Henryk KRAWCZYK, Michał NYKIEL
Politechnika Gdańska

OPTYMALIZACJA WSPÓŁPRACY URZĄDZENIA MOBILNEGO I CHMURY OBLICZENIOWEJ DLA EFEKTYWNEGO WYKONANIA ZŁOŻONYCH APLIKACJI INTERAKTYWNYCH

Streszczenie. Wykonywanie różnego typu aplikacji interaktywnych na urządzeniach mobilnych takich jak smartfony, czy iPady jest obecnie bardzo powszechne. W przypadku złożonych aplikacji, jak np. gra w szachy, możliwości obliczeniowe tych urządzeń mogą nie być wystarczające by aplikacja wykonywała się w czasie rzeczywistym. Rozwiązaniem jest wykorzystanie chmury obliczeniowej. Powstaje jednak problem optymalizacji wykorzystania zasobów urządzenia mobilnego i chmury obliczeniowej. Zaproponowano heurystyczny algorytm dynamicznego rozdziału aplikacji, w którym jako kryteria optymalizacji przyjęto minimalizację kosztów c realizacji kolejnych kroków aplikacji oraz wykonania ich w założonym przedziale czasu $\langle 0, t_{max})$.

OPTIMIZING COOPERATION BETWEEN MOBILE DEVICES AND COMPUTING CLOUD FOR EXECUTING COMPLEX INTERACTIVE APPLICATIONS

Summary. Using mobile devices such as smartphones or iPads for various interactive applications is currently very common. In case of complex applications, e.g. chess games, 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. The heuristic algorithm for dynamic application distribution is proposed, where the minimization of subsequent application steps costs c and their execution in assumed time frame $\langle 0, t_{max})$ are used as the optimization objectives.

1. Współpraca z chmurą obliczeniową

Aktualnie możliwości urządzeń mobilnych rozwijają się w bardzo dużym tempie. Średnia moc obliczeniowa procesora w smartfonie wzrosła prawie 4-krotnie pomiędzy rokiem 2011 i 2014 [1]. Dostęp do Internetu jest dzisiaj podstawową funkcją urządzeń mobilnych, a prędkość połączeń w sieciach GSM podwoiła się w roku 2013 - średnia prędkość pobierania danych wynosiła 526 Kbps w 2012 i prawie 1,4 Mbps w 2013 [3]. Szacuje się, że do 2018 roku będzie ponad 7,4 miliarda urządzeń mobilnych posiadających dostęp do sieci 3G i 4G, a globalny ruch sieciowy w tych sieciach przekroczy 15 eksabajtów miesięcznie. Jest to wynikiem nieustannie zwiększającego się zapotrzebo-

Tabela 1

Porównanie dostępnych rozwiązań

	Automatyczna optymalizacja	**Zmiany w aplikacji**	**Zmiany w systemie**	**Inne**
CloneCloud [2]	wydajność	brak	znaczne	-
Weblets [22]	wielokryterialna	znaczne	brak	wymaga projektowania aplikacji w specyficznej architekturze
μCloud [14]	brak	znaczne	brak	-
Cloudlet [18]	brak	brak	znaczne	wymaga serwera z niskim czasem odpowiedzi (prywatnej chmury)
eXCloud [13]	brak	brak	brak	-
MAUI [4]	wielokryterialna	niewielkie	brak	bazuje na przestarzałej wersji systemu operacyjnego
ThinkAir [9]	energia lub wydajność	niewielkie	brak	dedykowany kompilator
Cuckoo [8]	brak	niewielkie	brak	-

wania użytkowników, którzy oczekują ciągłego dostępu do usług Internetowych, multimediów, sieci społecznościowych, itp.

