10.1515/acsc-2017-0019

*Archives of Control Sciences*
*Volume* 27(LXIII), 2017
*No.* 2, *pages 293–307*

# Mobile devices and computing cloud resources allocation for interactive applications

HENRYK KRAWCZYK and MICHAŁ NYKIEL

Using mobile devices such as smartphones or iPads for various interactive applications is currently very common. In the case of complex applications, e.g. chess games, the capabilities of these devices are insufficient to run the application in real time. One of the solutions is to use cloud computing. However, there is an optimization problem of mobile device and cloud resources allocation. An iterative heuristic algorithm for application distribution is proposed. The algorithm minimizes the energy cost of application execution with constrained execution time.

**Key words:** mobile devices, computing cloud, task allocation, optimization.

## 1. Mobile cloud computing

The capabilities of mobile devices are increasing at a very fast pace. Average computing power of smartphone CPU increased almost 4 times between 2011 and 2014 [1]. Access to the Internet is a fundamental feature of mobile devices today, and the connection speed of GSM networks almost tripled in 2013 - average download rate was around 520 Kbps in 2012, and almost 1.4 Mbps in 2013 [4]. It is estimated that by the year 2018 there will be more than 7.4 billion mobile devices with constant access to 3G or 4G networks, and global traffic in these networks will exceed 15 exabytes per month. It is a result of the continuous growth of user demand, expecting direct access to Internet services, multimedia, social networks and others.

While the computing power, memory capacity, size and resolution of the screens, and number of various sensors in mobile devices grew rapidly, there is no similar development of battery capacity. For example, the first iPhone, presented in 2007, had battery capacity of 5180 mWh, and the iPhone 5s that debuted in 2013 had a 5960 mWh battery. The huge increase in computing power in mobile devices corresponded with only a 15% gain in battery capacity [20].

One of solutions for this problem is integration between the computing cloud and a mobile device [19]. Thanks to the new generation of mobile networks, applications are

The Authors are with Gdańsk University of Technology, Poland. E-mails: {hkrawk, micnykie}@pg.gda.pl. H. Krawczyk is the corresponding author.

Received 20.12.2016. Revised 29.04.2017.

able to transmit significant amounts of data to services in the cloud. The idea of using resource-rich remote machines to extend the capabilities of smartphones and tablets, allows one to run computationally expensive tasks, while using minimal amounts of device energy.

Many models and architectures for integrating mobile devices with the cloud were proposed in the literature [18]. They differ by scope of integration and method of application optimization – some were designed to maximize application performance [3], others focus on minimizing energy consumption. Some solutions require manual optimization, by annotating which components should be run on the cloud, and which on the mobile device [15]. There are also a few frameworks that are using multi-objective optimization [24]. Unfortunately, most of the proposed solutions require significant changes in the mobile operating system [21].

Analysis of existing solutions in the field of mobile cloud computing leads to the conclusion that there are a few common problems and challenges:

1. Implementing the mobile-cloud integration requires using specific architecture, patterns, complier or language; so an existing application must be rewritten for the most part.

2. Application partition must be often done manually, and automatic methods usually take into account only one objective.

3. Software frameworks require significant modifications in the mobile device's operating system, which is not practical in real applications.

The solution proposed in this paper is an attempt to solve these problems and at the same time accommodate experience from existing research in the field. The main difference from frameworks like Clonecloud [3] or ThinkAir [11] is that the proposed solution focuses on energy optimization while keeping the execution time under a specified value. It also does not require any changes in the mobile operating systems, unlike the abovementioned Clonecloud [3] or Cloudlets [21]. In contrast to eXCloud [14] or $\mu$Cloud [15], the optimization and allocation is fully automatic.

## 2. Interactive application model

Almost all mobile applications have some form of graphical user interface (*GUI*), and can be described as interactive applications; like games, social media, or different kinds of content editing applications. The user is performing actions during the whole life cycle of the application, inputs data, modifies the state in a nondeterministic way, and the application reacts to these actions. Optimization of this type of application is a bigger challenge than optimizing some batch programs, where the input is provided at the start of execution, and data flow is predictable.

Interactive applications are most commonly based on asynchronous events, coming directly from the user, like mouse clicks, button presses, touches of the screen, but also indirect interactions like timer events or network messages. In the case of the mobile applications events from various sensors should also be considered, e.g. changes of the device orientation, or the GPS location.

In response to these events, the application is presenting the output, typically in a visual way. Event handling and rendering of the GUI are implemented with the event queue and the event loop, which periodically consumes and processes events. Modern mobile applications are targeting to render 60 frames per second, which means that every event must be processed in under 16ms.

