Online Migration Solver Based on Instructions Statistics

Yuanren Song, Zhijun Li

Online Migration Solver based on Instructions Statistics

Algorithm for deciding offload function set on mobile cloud system

Yuanren Song[1], Zhijun Li[2]

[1]Harbin Institute of Technology, China
windstonedream@gmail.com

ABSTRACT. With the popularization of cloud computing, solving the power shortage problem of mobile devices in a cloud way comes into the eyes. And migrating the whole virtual machine which can save cost for both developers and users has been realized. This paper redefine the function migration problem, and use a dynamic programming algorithm to solve it. This paper releases all functions based on COMET. 3 groups of experiment are conducted, and give speed-up of 7.11x, 23.23x, 8.24x.

KEYWORDS: Mobile Cloud, migrate, offload.

1 Introduction

With the development of smartphone and the growing number of the users. Lots of sensors are deployed on the mobile devices. Of course, more applications will acquire the data, analyse the data and use the data. But it will cost much computing resources which are tight on the mobile devices too. Another problem is power shortage. In MAUI system, the battery dried up in 2 hours by downloading 100kb file repeatedly.

There are several systems based on migrating the whole virtual machine. MAUI is based on Microsoft .Net Framework. The system propose a method to partition the program by functions. Then profile the energy consumption for each function. At last, it uses a solver to decide which part of functions needs to be offloaded to Server. The system is released on CLR, which can save 90% energy when running face recognition programme, and has a speed up of 9.5x. Comparing to MAUI, Clonecloud system doesn't need any annotations when programming. It have a speed-up of 14.05x when running virus scan program.

Joule Benchmark style system model the energy consumption. The model indicates energy consumption is in proportional to the CPU cycles. So in the paper, we use CPU cycles to calculate the running time, and use running time as the optimization objective. We propose an algorithm to solve such a problem.

In Section 2, we give the definition of the problem and give an algorithm to solve the problem. In Section 3, we provides some details when we implement the system. In Section 4, three groups of experiments are conducted to evaluate the system performance.

adfa, p. 1, 2011.

2 Function Migration Problem

2.1 Definition

The problem is based on a FIT (Function Invoking Tree). FIT is used to reflect the construction of the program. Each function can form a node in FIT. When Function A calls Function B, B will be one of A's children. When a function is invoked by different function, there will be several node in FIT to present the function in different context. Here we import (V, E) to present the FIT. V means the nodes for function, and E= {u, v} means Function u called Function v.

We also needs some other symbols to complete the model. Ii presents where should the Function i be run. Ii =0 means it will be run on mobile device, and Ii =1 means it will be run on cloud. The instruction length of Function i is presents by Fi. The number of called times is presents by Ri. And SL and SR represents the CPU speed of mobile device and cloud server. Δi presents the time used to synchronize the memory between the mobile device and the cloud. The variables Fi, Ri, SL, SR, Δi, all can be measured or counted by statistics. Then we use the running time migration can save as the optimization objective, and the problem is as above:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in N} I_i \left(\frac{R_i \cdot F_i}{S_L} - \frac{R_i \cdot F_i}{S_R} \right) - \sum_{i \in V} \sum_{(i,j) \in E} \left| I_i - I_j \right| R_j \Delta_j \\ \text{subject to} \quad & I_i \in \{0, 1\} \end{aligned} \tag{1}$$

2.2 Algorithm

Comparing to former problem, energy is replaced by time, so there's no constraint in this problem. So it isn't a 0-1 ILP strictly, and it's a P problem. Here we propose a dynamic programming algorithm, which will cost O(|V|) time to calculate the optimal solution.

In the algorithm, there's a weight w(T, IT) for each node T. It means the time migration can save when node T chooses IT as the schedule. The weight equals the optimization objective, and the proof will be shown in next chapter.