Podczas gdy moc procesorów, pojemność pamięci, wielkość i rozdzielczość ekranu oraz liczba sensorów w urządzeniach mobilnych znacząco wzrosła, podobny rozwój nie jest zauważalny w przypadku pojemności baterii. Dla przykładu pierwszy iPhone, zaprezentowany w 2007 roku, posiadał baterię o pojemności 5180 mWh, natomiast iPhone 5s, wprowadzony na rynek w 2013, posiada baterię o pojemności 5960 mWh. Ogromny wzrost mocy obliczeniowej w urządzeniach mobilnych w tym czasie odpowiada jedynie 15% wzrostowi pojemności baterii [17]

Jednym z rozwiązań tego problemu jest integracja chmury obliczeniowej z urządzeniami mobilnymi [16]. Dzięki nowej generacji sieci komórkowych aplikacje mobilne są w stanie transmitować duże ilości danych do usług w chmurze. Koncepcja wykorzystania zdalnych serwerów i zasobów do rozszerzenia możliwości smartfonów czy tabletów pozawala na wykonywanie złożonych obliczeniowo zadań przy minimalnym wykorzystaniu energii urządzenia.

W literaturze zostało zaproponowanych wiele modeli i architektur integrujących urządzenia mobilne z chmurą obliczeniową. Różnią się one zakresem integracji i celem optymalizacji, niektóre z nich zostały zaprojektowane w celu zmaksymalizowania wydajności przetwarzania [2], inne z kolei, nastawione są głównie na minimalizację wykorzystania energii. Część rozwiązań wymaga ręcznej optymalizacji aplikacji, przez podzielenie jej na komponenty wykonywane w chmurze i na urządzeniu już na etapie implementacji [14]. Istnieje również kilka propozycji, które biorą pod uwagę optymalizację wielokryterialną [22]. Często do wdrożenia większości rozwiązań wymagane są znaczne zmiany w systemie operacyjnym urządzenia [18]. Najbardziej popularne metody zostały przedstawione w tabeli 1.

Analiza istniejących rozwiązań rozdziału obliczeń aplikacji na urządzenie mobilne i chmurę obliczeniową prowadzi do wniosku, że najpowszechniejsze wady wspomnianych modeli są następujące:

1. Realizacja rozdziału wymaga wykorzystania specyficznej architektury, wzorca lub kompilatora, przez co istniejące aplikacje muszę być przepisane na nowo.
2. Na ogół ręczna optymalizacja lub automatyczna podziału aplikacji bierze pod uwagę tylko jedno kryterium.
3. Przyjęte rozwiązanie wymaga znacznych zmian w systemie operacyjnym, przez co jest trudne do wdrożenia w praktycznych zastosowaniach bez wsparcia ze strony producenta systemu operacyjnego.

Rozwiązania przedstawione poniżej wychodzą naprzeciw tym ograniczeniom, uwzględniając doświadczenie i wnioski wynikające z dotychczas opracowanych rozwiązań.

2. Model aplikacji interaktywnej

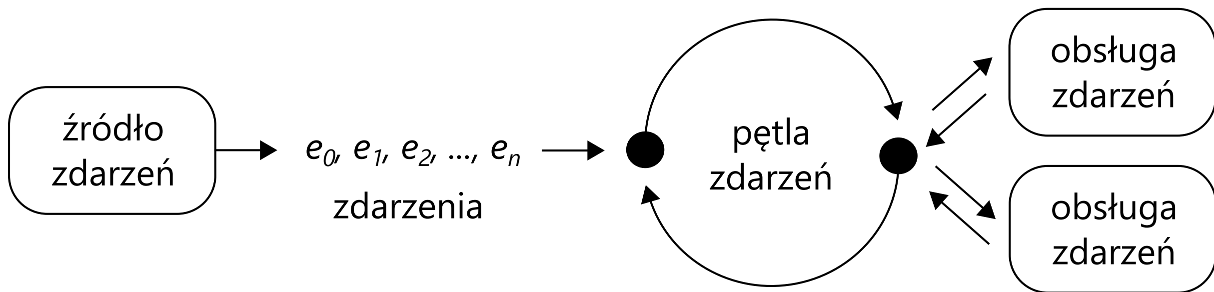
Aplikacje dla urządzeń mobilnych to w ogromnej większości aplikacje z graficznym interfejsem użytkownika, a ogólniej interaktywne, takie jak gry, serwisy społecznościowe czy edytory dokumentów. Użytkownik wykonuje akcje w ciągu całego czasu życia programu, wprowadza dane i zmienia ustawienia w niedeterministyczny sposób. Optymalizacja tego typu aplikacji jest znacznie większym wyzwaniem niż aplikacji, gdzie z góry znane są dane wejściowe, ponieważ przepływ sterowania nie jest możliwy do przewidzenia.