Obviously, there are events that cannot be processed in that time because, for example, the event requires some interaction with remote services. Hence, the event handling is usually done asynchronously, i.e. the actual processing is performed by threads different than GUI rendering. Only after the processing is finished, the event with results is sent to the event queue and it causes re-rendering of the interface. The concept of the event loop is presented in figure 1.

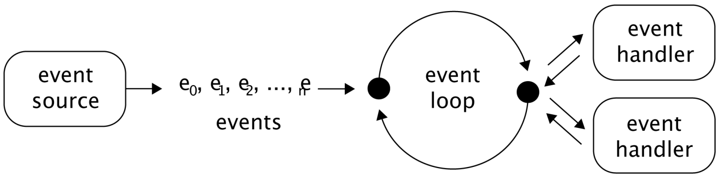


Figure 1: The event handling loop

The proposed application model is based on the functional-reactive programming (*FRP*) paradigm. The core concept of this paradigm is the reactive, instead of proactive, approach to data flow, i.e. reacting to events instead of passing messages to functions. The FRP is similar to the classic observer pattern, but the event flow is modeled with *event streams*, also called *observables* or *signals* in some programming languages. Event processing is done strictly by so called *operator functions*, which are adopted from the functional programming paradigm, e.g. mapping, filtering or reducing.

One of the first programming languages that introduced the FRP concept was Haskell [17]. However, it was never broadly used in programming interactive applications, most likely because of its purely functional nature. One of the new languages that is supposed to target the FRP paradigm is Elm [6]. There are also many frameworks and libraries for popular programming languages that make it easy to use FRP concepts, like Reactive Extensions [13] for C# or RxJS [16] and Cycle.js [23] for JavaScript.

Asynchronous events could be modeled as streams, i.e. sequences of events happening over time [7]. Event stream $S_e$ could be interpreted as a sequence of events $v_i$. The values of $v_i$ could be arbitrary, and the sequence could be finite, infinite or empty:

$$S_e = \langle v_i, v_{i+1}, v_{i+2}, \ldots \rangle \tag{1}$$

In interactive applications we could distinguish three types of streams:

- input streams - event are generated outside the application and processed inside it,
- internal streams - events are generated and processed inside the application,
- output streams - events are generated inside the application but processed outside.

An example of an input stream is the stream of user interactions, e.g. the stream of touch events. In this case the values are coordinates $(x,y)$ of the finger touch, and $x_{max}$ i $y_{max}$ is the width and height of the screen in pixels:

$$\begin{gathered} S_{touch} = \langle(307,204),(521,149),(122,501),\ldots\rangle \\ x \in [0,x_{max}], y \in [0,y_{max}] \end{gathered} \tag{2}$$

Other examples of input streams include network messages, timers or various types of system interruptions.

Internal event streams are modeling the data flow inside the application. They are generated by transforming one input stream, or a combination of multiple inputs. For example, a stream of user interface element interactions can be generated from the screen touch events. In this case the event values are identifiers of the GUI element:

$$S_{action} = \langle(home\_button),(back\_button),(text\_input),\ldots\rangle \tag{3}$$

The application is usually generating multiple output streams. The most common one is the stream of the graphical user interface state, which can be represented as the following sequence of matrices presenting pixel colors $p_{j,k}$:

$$S_{frames} = \langle \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_i , \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_{i+1} ,\ldots\rangle \tag{4}$$

The important characteristic of all event streams is that they are immutable, i.e. the event value $v_i$ is constant during the whole application lifecycle.

Input streams $S_{in}$ are processed and eventually converted to output streams $S_{out}$ with functions $f$ called operators. Theoretically, the whole application can be modeled as a single operator such as:

$$S_{out} = f(S_{in}) \tag{5}$$

In practice, the application is divided into multiple operators with smaller scope, however all of them have a common signature, i.e. they convert input streams to output streams. One exception to that rule is the merge operator $f_{merge}$, that just combines multiple streams into one:

$$S_{out} = f_{merge}(S_0,S_1,\ldots,S_n) \tag{6}$$

Operators can be divided into two categories, *pure* and *impure*, analogically to functions. Pure operators do not have any side-effects, and are stateless. It is a very convenient property of an operator, because it guarantees that for any input event value $v$, the operator will always produce the same result $w$:

$$f(\langle v_i, v_{i+1}, \dots \rangle) = \langle w_i, w_{i+1}, \dots \rangle : v_i = v_j \implies w_i = w_j \\ i \neq j \wedge i, j = 0, 1, 2, \dots \tag{7}$$

Thanks to this property, pure operators are deterministic, therefore it is easy to cache the results, and test the correctness of the operator. Regarding the application optimization, and partitioning the application to mobile devices and the cloud, the most important feature of a pure operator is their independence from the rest of the application. In other words, they can be isolated and moved to other environments without migrating any extra state.

