Mirror

A Computation-offloading Framework for Sophisticated Mobile Games

Jiang, M.H.; Visser, O.W.; Prasetya, I.S.W.B.; Iosup, A.

DOI
10.1109/WoWMoM.2017.7974351

Publication date
2017

Document Version
Final published version

Published in
18th IEEE International Symposium on a World of Eireless, Mobile and Multimedia Networks, WoWMoM 2017

Mirror: A Computation-offloading Framework for Sophisticated Mobile Games

M.H. Jiang*‡, O.W. Visser‡, I.S.W.B. Prasetya*, A. Iosup‡
*Utrecht University, the Netherlands
‡Delft University of Technology, the Netherlands

***Abstract*—The low performance and power limitations of mobile devices severely limit the complexity and the duration of playing sessions of mobile games. This article examines the possibility of using computation-offloading to mitigate these problems while keeping the game playable. We design *Mirror*, a framework for offloading computation targeted at the demanding performance requirements of sophisticated mobile games. The key conceptual contributions of *Mirror* are design decisions that allow for dynamic fine-grained client-side offloading decisions, and a protocol for real-time asynchronous offloading for bounding network delays. We implement a prototype of *Mirror* and test it by performing offloading for the game *OpenTTD*. The results are promising, showing that *Mirror* can increase the performance and decrease the power consumption of games while keeping the gameplay fairly smooth.**

I. Introduction

Mobile devices, such as smart phones and tablets, have limited computational capacity and power supply. These factors severely limit the complexity and scale of mobile games that can be reasonably run on mobile platforms. A potential way to mitigate these problems is through *computation-offloading*, that is, letting remote machines do the resource-intensive parts of game logic and sending the results to the mobile client. This can potentially free up local resources on the mobile client and possibly save some of its valuable battery power [1], but also threatens to cancel out any benefits by introducing latency in game updates that make the game too slow to enjoy.

Very few offloading frameworks have been created with games in mind [2]. A few frameworks that have been tested with games have only done it for very simple games [3]–[6]. In this work we propose *Mirror*, a framework for computation-offloading targeting *sophisticated* mobile games that achieves a *desirable trade-off* between computational performance increase and game playability.

The main contributions of this work are as follows. First, we design a new computation-offloading framework called the *Mirror*. Second, we propose a new program-partitioning scheme that allows for dynamic fine-grained client-side offloading decisions Third and last, extensive experiments are conducted with the framework, resulting in a thorough examination of the different effects, both positive and negative, of offloading on a game. The results from these experiments show that the framework significantly increases the game's performance while providing a fairly smooth gaming experience. The results also show that the framework allows for a trade-off between performance and power consumption.

978-1-5386-2723-5/17/$31.00 ©2017 IEEE

II. The Mirror framework

We design in this section the *Mirror* framework for computation-offloading targeting sophisticated mobile games. Key to our design is a mobile game using the *Mirror* framework must implement a *Mirror* Client and a *Mirror* Server; the two will interact to offload computation.

A. Overview of the Mirror framework

Using the *Mirror* framework requires the developers to attach the framework to their game by implementing several interfaces. These interfaces allow the framework to control the overall flow of the game like starting, stopping and pausing, or are used for program partitioning (Section II-B). The implementation of the *Mirror* Client and *Mirror* Server should be identical with the only difference of setting a flag for whether to run the game in *Mirror* client mode or server mode. Additionally, the game must be implemented in such a way that given an initial game state, all subsequent states are deterministic and are separated by a pre-determined delta-time, which we will call a *tick*.

An *offloading session* starts with the client and the server connecting and agreeing on a game starting point using a save game. The client can simulate the whole game on the local device, but it can also communicate with the server which parts it wants to offload by subscribing to the results of calls to a certain function. The server will simulate the game ahead of the client to obtain and send the results of the offloaded functions that the client needs ahead of time, so when the client itself arrives at the particular tick at which that specific call occurred, it will already have the result of the call and can apply it directly to its own game state. This way the client does not need to pause the game and wait for a full network round trip time to offload every single call to a function it wants to offload.

The framework automatically ensures that the server is simulating sufficiently far ahead of the client for results of function calls to arrive at the client in time. It does this by periodically measuring the single trip time of the network connection between the machines and calculating how many in-game ticks would elapse during that time. This number is the minimum number of ticks that the server should be ahead of the client. Both the client and the server constantly notify each other of their current tick. If the client notices its own tick is too close to the server's, it will temporarily pause its own game to wait for the server.

The framework does not allow the server to be too far ahead of the client either, because when synchronizing user input, the framework needs to execute the action associated with the input synchronously on both machines. This means that the action must be scheduled in the future of both the client *and* the server. Allowing the server to simulate infinitely far ahead of the client would cause the action to be infinitely delayed too. So similarly to the client, if the server notices it is too far ahead of the client, it will also pause its own game.