Aplikacje interaktywne z graficznym interfejsem użytkownika opierają się na asynchronicznych zdarzeniach, takich jak kliknięcie myszą czy naciśnięcie klawisza przez użytkownika, ale również zdarzenia czasowe i komunikaty sieciowe. W przypadku urządzeń mobilnych są to również dotknięcia ekranu, zmiana orientacji urządzenia oraz inne zdarzenia pochodzące od pozostałych sensorów urządzenia.

W odpowiedzi na zdarzenia aplikacja przedstawia użytkownikowi dane w sposób graficzny. Obsługa zdarzeń i renderowanie interfejsu są zazwyczaj implementowane za pomocą kolejki i pętli zdarzeń (ang. *event loop*), która cyklicznie pobiera i przetwarza zdarzenia. W dzisiejszych aplikacjach dąży się do renderowania z prędkością 60 klatek na sekundę, co oznacza, że pojedyncze zdarzenie powinno zostać przetworzone w czasie około 16ms.

Z oczywistych względów nie jest możliwe zapewnienie takiego czasu przetwarzania dla każdego zdarzenia. Z tego względu obsługa zdarzeń od użytkownika zazwyczaj odbywa się asynchronicznie, tzn. faktyczne obliczenia obsługiwane są w innym wątku niż pętla zdarzeń. Dopiero po zakończeniu obliczeń do kolejki trafia odpowiednie zdarzenie powodujące faktycznie renderowanie odpowiedzi na interfejsie użytkownika. Schemat działania pętli zdarzeń został przedstawiony na rysunku 1.

W niniejszym artykule zaproponowany został model aplikacji oparty na paradygmacie programowania funkcyjno-reaktywnego, tzw. FRP (ang. *Functional Reactive Programming*). Podstawą FRP jest reagowanie na zdarzenia (stąd reaktywne), podobnie jak w przypadku wzorca obserwatora w klasycznym programowaniu obiektowym. Różnicą w stosunku do klasycznego podejścia jest jawne modelowanie wartości zmiennych



Rys. 1. Pętla zdarzeń

w czasie za pomocą strumieni, zwanych również sygnałami. Reakcja na zmianę wartości odbywa się za pomocą funkcji (operatorów) wywiedzionych wprost z paradygmatu funkcyjnego, np. operator mapowania, filtrowania czy redukcji.

Klasycznym językiem programowania, w którym pierwszy raz pojawiło się pojęcie FRP jest Haskell [15]. Jednak ze względu na to, że jest to język czysto funkcyjny nie znalazł on szerszego zastosowania przy programowaniu aplikacji interaktywnych. W ostatnim czasie pojawiły się jednak nowe języki, takie jak np. Elm [5], a także biblioteki i frameworki do popularnych języków takich jak C# (Reactive Extensions [11]) oraz JavaScript (RxJS [21] oraz Cycle.js [20]).

Asynchroniczne zdarzenia można modelować jako strumienie, czyli sekwencje zdarzeń następujących w czasie [6]. Strumień zdarzeń S_e można interpretować jako ciąg par (t_i, v_i), gdzie t_i oznacza punkt w czasie, w którym nastąpiło i-te zdarzenie, a v_i oznacza wartość tego zdarzenia. Wartości t_i są rosnące i dodatnie, v_i może być dowolne, ciąg zdarzeń może być nieskończony, jak również pusty, tzn:

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

W aplikacji interaktywnej można rozróżnić trzy rodzaje strumieni:

- wejściowe - zdarzenia generowane poza aplikacją i przetwarzane w ramach aplikacji,
- wewnętrzne - zdarzenia generowanie i przetwarzane w aplikacji,
- wyjściowe - zdarzenia generowanie w aplikacji i przetwarzane poza nią.