Two primary examples of pure operators are mapping and filtering. Mapping operator $f_{map}$ is essentially a function in a mathematical sense, that for every input value $v_i$ assigns an output value $g(v_i)$:

$$f_{map}(\langle v_i, v_{i+1}, \dots \rangle) = \langle g(v_i), g(v_{i+1}), \dots \rangle \tag{8}$$

The filtering operator $f_{filter}$ generates an event stream with unchanged values, however some values can be omitted:

$$f_{filter}(\langle v_i, v_{i+1}, \dots \rangle) = \langle v_i : g(v_i) > 0 \rangle \tag{9}$$

For practical reasons, not all operators could be pure and stateless. The most common operator that requires storing an internal state is the accumulation operator $f_{scan}$, sometimes called scan operator:

$$f_{scan}(\langle v_0, v_i, v_{i+1}, \dots \rangle) = \langle w_i : w_o = h(v_0, 0) \wedge w_i = h(v_i, h(v_{i-1})) \rangle \tag{10}$$

The interaction between the computer and the user is a two-way process in which both sides produce and consume data, described in literature as *human-computer interaction (HCI)*. The process is cyclical, however there is no strict order of the interactions, i.e. the user can execute multiple actions without response from the computer, and the computer can produce output without analyzing any input from the user.

In practice all interactions between the user and the computer are done through various I/O (*input/output*) devices. The computer may be modeled as a system, which includes I/O drivers and an application working in the operating system environment. The application is not interacting directly with the user, instead it is communicating with an abstraction layer provided by the operating system. Hence, from the application's point of view, we can simplify this model to the interaction between an application and an operating system. The communication is usually done in an asynchronous way, in the form of events, signals or interruptions. In consequence the interaction between the application and the system can be modeled as a cycle of stream processing.

Although the whole interaction is a cyclic and nondeterministic process, the event processing performed inside the application can be modeled as a clearly defined directed acyclic graph (*DAG*), in which nodes are operators, and edges are internal event streams. The graph represents all possible execution flows for a single iteration – a time between a single user interaction and the application outputting a new state (result). It is important that although two operators are connected with an edge (stream) they not necessarily have to produce and consume an event in every iteration. Hence, in the lifecycle of the application some operators may be executed more frequently than others. An example model of an application, namely a mobile chess game, is presented in figure 2.