B. Program Partitioning

The *Mirror* framework uses a combination of offline and online program partitioning by making use of the existing class-based structure of games to define game objects. It does so through the *offloadable entity* (OE) interface. For each type of game object that the developers want to offload, the class of the object needs to implement the OE interface. By doing so, the behavior of the object is separated into a non-offloadable part and a set of offloadable parts. For each of the offloadable parts, a function must be defined to package and send the result of a call to that part, and a function to handle and process such a result.

At run-time, each instance of a class that has implemented the OE interface can be offloaded individually from all the others. Such an instance is called an *offloadable instance* (OI). When the client wants to offload an instance, it subscribes to the results of all future calls to the offloadable parts of the instance. If the client decides to offload or stop offloading an instance, it will communicate this decision with the server using the ID of that instance and stop or start running the offloadable parts of that instance itself.

Each time the server finishes a call to an offloadable part of an OI, it will send an *event message* (EM) to the client with the ID of the instance, the ID of the function call, the result of the call, and the in-game time at which the call occurred. Upon receiving an EM, the client processes the message by passing the EM and calling the corresponding processing function of the OI associated with the message.

III. Experimental Setup

We have implemented *Mirror* and applied it to *Open Transport Tycoon Deluxe* (OpenTTD), a popular and active open-source re-implementation of the real-time strategy game *Transport Tycoon Deluxe* (1994). OpenTTD provides many new extensions over the original, and allows more players and more objects to be simultaneously in the game, which makes OpenTTD much more computation-intensive than the 1994-original and a good real-world sophisticated game to experiment with.

Our implementation offloads *2 types of objects* of OpenTTD. First, all types of road vehicles can be offloaded. The offloadable parts consist of the path-planning and the collision-detection functions. Second, the AI of non-player controlled companies can be offloaded. In terms of offloading granularity, the computation associated with road vehicles is computation are fine-grained, whereas the AI is much more computation-intensive and thus coarse-grained.

We conduct experiments with OpenTTD and the *Mirror* framework using 3 different mobile devices and 4 different server devices. As mobile devices, we use a Galaxy Nexus, a Nexus 6 and a Nexus 7. As server-devices, we use a Samsung Q330 laptop, an Amazon EC2.nano and an Amazon EC2.normal server located in Frankfurt, and the DAS4 commodity cluster of the Delft University of Technology. These client and server devices were chosen to experiment with different real-life specifications and situations.

As *workload*, we have created 6 save games, to act as the game starting points in the experiments. Each save game was created by running a single game of OpenTTD with AI players starting from a single map and periodically saving the game as it progresses. This results in each save game having more entities in the game than the one before it and therefore also have a higher computational load.

A *simulation run* consists of using a certain client-device, connecting it with one of the server machines through a university Wi-Fi connection, and starting the game at one of the save games. Each combination was run 3 times with each run lasting around 4 minutes. An additional set of simulations without offloading were also performed to act as the baseline for comparison. These baseline tests were performed for every combination of client device and save game and were also performed 3 times for each combination with a running time of 4 minutes each.

A special type of setup that was only available for the Galaxy Nexus device was used to measure the effect of offloading on the power consumption on the client-side. The equipment used to do the actual power measuring was the *Power Monitor* by Monsoon Solutions Inc. [7], which is a high-frequency, high-accuracy device that is also intrusive (it requires soldering on the mobile hardware). The power consumption experiments were done by connecting the Galaxy Nexus with the Amazon EC2.nano server and testing all save games with and without offloading. Each parameter combination for this last setup was only run once for 4 minutes each.

IV. Results

We present in this section only representative results: graphs use only the data from the experiments using the Galaxy Nexus client device connecting with the Amazon EC2.normal server machine only. The number of OIs in each save game is used an an indicator of the computational load of each save game.

A. Performance and Playability

The number of game simulation ticks per second (TPS) the client can perform is used to measure the performance of the game; higher values are better. Figure 1 shows the TPS results with and without offloading. From our experience, OpenTTD is fairly playable as long as the TPS stays above 20. If the game runs even slower than 20 TPS, the user-interface (UI) becomes too unresponsive to comfortably play. This limit is

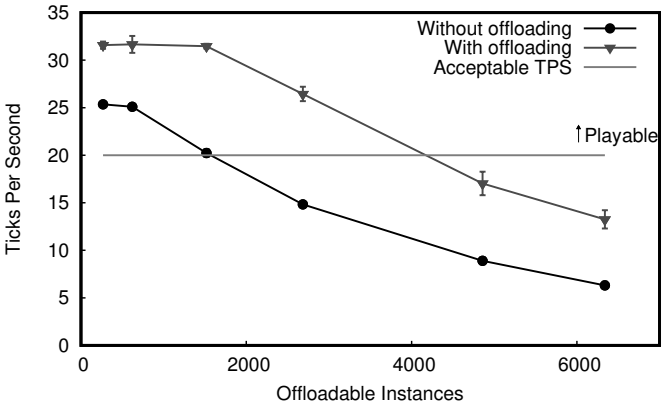

Fig. 1: *The performance results* in ticks per second. Each sample is the average over three runs.