Przykładowym strumieniem wejściowym są wszystkie interakcje użytkownika, np. strumień dotknięć ekranu. W tym wypadku wartościami zdarzenia są współrzędne miejsca (x, y), gdzie ekran został dotknięty, a x_{max} i y_{max} to odpowiednio szerokość i wysokość ekranu w pikselach np.:

$$\begin{aligned} S_{touch} = \langle (0.1, (307, 204)), (0.7, (521, 149)), (1.3, (122, 501)), \ldots \rangle \\ x \in [0, x_{max}], y \in [0, y_{max}] \end{aligned} \tag{2}$$

Innymi przykładami strumieni wejściowych zdarzeń są pakiety sieciowe, różnego rodzaju timery lub przerwania systemowe.

Strumienie wewnętrzne modelują przepływ danych wewnątrz aplikacji. Najczęściej są generowane przez transformację pewnego strumienia wejściowego lub ich kombinacji. Przykładowo, ze strumienia dotknięć ekranu może zostać wygenerowany strumień dotknięć elementów w graficznym intefejsie użytkownika. W tym wypadku war-

tościami zdarzenia są identyfikatory elementów:

$$S_{action} = \langle (0.1, home_button), (0.7, back_button), (1.3, text_input), \dots \rangle \quad (3)$$

Aplikacja zazwyczaj generuje kilka strumieni wyjściowych. Podstawowym jest strumień stanu interfejsu lub wręcz strumień macierzy pikseli, które mają zostać wyświetlone na ekranie, w zależności czy silnik renderujący jest częścią systemu operacyjnego czy częścią aplikacji (najczęściej w przypadku gier). W drugim przypadku strumień generowany ze stałą częstotliwością zdarzeń, np. 60 razy na sekundę:

$$S_{frames} = \langle (0, \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_i), (0.17, \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_{i+1}), \dots \rangle \quad (4)$$

Istotną cechą wszystkich strumieni zdarzeń jest to, że są one niemutowalne, tzn. wartość v w strumieniu w danym punkcie czasu t jest stała przez cały cykl życia aplikacji.

Strumienie wejściowe S_{in} przetwarzane są na strumienie wyjściowe S_{out} za pomocą funkcji f, zwanych operatorami. W ogólności można zamodelować całą aplikację interaktywną jako jeden operator:

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

W praktyce aplikację dzieli się na wiele mniejszych operatorów, jednak prawie wszystkie mają taką samą sygnaturę, a więc przetwarzają strumienie na inne strumienie. Wyjątkiem są operatory scalenia, które łączą wiele strumieni w jeden:

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

Operatory można podzielić na czyste (ang. *pure*) oraz nieczyste (ang. *impure*), analogicznie jak w przypadku funkcji. Operatory czyste nie mają żadnych efektów ubocznych oraz nie przechowują stanu. Jest to bardzo pożądana właściwość operatora, ponieważ zachowuje on wtedy właściwości funkcji matematycznej, tzn. dla danej wartości zdarzenia v zawsze zwraca ten sam wynik, bez względu na czas t:

$$\forall i, j \in \mathbb{N} \wedge f(\langle (t, v)_i, (t, v)_{i+1}, \dots \rangle) = \langle (t', w)_i, (t', w)_{i+1}, \dots \rangle : v_i = v_j \implies w_i = w_j \\ i \neq j \wedge i, j = 0, 1, 2, \dots \quad (7)$$

Dzięki tej własności czyste operatory są przewidywalne, a więc można np. stosować techniki zapamiętywania wyników w pamięci podręcznej (ang. *cache*) oraz w łatwy sposób testować poprawność działania. Z punktu widzenia optymalizacji i rozdziału aplikacji na urządzenie mobilne i chmurę obliczeniową najważniejszą cechą czystych operatorów jest ich niezależność, tzn. możliwość izolacji i przeniesienia w inne środowisko bez konieczności migracji stanu pamięci.