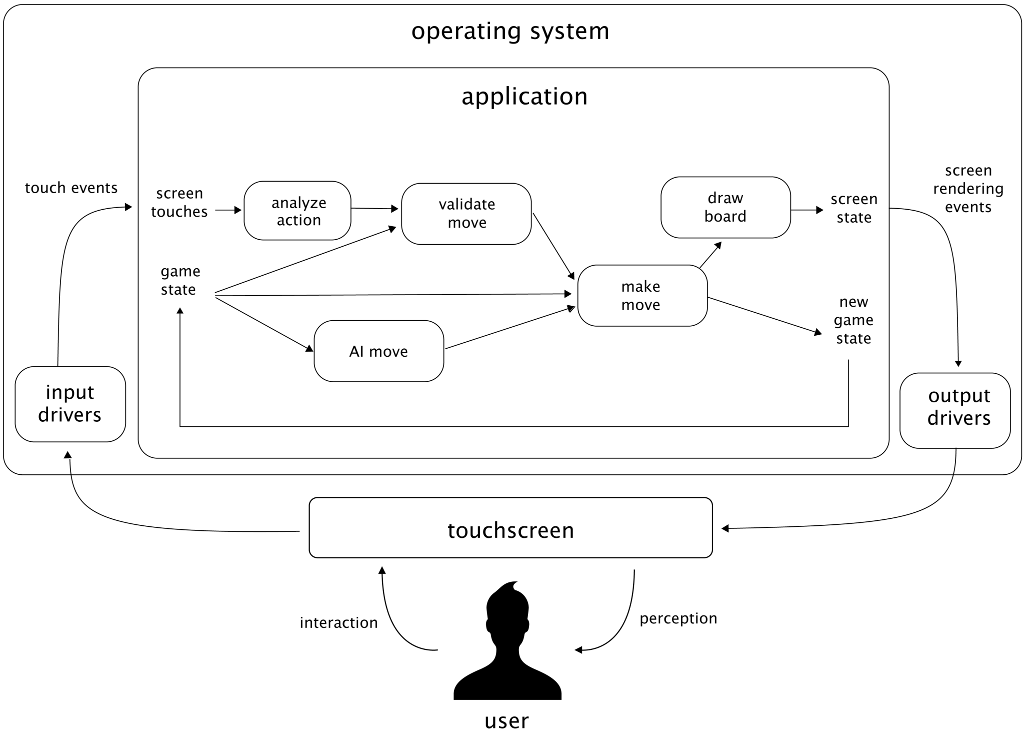


Figure 2: Model of a chess game application

The application contains one input stream – screen touches. It is important to note that the application does not have any global state, instead the game state is passed inside the application in the form of a cyclic stream. There are $N = 5$ operators identified in the application. The screen touches are mapped to actions (*analyze action*), i.e. certain moves of the figures on the chessboard. From the stream of potential figure moves only the valid moves are selected (*validate move*), and then merged with the current state of the game, which in turn generates the new state of the game (*make move*). After every other generated state the computer move is produced (*AI move*). In order to do this, the state stream is mapped to an optimal move – computed by the AI algorithm – and merged with the aforementioned potential figure moves stream. Lastly, the current state of the game is drawn as a chessboard on the screen (*draw board*).

It is possible to split operators further, and obtain a graph with higher granularity. Especially the *AI move* operator, seems like a quite complex mapping function, and thus a good candidate to divide into a group of simpler operators. However, due to the implementation of the AI algorithm (a variant of the Minimax algorithm [10]), the resulting set of operators would be tightly connected, with significant amounts of communication between them, so moving some of those operators to the cloud would most likely result in lower performance of the application.

## 3. Problem description and solution space

Optimization of a mobile application, modeled as described in section 2, relies on partitioning the operators into two sets: those executed on the mobile device and those executed in the computing cloud. The objective of the optimization is to minimize the cost $c$ of application execution and execution in the assumed time-frame $t_{max}$. The cost is interpreted as the sum of all operators' costs:

$$C = \sum_{n=0}^{N-1} c(f_n) \tag{11}$$

In general, the cost is proportional to the CPU time used for executing the operator, and may be also interpreted as the energy required to run the operator. Because some of the operators may be executed in parallel on multi-core processors, the execution time is equal to the critical path in the application model.

Theoretically, the cost and time of execution of every operator may be infinite, because the number of cycles in an interactive application may be infinite. Furthermore, the number of cycles cannot be estimated, because it depends on nondeterministic behavior of the user. In order to find the exact solution of the optimization, every possible combination of events should be considered. For instance, in the chess game application there are 20 possibilities for the first move, and the number of possible game states is increasing exponentially. The game finally ends in one of three states: white (user) win, black (computer) win, or draw. The illustration of the game iterations is presented in figure 3.

The number of all possible games $P$ for $I$ iterations can be calculated based on the number of possible states $m$ on every turn:

$$P = m_0 \cdot m_1 \cdot \ldots \cdot m_{I-1} \tag{12}$$

The average number of turns in a chess game is estimated at around 40 for advanced players [2]. In every turn there are two iterations – white player move and black player move – so we can estimate that there are around 80 iterations in a typical chess game.

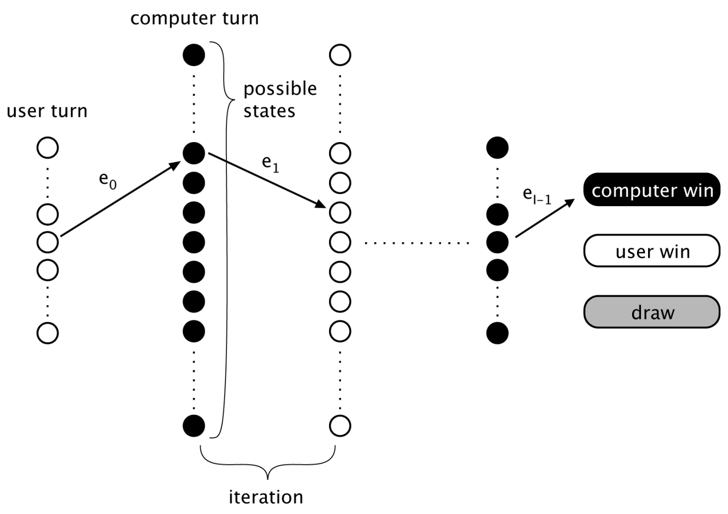


Figure 3: Illustration of chess game iterations

The number of possible states can be different for every turn because of the varying number and positions of the figures. However, the average number of states is estimated at around 30 throughout the whole game [22]. Therefore, the number of possible chess games could be estimated as:

$$P \approx \bar{m}^I = 30^{80} \approx 10^{118} \tag{13}$$

Considering that number of possible games, finding the exact optimization of the whole game is impossible in a reasonable time. Hence, we will analyze the cost and the time of a single iteration, i.e. execution of all operators between subsequent input events:

$$C = \sum_{i=0}^{I-1} \sum_{n=0}^{N-1} c(f_n(e_i)) \tag{14}$$

Despite the fact that, for every event, the cost and the time of executing the operator may be different, the average cost and time of a single iteration is equal to the sum of average costs and execution times of all operators:

$$\bar{C_{it}} = \frac{1}{I} \sum_{i=0}^{I-1} C_i = \frac{1}{I} \sum_{i=0}^{I-1} \sum_{n=0}^{N-1} c(f_n(e_i)) \tag{15}$$

Therefore, in order to minimize the cost of application execution, we must minimize the average cost of executing every operator. Taking into consideration the fact that operators may be executed in two environments – a mobile device or the cloud – the cost and time of the data transfer must be included in the calculations. The easiest way to achieve that is to model the communication as an additional *transfer operator*, that is added on every internal stream which is transferred between the environments.

## 4. Iterative optimization algorithm

To perform the exact optimization of an interactive application: first, we would have to construct a graph that covers all the iterations, i.e. an $I$ number of connected operator graphs. Therefore, the optimization must be performed on a graph that has $N * I$ nodes, which would be very expensive for a high number of iterations, even when using a heuristic algorithm [12].

However, the iterative nature of interactive applications allows you to effectively use a greedy optimization strategy. In every iteration the algorithm finds the single most profitable transition, i.e. the operator that, when moved to the other environment, would have the maximum impact on lowering the summary execution cost, and still keep the execution time in the assumed time-frame. We expect that after some number of iterations, that is significantly lower than the total number of iterations for the given application, the heuristic algorithm should achieve near-optimal application partition.

```
function OPTIMIZE(operators)
    $dc_{max} \leftarrow 0$
    $operator_{max} \leftarrow null$
    for all operators do
        if operator in cloud then
            $dt \leftarrow mobileTime(operator) - cloudTime(operator)$
            if $t + dt \leqslant t_{max}$ then
                $dc \leftarrow cloudCost(operator) - mobileCost(operator)$
                if $dc > dc_{max}$ then
                    $dc_{max} \leftarrow dc$
                    $operator_{max} \leftarrow operator$
        else
            $dt \leftarrow cloudTime(operator) - mobileTime(operator)$
            if $t + dt \leqslant t_{max}$ then
                $dc \leftarrow mobileCost(operator) - cloudCost(operator)$
                if $dc > dc_{max}$ then
                    $dc_{max} \leftarrow dc$
                    $operator_{max} \leftarrow operator$
    return $max_{operator}$
```

The algorithm pseudocode is presented in the listing. It returns an identifier of an operator that should be moved from the mobile device to the cloud or vice versa. The $mobileTime()$, $mobileCost()$, $cloudTime()$ and $cloudCost()$ functions return costs and execution times for a given operator computed from the previous operator calls. We assume that initial costs and times are equal to 0. The proposed algorithm's time complexity for a single iteration is $O(N)$ depending on the number of operators.

A software framework and management module is required to enable dynamic allocation of an operator to the cloud or a mobile device in every iteration. Such an offloading framework was developed, and a testing environment was set up, in order to test the algorithm. The framework was built on top of RxJS [16] and CycleJS [23] libraries that support developing interactive applications with architecture based on event streams.