The algorithm start from the root node of the FIT. And the algorithm can be divided into 2 parts: calculate weight for each node, and decide whether to offload for each node. The algorithm is as below:

```
program CalcWeight(w(T, I))
  {Assuming Input the Node T and Decision I};
  begin
    If w(T, I) is calculated then
      Return w(T, I)
    Else
      If T is leaf node then
        w(T, 0) := 0
      Else
```

$$w(T,0) = \sum_{(T,i)\in E} \frac{R_i}{R_T} \cdot \max\left(CalcWeight(i,0), CalcWeight(i,1) - \Delta_i\right) \tag{2}$$

```
      End if
      If node T is native function then
        w(T, 1) := -∞
      Else
```

$$w(T,1) := \sum_{(T,i)\in E} \frac{R_i}{R_T} \cdot \max\left(w(i,0) - \Delta_i, w(i,1)\right) + F_T\left(\frac{1}{S_L} - \frac{1}{S_R}\right) \tag{3}$$

```
      End if
    End if
end
```

2.3 Proof

Theory 1. The root node weight calculated by the algorithm is equal to the optimization objective.

Proof: The root node weight is decided by max function in the algorithm. Then the max function can be replaced by I and 1-I. So the root weight w can be written:

$$\begin{aligned}
w &= I_{root} w(root,1) - I_{root}\Delta_{root} + (1-I_{root}) w(root,0) \\
&= I_{root} \sum_{(root,i)\in E} \frac{R_i}{R_{root}} \left(I_i w(i,1) + (1-I_i) w(i,0) - (1-I_i)\Delta_i\right) + I_{root} F_{root}\left(\frac{1}{S_L} - \frac{1}{S_R}\right) \\
&\quad + (1-I_{root}) \sum_{(root,i)\in E} \frac{R_i}{R_{root}} \left(I_i w(i,1) - (I_i)\Delta_i + (1-I_i) w(i,0)\right) \\
&= \frac{R_j}{R_{root}} \sum_{(root,i)\in E} \sum_{(i,j)\in E} I_{root} I_i I_j w(i,1) + (1-I_{root}) I_i I_j (i,1) + I_{root} I_i (1-I_j) w(i,0) + \\
&\quad + (1-I_{root}) I_i (1-I_j) w(i,0) + I_{root} (1-I_i) I_j w(i,1) + (1-I_{root})(1-I_i) I_j w(i,1) \\
&\quad I_{root} (1-I_i)(1-I_j) w(i,0) + (1-I_{root})(1-I_i)(1-I_j) w(i,0) + I_{root} F_{root}\left(\frac{1}{S_L} - \frac{1}{S_R}\right) \\
&\quad - I_{root} I_i (1-I_j)\Delta_j - I_{root}(1-I_i) I_j \Delta_j - (1-I_{root})(1-I_i) I_j \Delta_j - (1-I_{root}) I_i (1-I_j)\Delta_j \\
&\quad + I_{root} \sum_{(root,i)\in E} \frac{R_i}{R_{root}} I_i F_i\left(\frac{1}{S_L} - \frac{1}{S_R}\right) + (1-I_{root}) \sum_{(root,i)\in E} \frac{R_i}{R_{root}} I_i F_i\left(\frac{1}{S_L} - \frac{1}{S_R}\right) \\
&= \frac{R_j}{R_{root}} \sum_{(root,i)\in E} \sum_{(i,j)\in E} I_j w(i,1) + (1-I_j) w(i,0) - |I_i - I_j|\Delta_j \\
&\quad + I_{root} F_{root}\left(\frac{1}{S_L} - \frac{1}{S_R}\right) + \sum_{(root,i)\in E} \frac{R_i}{R_{root}} I_i F_i \left(\frac{1}{S_L} - \frac{1}{S_R}\right)
\end{aligned} \tag{4}$$

Expand all the node in FIT and let $R_{root} = 1$, we'll get:

$$w = \sum_{i \in V} I_i R_i F_i \left(\frac{1}{S_L} - \frac{1}{S_R} \right) - \sum_{i \in V} \sum_{(i,j) \in E} \left| I_i - I_j \right| \Delta_j \tag{5}$$

Theory 2. This problem satisfies dynamic programming property.

Proof: For one node i and one of its son Node j. Let the weight for i be w_i which includes largest weight w_j, but w_i isn't the largest weight for i. Let the largest weight for i be w_i', and it includes Node j's solution w_j'. Then we have $f_i' = f_i - f_j + f_j' < f_i$, which is contradict with that fi' is the largest weight for i.