Do czystych operatorów można zaliczyć operator mapowania (selekcji) oraz filtrowania. Mapowanie polega na przyporządkowaniu każdej wartości v_i ze strumienia wejściowego dokładnie jednej wartości za pomocą funkcji tzw. selektora $g(v)$:

$$f_{map}(\langle (t_i, v_i), (t_{i+1}, v_{i+1}), \dots \rangle) = \langle (t'_i, g(v_i)), (t'_{i+1}, g(v_{i+1})), \dots \rangle \quad (8)$$

Operator filtrowania generuje strumień z niezmienionymi wartościami zdarzeń, niektóre zdarzenia mogą natomiast zostać pominięte:

$$f_{filter}(\langle (t_i, v_i), (t_{i+1}, v_{i+1}), \dots \rangle) = \langle (t'_i, v_i) : g(v_i) > 0 \rangle \quad (9)$$

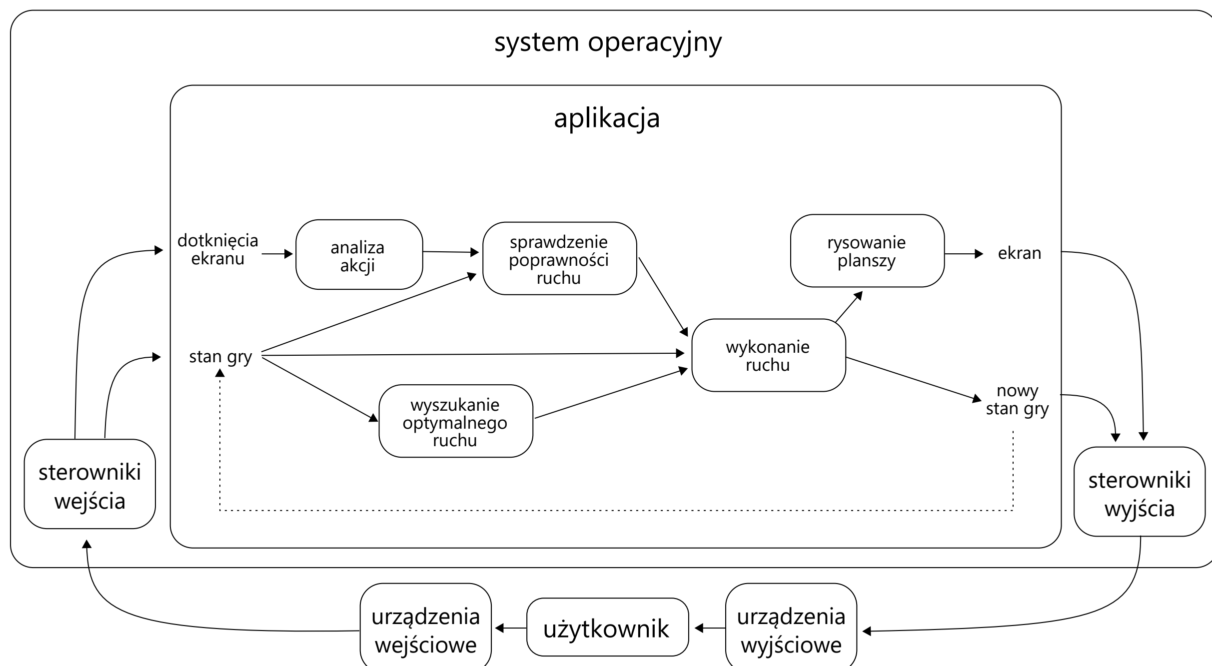
Ze względów praktycznych nie wszystkie operatory w aplikacji mogą pozostać bezstanowe. Często stosowanym operatorem, który wymaga przechowywania wewnętrznego stanu (pamięci), jest operator akumulacji (zwany również operatorem skanowania):

$$f_{scan}(\langle(t_0,v_0),(t_i,v_i),(t_{i+1},v_{i+1}),\ldots\rangle) = \langle(t'_i,w_i) : w_o = g(v_0,0) \wedge w_i = g(v_i, g(v_{i-1}))\rangle \quad (10)$$

Interakcja pomiędzy aplikacją i użytkownikiem jest (ang. *human-computer interaction, HCI*) to dwustronny proces, w którym obie strony produkują i konsumują informacje. Jest to proces cykliczny, nie ma natomiast ściśle określonej kolejności akcji, tzn. użytkownik może wykonać wiele interakcji bez oczekiwania na odpowiedź, tak samo komputer może produkować dane wyjściowe bez interakcji użytkownika.