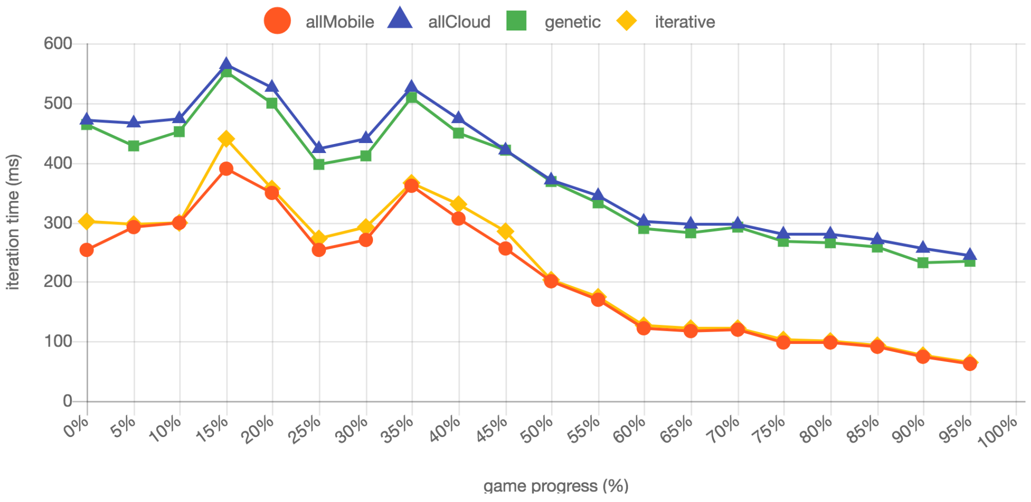
The proposed iterative heuristic was also compared to the genetic optimization algorithm [12] that calculated operator allocation a priori, before the application execution. The genetic algorithm was proposed for static applications, where execution times are constant throughout the lifecycle of the application, so the allocation for given operator is also constant. As the algorithm requires historical data to perform the optimization, operator execution and transfer times for 20 games were measured and averaged.

The tests were performed for the implemented chess application corresponding to presented one in figure 2. Four methods of application allocation were evaluated:

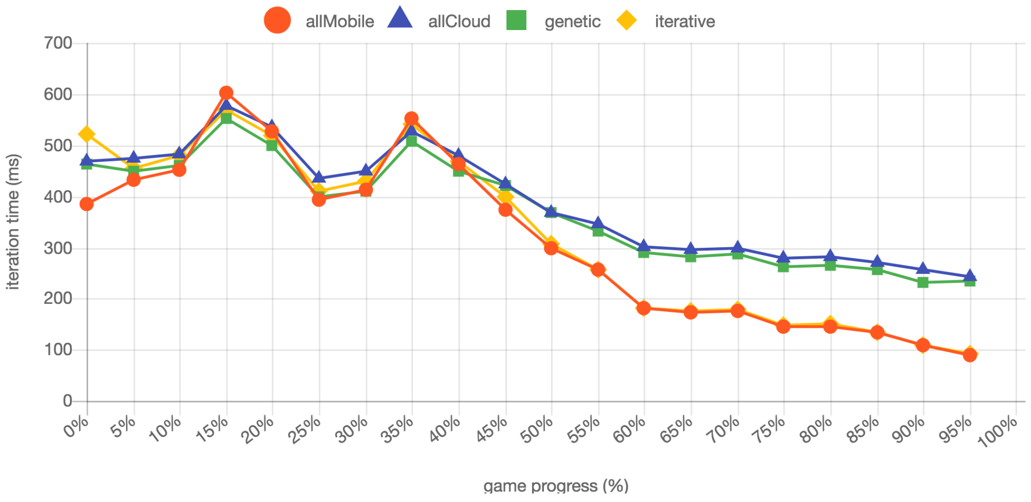
- *allMobile* - the application running only on the mobile device,
- *allCloud* - all operators running in the cloud, with the mobile device acting as a thin client,
- *genetic* - operators allocated to the mobile device or to the cloud before the application execution by the heuristic genetic algorithm, optimization based on the average costs for all iterations
- *iterative* - operators allocated to the mobile device or to the cloud on every iteration by the heuristic optimization algorithm proposed above.

The tests were run on a mobile device emulator. As the performance of the actual mobile devices vary greatly between different models, several ratios of the cloud computing performance to the mobile device performance were simulated, ranging from the 1:1 ratio (the same performance on both environments) to 3:1 ratio (3 times faster computation on the cloud). The network latency, for communication between the device and the cloud, was set to 200ms. The charts presented in figure 4 show the varying iteration duration (in milliseconds) throughout the whole game, from the beginning (0%) until the end (100%). The data was averaged from 20 different chess games.

The average duration of the whole game for different cloud to mobile performance ratios and allocation configurations are presented in table 1. Although the iterative optimization algorithm does not give the optimal solution in every case, the difference from the best allocation is always relatively small. Hence, with the iterative optimization algorithm enabled, and with dynamic operator allocation, the average game duration across all the cases is 13.4% lower than the *allMobile* allocation, and 12.5% lower than the *allCloud* one. The *genetic* optimization algorithm favored moving the single operator with the highest cost to the cloud (*AI move*), but because this algorithm does not consider changing execution times, the results are comparable to the *allCloud* scenario.