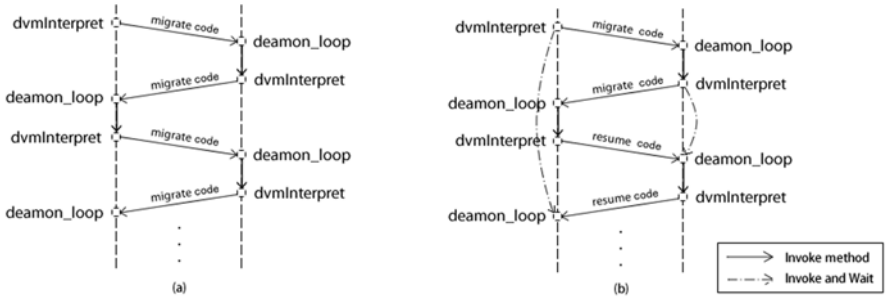
3 System Implements

The system is realized based on COMET which is published by 2013. Comparing to the Clonecloud system, COMET uses DSM (Distributed Shared Memory) instead of migrating whole stack. So all the threads can access the same block memory simultaneous in a synchronized way. And more than one threads can be migrated to the cloud server at the same time.

COMET is an open source project which modifies the Dalvik Virtual Machine used in Android system. We save the part of network transportation and DSM synchronization, and add our features on it.

First we run a group of test program to get the real instructions in the program and measure the running time to model the CPU speed on both mobile devices and cloud server.

Fig. [1]. (a) COMET migration procedure (b) Our system migration procedure



Then we start to run our algorithm to calculate the set of function to offload. When calculating the function set, we need to know the length of instructions F for each function. We just count the running time of the function, then use the mobile speed to evaluate the length of instructions. Another variables Δ should be measured during migration. For each function we set Δ be the RTT between mobile devices and cloud server. As the migration done, we update Δ for each migrated function.

For native method, we use the dictionary provided by COMET to judge which to offload or not. The offloadable methods include some math library, thread library and

so on. So this part of method can be run on the cloud server, which can save lots of work for the mobile device.

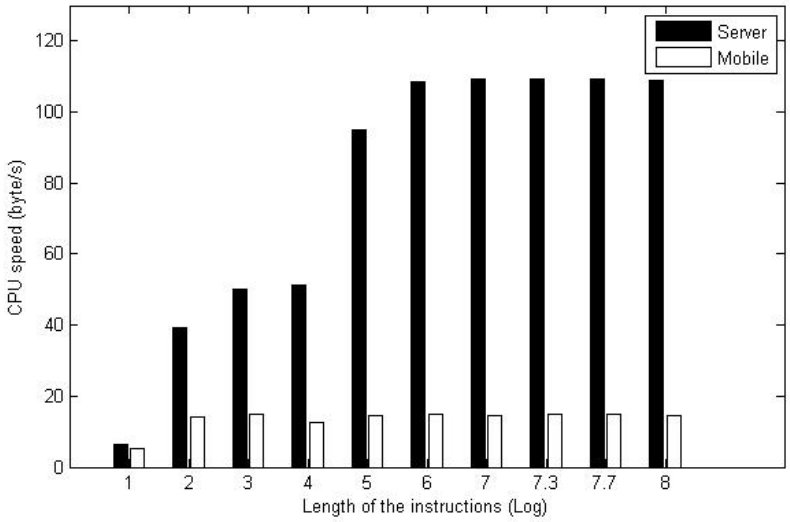
In COMET, it uses migration code to trigger migration. The code is handled by a method, and when the thread is migrated back it calls another such method to handle the migration code. So the original method will never return (Fig 1a). With more times of migration, the stack grows. We add the function of return, in order to solve the problem and we can add some other work after the migration (Fig 1b).

4 Experiments and Results

4.1 CPU Speed Measure

We program the test program with basic arithmetic instructions, control instructions and loop instructions. The loop is used to control the whole length of the instructions. In order not to increase any extra waste of CPU cycles, we counts length of instructions when a jump instruction comes. The measure of the CPU speed is shown as Fig 2.

