based on Haar wavelets in C#. The program has one major loop controlled by the number of images. This loop contains many nested loops but each has a fixed number of iterations. Hence, a checkpoint inside the outer loop is sufficient to determine the amount of computation in these inner loops. One exception is the loop to compare images' features. This loop can be controlled by the similarity between the query image and the images in the collection. Each iteration of the outer loop corresponds to one image from the image collection being compared with the query image. Each iteration of the inner loop corresponds to one feature being compared; hence, the number of iterations gives the total number of features compared. We use a collection of 10,000 images in our experiments. We modify the program so that it can terminate at a given time or for a given number of images. We also modify the program so that it can compare either a fixed or a variable number of features. Checkpoints are inserted manually. We implement our graph counter with eight nodes (m=8) to obtain the amount of computation performed by the program. The maximum possible value of this counter is 16,777,215 (base 10) and is sufficient for our experiments. Since the program contains several hundreds of lines of code and performs many complex image processing operations, the overhead of executing the checkpoints is less than 0.5% of the total execution time and is considered acceptable.

4.2 Setup

We use an HP iPAQ hw6945 and a National Instruments data acquisition card for energy measurements. We insert a 0.25 Ω resistor in series between the iPAQ and its battery. We collect voltage samples across the battery at a frequency of 10,000 samples per second using the data acquisition card. We measure the power drawn from the battery of the iPAQ when (1) it is performing computation and (2) it is idle with the wireless network in the poll mode. The average power of p_c and p_i is 866 mW and 299 mW, respectively.

Three servers are used for comparison. Their configurations are shown in Table 1. The PDA communicates with the servers using TCP sockets through a WiFi network. The program is compiled using the .NET framework and the executable is obtained. The executable is run on the PDA and the servers. Since the size of the executable is only 20kB, we do not consider the energy for sending the executable to the servers. The average size of an image is 30kB. The program transmits the query image to the server and returns the top 5 results to the client. The total amount of data exchanged between the client and a server does not exceed 200kB. The WiFi network's bandwidth is 600kB/sec so communication take no more than 0.33 seconds; we can ignore the communication energy e_0. The value of c is computed by adding the counters at the client. Since at most two counters are

Table 1: PDA and server configurations.

Item	Speed (MHz)	Processor	Memory (MB)
PDA	416	XScalePXA270	64
server 1 (S1)	2160	AMD 64x2 Duo	3006
server 2 (S2)	1200,599	Intel Pentium M	376
server 3 (S3)	1600,221	Intel Pentium M	1500

inserted in our experiments, the energy overhead for calculating c is negligible.

4.3 Ranking Servers using One Checkpoint

We first consider using one checkpoint to estimate the amount of computation by each server. As described in Section 4.1, the CBIR program has two major loops. The outer loop is controlled by the number of images and the inner loop is controlled by the number of features. In this experiment, one checkpoint is inserted inside the outer loop and 60 features are used for every image. Hence, the second checkpoint is unnecessary. The servers are ranked by using two terms: c and t. We consider their ranks in two cases.

4.3.1 *Fix t and Observe c*

In the first case, the waiting time t at the client is fixed and this time is also sent to the server. The offloaded program has a timer. When the timer expires, the program selects the best five matches so far and sends them to the client. The amount of computation performed by each server is shown in Figure 2. For $t = 20$ seconds, S1 searches 183 images ($c = 183$), S2 searches 94 images, and S3 searches 51 images. The relative performance, normalized to S1, is 1, 0.51, and 0.28 respectively. S1 provides the best energy savings since it performs the most computation in the same amount of time. If the server is executing multiple tasks, c will reduce for a given t, and this will lower the rank of the server.

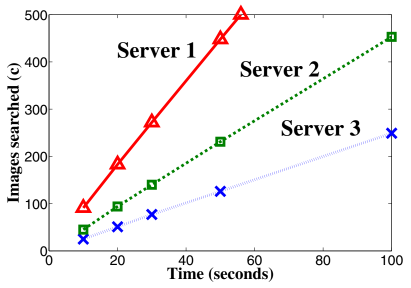


Figure 2: Amount of computation c observed when t is fixed.