(a) 1:1 performance ratio

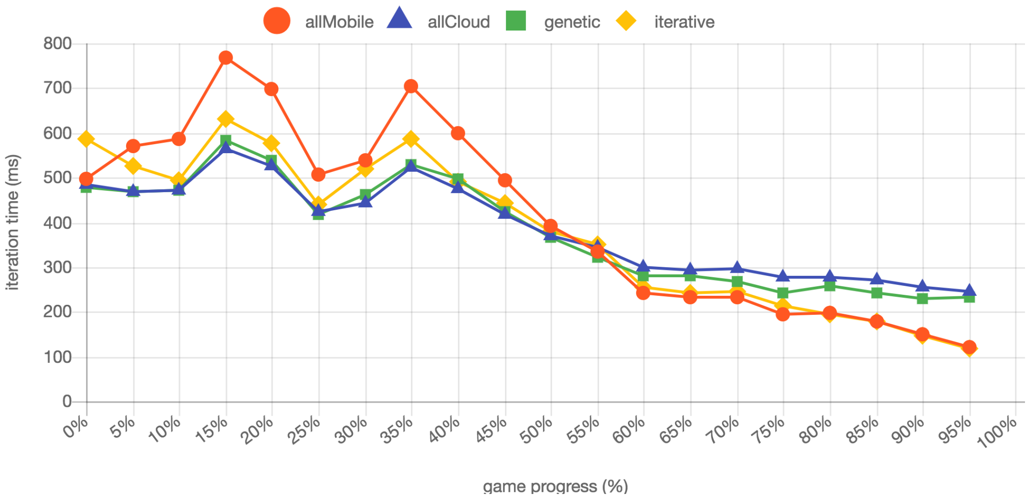


(b) 1.5:1 performance ratio

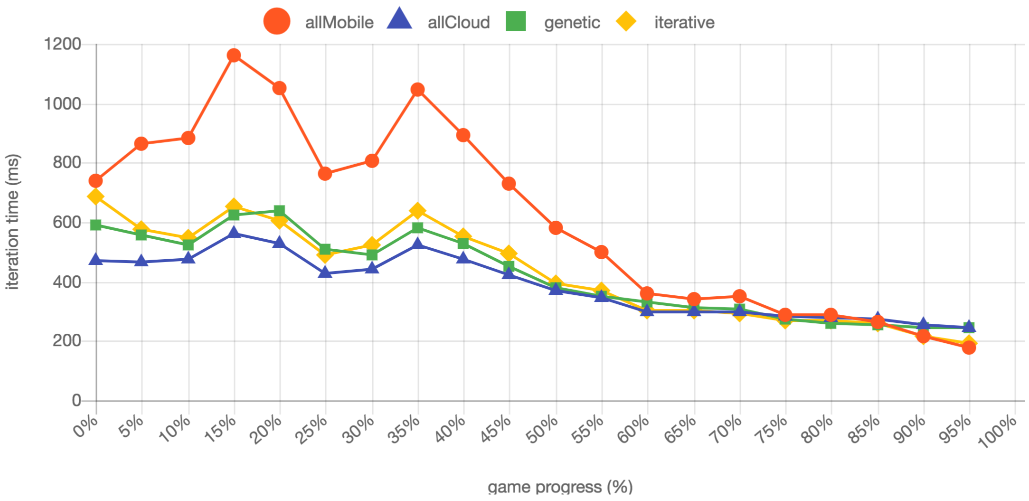
Figure 4: Iteration duration (in milliseconds) throughout the game

## 5. Conclusions

Optimization of an interactive mobile application using cloud computing has three possible results: all operators are executed on the device, all operators are offloaded to the cloud, or the operator graph is partitioned between those two environments. The example



(c) 2:1 performance ratio



(d) 3:1 performance ratio

Figure 4: Iteration duration (in milliseconds) throughout the game

chess game application showed that using only the mobile device is not optimal, because of the complex processing involved in computing the AI moves. Offloading the whole application to the cloud is also not ideal, because of the significant output data size for the *draw board* operator. In fact, the optimal solution is to move only one operator to

Table 1: Average game durations

| **performance ratio** | **allMobile** | **allCloud** | **genetic** | **iterative** |
|---:|---:|---:|---:|---:|
| 1:1 | 18.60 | 34.77 | 33.31 | 19.70 |
| 1.5:1 | 27.99 | 35.08 | 33.40 | 28.94 |
| 2:1 | 36.63 | 34.78 | 34.19 | 33.71 |
| 3:1 | 54.72 | 34.73 | 37.97 | 38.34 |
| average | 34.49 | 34.84 | 35,88 | 30.17 |

the cloud, for the middle part of the game, hence the iterative heuristic algorithm and dynamic operator allocation are suitable in this case.