Fig. [2]. The CPU speed becomes stable with the growth of the instruction length



The result means we can't use linear model for the functions that have few instructions. But when instructions' length grows more than 106 bytes, the CPU speed becomes stable. And the mobile device and server speed SL, SR is:

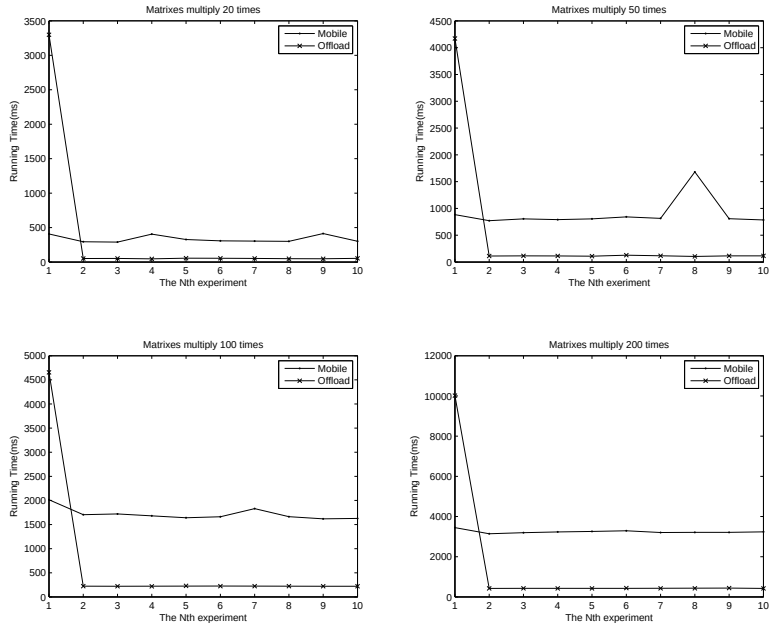
$$S_L = 109.207MB / s$$
$$S_R = 14.729MB / s \quad (6)$$

Further, when the length is less than 106 bytes, the running time is less than 106/SL=68ms. So the functions runs less than 68ms on mobile are ignored. And when the length is less than 106 bytes, the migration saves time less than 106 / SL - 106 / SR = 59ms. In ordinary wireless channel, the RTT time usually more than 59ms. So the ignoring is reasonable.

4.2 Matrix Multiplication

Matrix Multiplication experiment is designed for situation that we want to run specific length of instructions as soon as possible. The experiment use with native algorithm time cost of $O(N^3)$. Multiply the 200*200 matrixes 20, 50, 100, 200 times. The result is shown in Fig 3.

Fig. [3]. The time cost of matrix multiplication 20, 50, 100, 200 times on mobile and server



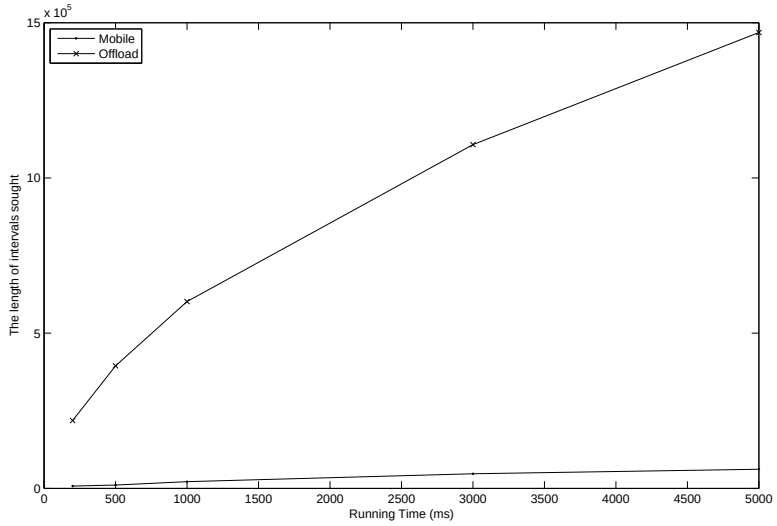
As shown above, the first migration cost much time, because the memory data needs to be synchronized is accumulated. After the first migration, the memory modification is much less, so the running time on server falls into an acceptable threshold. Another question is why it can still offload again after the first migration cost much of the time. Because it first chooses thread creating function as the migration function, then it chooses matrix multiplication function. And we have an average speed-up of 7.11x.