4.3.2 *Fix c and Observe t*

In the second comparison, we fix the amount of computation c performed by each server and observe the waiting time t. When each server compares thirty images ($c = 30$), the values of t are observed to be 3, 7, and 12 seconds respectively. The ranks obtained (normalized to S1) are 1, 0.43, and 0.25 respectively. As can be seen in this comparison, S1 consistently has a better rank. Figure 3 shows t for different values of c. A server may also execute other tasks and take longer to complete the program. As a result, t may become larger and the server will be ranked lower when t is too large.

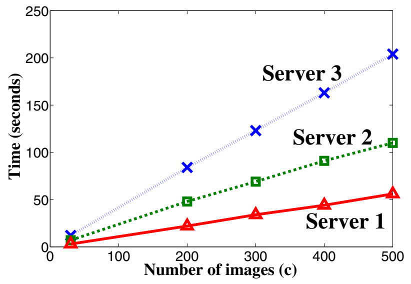


Figure 3: Time t observed for different values of c.

4.4 Ranking Servers using Multiple Checkpoints

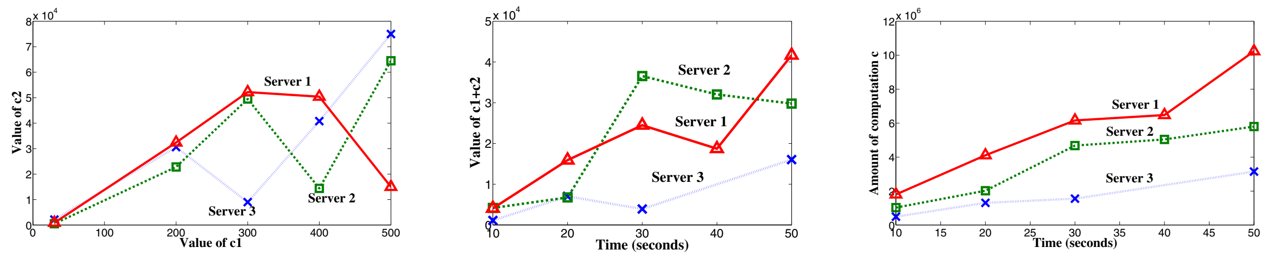
Next, we consider the scenarios when the numbers of iterations vary in both the outer and the inner loops. The numbers of features may depend on the similarity between the query image and the rest of the images in the collection, as suggested in [21]. When the query image is sufficiently different from most of the images, only a few features can distinguish the best matches. When the query image is similar to most of the images, more features are needed to select the best matches.

Let $c1$ and $c2$ be two checkpoints placed inside the outer loop and the inner loop. The first checkpoint provides the number of images compared. The second checkpoint is the sum of all features compared. In this comparison, a fixed waiting time t is given to the servers. Figure 4(a) shows the values of $c2$ for different $c1$'s. In this figure, $c1$ indicates the number of images compared. When $c1$ is 200, 200 images are randomly selected from a total collection of 10,000 images. As can be seen in this figure, $c2$ is not monotonically increasing as $c1$ becomes larger. When $c1$ increases from 200 to 500, $c2$ may increase or decrease, depending on which 500 images are chosen from the 10,000 images. This figure suggests that when the numbers of iterations of the two nested loops are independently controlled, one checkpoint does not provide sufficient information about the other checkpoint. Both checkpoints are required to estimate the amount of computation.

The sum of the two counters does not accurately rank the servers. Figure 4(b) shows that in some cases, S2 may be ranked higher even though it is actually a slower computer than S1. This occurs because the two nested loops perform different amounts of computation. The outer loop executes 16984 statements to process the images for comparison. The inner loop executes only 63 statements to compare the features produced from the outer loop. Hence, the two loops cannot be treated equally. If we scale the two counters proportionally to the numbers of statements, Figure 4(c) shows that the three servers are correctly ranked. The ranks (normalized to S1) are calculated to be 1, 0.49, and 0.31 respectively.