## References

[1] H. BAUER, Y. GOH, S. SCHLINK and C. THOMAS: The supercomputer in your pocket. *McKinsey on Semiconductors*, (Autumn), (2012), 14-27,

[2] Chessgames Services LLC: Chess Statistics.
http://www.chessgames.com/chessstats.html, 2016.

[3] B. CHUN, S. IHM, P. MANIATIS, M. NAIK and A. PATTI: Clonecloud: elastic execution between mobile device and cloud. *Proc. of the 6th Conf. on Computer Systems*, (2001), 301-314.

[4] Cisco, Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018. 2014.

[5] E. CUERVO. A. BALASUBRAMANIAN, D. CHO, A. WOLMAN, S. SAROIU, R. CHANDRA and P. BAHL: MAUI: making smartphones last longer with code offload. *Proc. of the 8th Int. Conf. on Mobile Systems, Applications, and Services*, (2010), 49-62.

[6] E. CZAPLICKI: Elm: Concurrent FRP for Functional GUIs. Senior thesis, Harvard University, 2012.

[7] O. ETZION and P. NIBLETT: Event Processing in Action. Manning Publications Co., 2010.

[8] J.X. HAO and J.B. ORLIN: A faster algorithm for finding the minimum cut in a directed graph. *J. of Algorithms*, **17**(3), (1994), 424-446.

[9] R. Kemp, N. Palmer, T. Kielmann and H. Bal: Cuckoo: a computation offloading framework for smartphones. Mobile Computing, Applications, and Services. Springer, 2010, 59-79.

[10] R.E. Korf and D.M. Chickering: Best-first minimax search. *Artificial Intelligence*, **84**(1), (1996), 299-337.

[11] S. Kosta, A. Aucinas, P. Hui, R. Mortier and X. Zhang: Unleashing the power of mobile cloud computing using thinkair. arXiv:1105.3232 [cs.DC], (2011).

[12] H. Krawczyk, M. Nykiel and J. Proficz: Mobile offloading framework: Solution for optimizing mobile applications using cloud computing. *Computer Networks*, Springer International Publishing, (2015), 293-305.

[13] J. Liberty, P. Betts and S. Turalski: Programming Reactive Extensions and LINQ. Springer, 2011.

[14] R.K. Ma, K.T. Lam and C. Wang: eXCloud: Transparent runtime support for scaling mobile applications in cloud. *Int. Conf. on Cloud and Service Computing*, (2011), 103-110.

[15] V.March, Y. Gu, E. Leonardi, G. Goh, M. Kirchberg and B.S. Lee: $\mu$cloud: towards a new paradigm of rich mobile applications. *Procedia Computer Science*, **5** (2011), 618-624.

[16] Microsoft Open Technologies: The Reactive Extensions for JavaScript. https://github.com/Reactive-Extensions/RxJS, 2016.

[17] H. Nilsson, A. Courtney and J. Peterson: Functional reactive programming, continued. *Proc. of the 2002 ACM SIGPLAN Workshop on Haskell*, (2002), 51-64.

[18] M. Othman, S.A. Madani and S.U. Khan: A survey of mobile cloud computing application models. *IEEE Communications Surveys & Tutorials*, **16**(1), (2014), 393-413.

[19] A. Pathak, C. Hu, M. Zhang, P. Bahl and Y. Wang: Enabling automatic offloading of resource-intensive smartphone applications. Purdue University, Electrical and Computer Engineering Technical Report, 2011.

[20] M. Satyanarayanan: Mobile computing: the next decade. *Proc. of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, (2010).

[21] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies: The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, **8**(4), (2009), 14-23.

[22] C. SHANNON: Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and J. of Science*, **41**(314), (1950), 256-275.

[23] A. STALTZ: Cycle.js – a Functional and Reactive JavaScript Framework for Cleaner Code. http://cycle.js.org/, 2016.

[24] X. ZHANG, A. KUNJITHAPATHAM, S. JEONG and S. GIBBS: Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Networks and Applications*, **16**(3), (2011), 270-284.