W praktyce pomiędzy użytkownikiem i komputerem znajdują się urządzenia wejścia/wyjścia, przez które następuje interakcja. Komputer może być zamodelowany jako system operacyjny, w skład którego wchodzą sterowniki wejścia i wyjścia, oraz aplikację interaktywną działającą w ramach tego systemu. Aplikacja nie prowadzi interakcji bezpośrednio z użytkownikiem, a raczej z warstwą abstrakcji zapewnianą przez system operacyjny. Zatem z punktu widzenia aplikacji można powyższy schemat uprościć jedynie do wymiany informacji pomiędzy systemem i aplikacją. Ponieważ komunikacja odbywa się zwykle w sposób asynchroniczny, w postaci przerwań, zegarów lub zdarzeń, współdziałanie sytemu operacyjnego z aplikacją można zamodelować jako cykl przetwarzania strumieni zdarzeń.

Strumienie zdarzeń przekazywane są do aplikacji, która przetwarza je zgodnie z góry określonym acyklicznym grafem skierowanym, w którym wierzchołki oznaczają operatory, a krawędzie oznaczają strumienie pomiędzy nimi. Dzięki koncepcji strumieni zdarzeń oraz cyklu pomiędzy system i aplikacją zamodelowany został asynchroniczny i niedeterministyczny przepływ sterowania. Dla przykładu na rysunku 2 przedstawiony został model gry w szachy na urządzeniu mobilnym.



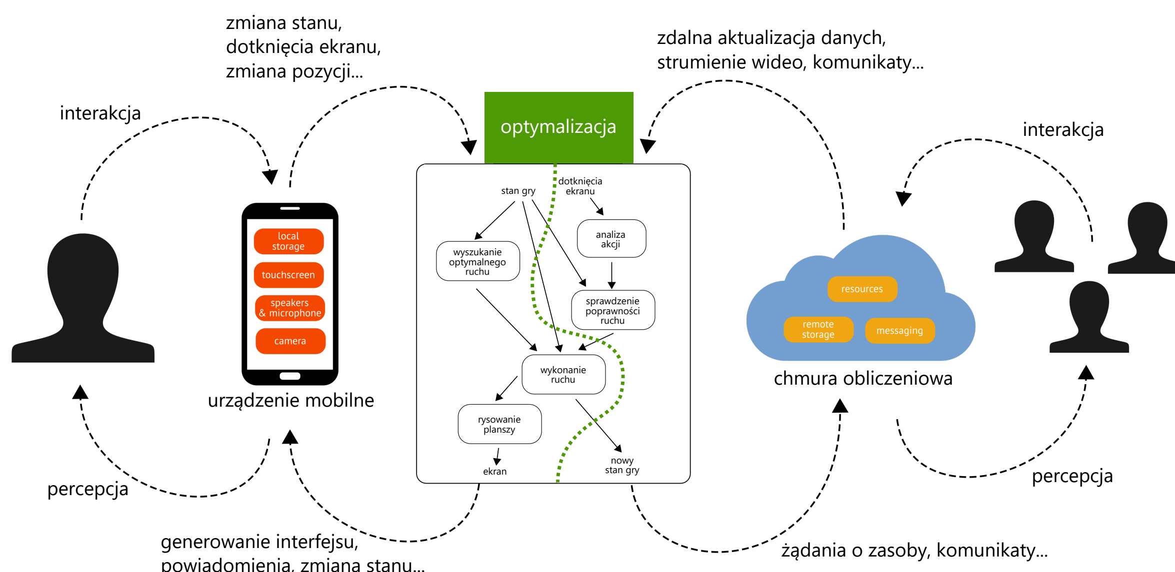
Rys. 2. Model aplikacji do gry w szachy

Aplikacja posiada dwa strumienie wejściowe: dotknięcia ekranu oraz stan gry.

Należy zauważyć, że aplikacja nie posiada żadnego globalnego stanu gry, jest on natomiast przekazywany do aplikacji i z aplikacji w postaci cyklicznego strumienia (oznaczonego przerywaną linią). Dotknięcia ekranu zostają mapowane na akcje, tzn. ruchy figur na szachownicy. Ze strumienia ruchów wybrane zostają tylko dozwolone ruchy i po scaleniu z aktualnym stanem gry generują nowy stan. Co drugi stan gry wykonany zostaje ruch przeciwnika, tzn. komputera. W tym celu, na podstawie strumienia stanu wyszukany zostaje optymalny ruch, który przekazywany jest w postaci strumienia do omawianego już operatora wykonania ruchu.