5. CONCLUSION

We present energy-based ranking of servers for computation offloading. The ranks are determined by the amounts of



(a) (b) (c)

Figure 4: (a) The values of $c2$ vary for different $c1$'s. (b) Iterations of $c2$ observed for different values of t. (c) Amount of computation c for different values of t.

computation performed at each server and the waiting time of the client. The former is obtained by inserting counters in the offloaded programs. These counters use radix graphs that are more difficult to detect and hence prevent servers from directly manipulating the counters without executing the offloaded programs. We implement our method on a PDA and three servers and demonstrate that our method can consistently rank the three servers.

6. FUTURE WORK

In this paper, the checkpoints are manually inserted and the amount of computation is scaled based on the number of statements. Our work may be extended by using a compiler to insert checkpoints based on control flow analysis. The compiler can also scale the amount of computation based on the number of instructions. This can be more accurate than using the number of statements.

Acknowledgments

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

7. REFERENCES

[1] ME Anagnostou, A. Juhola, and ED Sykas. Context Aware services as a step to pervasive computing. In *Lobster Workshop on Location based Services*, pages 4–5, 2002.

[2] S. Sivavakeesar, O.F. Gonzalez, and G. Pavlou. Service Discovery Strategies in Ubiquitous Communication Environments. *IEEE Communications Magazine*, 44(9):106–113, 2006.

[3] Shumao Ou, Kun Yang, and Liang Hu. Cross: A combined routing and surrogate selection algorithm for pervasive service offloading in mobile ad hoc environments. In *IEEE GLOBECOM*, pages 720–725, November 2007.

[4] T. Guan, E. Zaluska, and DD Roure. Extending Pervasive Devices with the Semantic Grid: A Service Infrastructure Approach. In *IEEE Conference on Computer and Information Technology*, 2006.

[5] T. Guan, E. Zaluska, and D. De Roure. A Grid Service Infrastructure for Mobile Devices. In *First IEEE International Conference On Semantics, Knowledge, and Grid*, volume 200.

[6] H. Zeng, C.S. Ellis, A.R. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. In *USENIX Annual Technical Conf*, 2003.

[7] H. Sonobe, S. Takagi, and F. Yoshimoto. Mobile computing system for fish image retrieval. In *International Workshop on Advanced Image Technology*, pages 33–37, 2004.

[8] M. Noda, H. Sonobe, S. Takagi, and F. Yoshimoto. Cosmos: convenient image retrieval system of flowers for mobile computing situations. In *IASTED*, pages 25–30, 2002.

[9] J.M. Hammersley and D.C. Handscomb. *Monte Carlo Methods*. Methuen Young Books, 1964.

[10] Mark Kritzman, Simon Myrgren, and Sebastien Page. Portfolio Rebalancing: A Test of the Markowitz-Van Dijk Heuristic. *SSRN eLibrary*, 2007.

[11] Y. Zhang, S. Zhang, and H. Tong. Adaptive Service Delivery for Mobile Users in Ubiquitous Computing Environments. *Lecture Notes in Computer Science*, 4159:209, 2006.

[12] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi. Using bandwidth data to make computation offloading decisions. In *International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

[13] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Adaptive computation offloading for energy conservation on battery-powered systems. In *International Conference on Parallel and Distributed Systems*, pages 1–8, December 2007.

[14] 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.

[15] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *POPL '03*, pages 115–125, 2003.

[16] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[17] Karthik Kumar, Yamini Nimmagadda, and Yung-Hsiang Lu. Establishing Trust for Computation Offloading. In *International Conference on Computer Communications and Networks*, 2009.

[18] R. Chandramouli and N. Memon. Analysis of LSB based image steganography techniques. In *ICIP*, volume 3, pages 1019–1022, 2001.

[19] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Dynamic Frequency Scaling with Buffer Insertion for Mixed Workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1284–1305, 2002.

[20] Yu-Ju Hong, Karthik Kumar, and Yung-Hsiang Lu. Energy Conservation by Adaptive Feature Loading for Mobile Content-based Image Retrieval. In *ISCAS*, 2009.

[21] Karthik Kumar, Yamini Nimmagadda, Yu-Ju Hong, and Yung-Hsiang Lu. Energy conservation by adaptive feature loading for mobile content-based image retrieval. In *ISLPED*, pages 153–158. ACM, 2008.