4.3 Seeking Primes

Seeking Primes experiment is designed for situation that we want to get the result as better as possible. We modify sieve algorithm to satisfy the requirement. We divide the positive integers into intervals, and each interval have 8192 integers. The program seeks an interval and test the time then. The length of the interval sought is used as the result. The result is shown in Fig 4.

When the RTT is more than 200ms, when we ask the program run 200ms, it will not be offloaded. Until RTT is less than 150ms, it can be offloaded. The speed-up increases obviously, and average speed-up is 23.23x.

Fig. [4]. The relation between the running time and length of interval

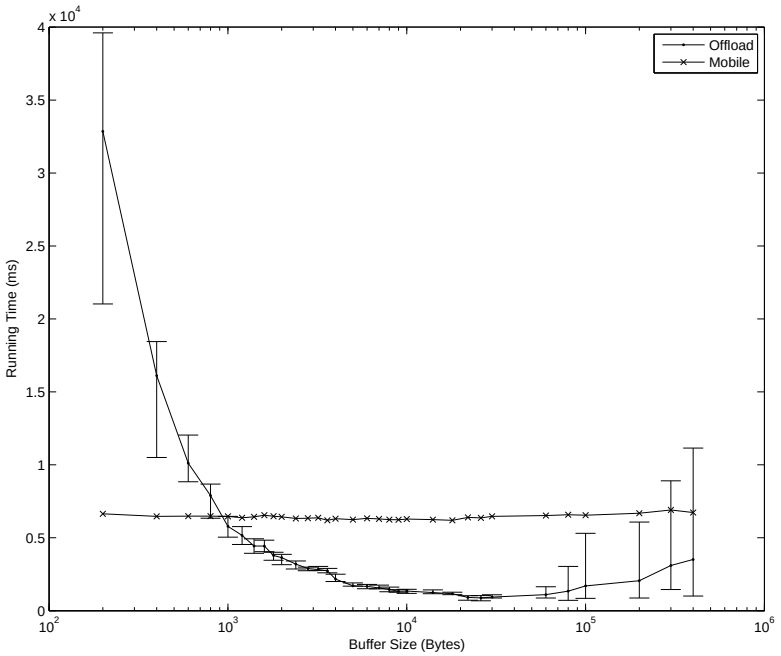


4.4 IO Test

IO Test experiment is designed for situation that we process the file on the mobile device. We modify the text statistics program provided by Bell Lab. The input file is a 64K english essay. We test for different length of buffer size, and the result is shown in Fig 5.

The running time on mobile is always 6s. When it comes to offloading, it will be stable when the buffer size is 10000 bytes. In this situation, the time for network communication is minimum. When the buffer size is too small, server will require the file for many times, which cost much time for network communication. When the buffer size is too large, it will need more memory, which will causes more errors during synchronization. So the running time become larger and more unstable. The average speed-up is 8.24x, according to the 7 experiments in the middle.

Fig. [5]. The relationship between running time and buffer size



5 Conclusion

The paper proposes a problem with the time migration can save as the optimization objective. And a dynamic programming algorithm solves the problem. Then we implement the algorithm on COMET system. 3 groups of experiment are conducted, which shows speed-up of 7.11x, 23.23x, 8.24x.

6 References

1. L. Siegele, Let it rise: a special report on corporate it, http://www.economist.com/node/12411882, 2008.
2. E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, in: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys'10, ACM, New York, NY, USA, 2010, pp. 49–62.
3. B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, Clonecloud: elastic execution between mobile device and cloud, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys'11, ACM, New York, NY, USA, 2011, pp. 301–314.

4. D. Huang, X. Zhang, M. Kang, J. Luo, Mobicloud: building secure cloud framework for mobile computing and communication, in: Proceedings of the Fifth IEEE International Symposium on Service Oriented System Engineering, SOSE, pp. 27–34.
5. Mark S. Gordon and et al., "COMET: Code Offload by Migrating Execution Transparently," 10th USENIX Symposium on Operating Systems Design and Implementation, pp. 93-106, 2012
6. A. Kansal, F. Zhao, Fine-grained energy profiling for power-aware application design, SIGMETRICS Performance Evaluation Review 36 (2008) 26–31.