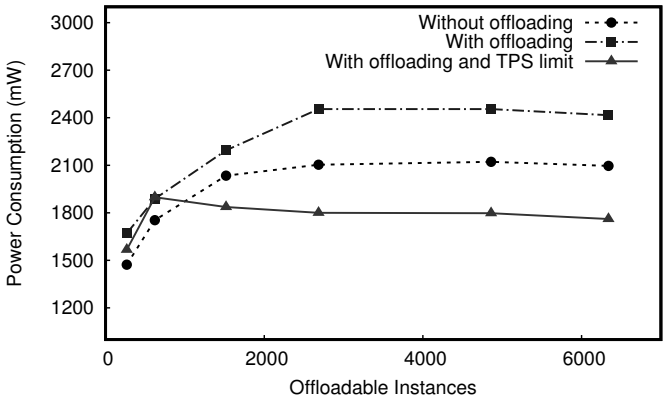

Fig. 2: The power consumption results in milliwatts. Each parameter combination was only run once.

consistent with previous findings about the impact of game latency on user experience, for real-time strategy games [8].

The Galaxy Nexus is a fairly old device and the results show that it cannot manage to run OpenTTD at the maximum TPS even at very low computational loads without offloading. The performance of the game rapidly drops below the playable level as the computational loads increase. The results show that turning offloading on significantly increases the performance of the game and is able to nearly double the number of ticks that the client can do at higher computational loads. This shows that, from a pure performance perspective, the *Mirror* framework is able to beneficially offload very fine-grained functions and is also able to scale well with the computational load of the program.

Section II-A explained that the client may sometimes need to pause its own game to wait for the server. This can cause a decrease in the game's smoothness and playability. From our experiments, we have observed that these situations are rare, and if they occur, are so short that the user might not even notice. Another aspect explained in that section is user input delay. In our experiments, we have observed delays ranging from 90 to 300 milliseconds, which is too high for most genres of games, but acceptable for playing real-time strategy games like OpenTTD [9].

B. Power Consumption

Figure 2 shows the power consumption on the mobile device with and without offloading. The results show a clear trade-off the game developer and the player of the game can make between performance and power consumption. Using offloading on a device that (by far) cannot run the game at the maximum speed will result in a significant performance increase, but also in power consumption. If the device can run the game at or near the maximum speed, offloading can slightly decrease the power consumption of the game. If the player is satisfied with a certain performance of the game without offloading, the game should allow the player to limit its simulation speed to save power with offloading.

V. Conclusion and Future Work

This article presented the *Mirror* Framework, a new computation-offloading framework for sophisticated mobile games. *Mirror* works asynchronously, which is necessary in enabling a game to run smoothly. The results show that computation-offloading can be beneficially used to increase the performance of sophisticated mobile games while keeping it playable. However, as trade-off it becomes necessary to delay user inputs. With the current design, the delay can be too long for high speed action games, but is still acceptable for many other game genres, including real-time strategy games. The results also show that offloading may either decrease or increase the power consumption of the mobile device, depending on the performance of the device without offloading, and on how the game developer or player appreciate the trade-off between performance and power consumption.

References

[1] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[2] A.-C. Olteanu and N. Ţăpuş, "Offloading for mobile devices: A survey," *UPB Scientific Bulletin*, 2014.

[3] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: a partition scheme," in *Proceedings of the 2001 int. conf. on Compilers, architecture, and synthesis for embedded systems*. ACM, 2001, pp. 238–246.

[4] H.-h. Chu, H. Song, C. Wong, S. Kurakake, and M. Katagiri, "Roam, a seamless application framework," *Journal of Systems and Software*, vol. 69, no. 3, pp. 209–226, 2004.

[5] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Int. Conf. on Mobile Computing, Applications, and Services*. Springer, 2010, pp. 59–79.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th int. conf. on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[7] Monsoon Power Inc, "Power Monitor, www.msoon.com."

[8] M. Claypool, "The effect of latency on user performance in real-time strategy games," *Computer Networks*, vol. 49, no. 1, pp. 52–70, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.04.008

[9] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu, "The effect of latency on user performance in Warcraft III," in *Proceedings of the 2nd workshop on Network and system support for games*. ACM, 2003, pp. 3–14.