3. Opis problemu i przestrzeń rozwiązań

Optymalizacja aplikacji mobilnej w modelu opisanym w rozdziale 2. polega na rozdziale operatorów na wykonywane na urządzeniu i w chmurze obliczeniowej. Koncepcja rozdziału aplikacji została przedstawiona na rysunku 3.



Rys. 3. Koncepcja optymalizacji

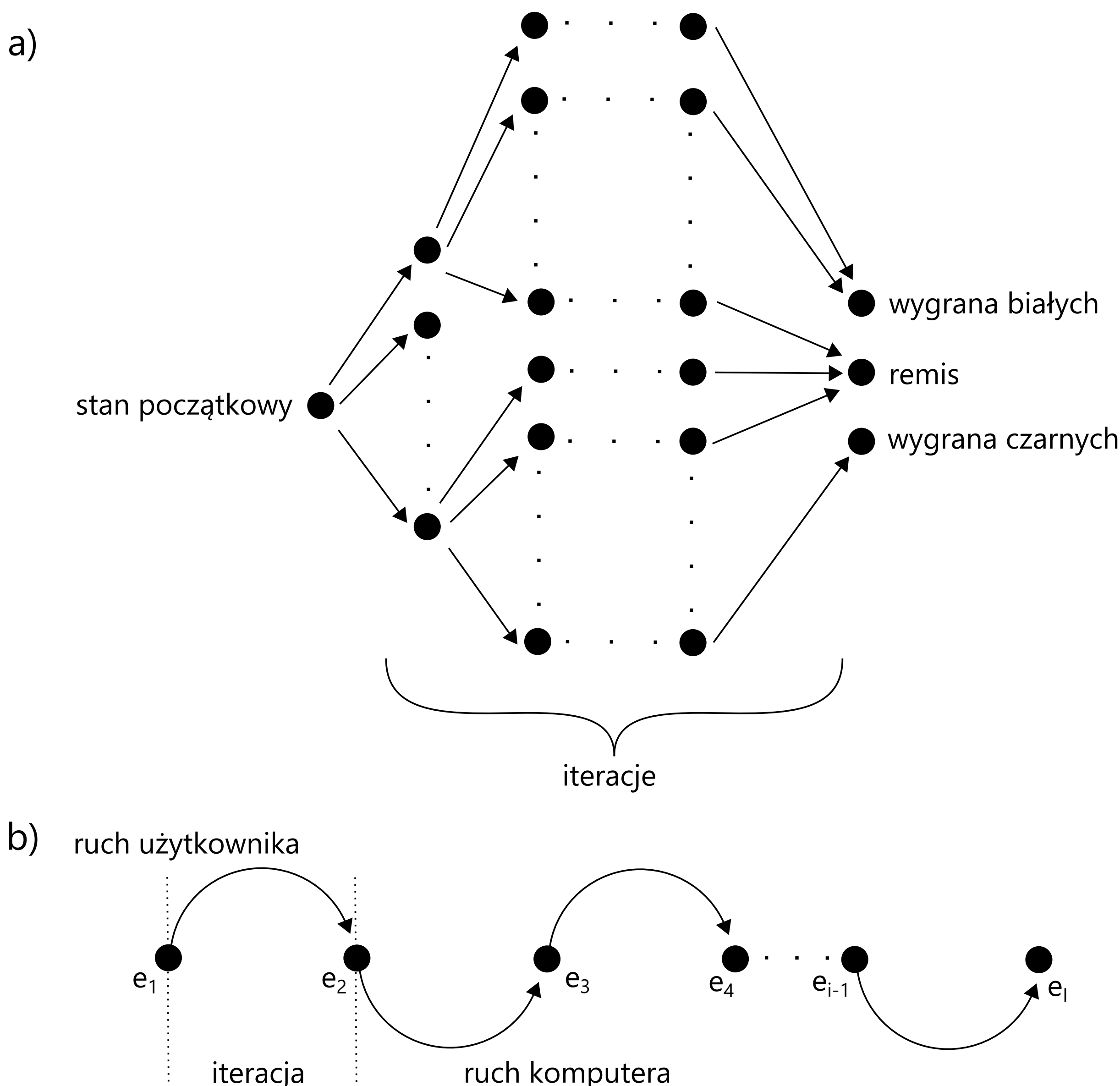
Celem optymalizacji jest minimalizacja kosztu c wykonania aplikacji przy czasie wykonania aplikacji poniżej założonego czasu maksymalnego t_{max}. Koszt wykonania aplikacji rozumiany jest jako suma kosztów wykonania wszystkich operatorów:

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

Koszt wykonania w ogólności jest proporcjonalny do czasu wykorzystania procesora, zsumowanego dla wszystkich rdzeni i może być interpretowany jako wykorzystana energia lub koszt pieniężny np. w przypadku usługi IaaS (ang. *Infrastructure as a Service*).

Niektóre operatory mogą wykonywać się równolegle, a więc czas wykonania aplikacji jest równy długości ścieżki krytycznej w grafie skierowanym, w którym wierzchołkami są operatory, krawędziami strumienie pomiędzy nimi, a wagami czasy wykonania operatorów.

Należy zauważyć, że w ogólności koszt i czas każdego operatora może być nieskończony, ponieważ liczba iteracji może być nieskończona. Nie można również w żaden sposób oszacować liczby iteracji, ponieważ jest ona zależna od niedeterministycznego zachowania użytkownika. W celu zoptymalizowania aplikacji z góry, należałoby rozważyć wszystkie możliwe przebiegi sterowania, a następnie po każdym kroku wybierać optymalny rozdział operatorów. Liczba możliwych przebiegów aplikacji rośnie wykładniczo względem liczby iteracji, dla przykładu w przypadku gry w szachy pierwszy ruch może zostać rozegrany na 20 sposobów, kolejny ruch znowu na 20 sposobów, co oznacza 400 możliwych przebiegów gry po zaledwie dwóch ruchach. Ostatecznie przebieg gry kończy się na jednym z trzech możliwych stanów: wygrana białych, wygrana czarnych lub remis. Rysunek 4 przedstawia ilustrację potencjalnych przebiegów gry.



Rys. 4. a) Przestrzeń rozwiązań b) Przykładowa ścieżka przebiegu gry

Liczbę wszystkich przebiegów P gry trwającej I iteracji można obliczyć na podstawie liczby możliwości m w każdym ruchu:

$$P = m_0 \cdot m_1 \cdot \ldots \cdot m_I \tag{12}$$

Średnia liczba ruchów w partii wynosi około 40 ruchów w przypadku rozgrywki zaawansowanych graczy [12]. W każdym ruchu odbywają się dwie iteracje, posunięcie gracza białego i posunięcie gracza czarnego, a więc łącznie gra ma około 80 iteracji. Średnia liczba możliwości w danym posunięciu jest różna w zależności od liczby figur na planszy, ale przez większość gry wynosi około 30 [19]. Można więc oszacować, że przeciętna liczba możliwych przebiegów gry w szachy wynosi:

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

Jak łatwo zauważyć przestrzeń potencjalnych przebiegów gry, a co za tym idzie rozwiązań optymalizacji, jest bardzo duża, a więc ten sposób optymalizacji wykonania aplikacji jest niepraktyczny. W związku z tym rozważać będziemy koszt i czas wykonania pojedynczej iteracji aplikacji, tzn. wykonania aplikacji dla wystąpienia zdarzenia w dowolnym ze strumieni wejściowych:

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

Mimo, że dla każdego zdarzenia koszt i czas każdego z operatorów może być inny, a nawet zerowy, średni koszt pojedynczej iteracji jest równy sumie uśrednionych kosztów wszystkich operatorów:

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

W związku z tym w celu minimalizacji kosztu wykonania aplikacji należy zminimalizować średni koszt wykonania każdego operatora. Wykonywanie operatorów w dwóch środowiskach oznacza, że do kosztów i czasu wykonania aplikacji należy doliczyć koszt i czas transferu danych pomiędzy urządzeniem i chmurą. Najprościej zamodelować koszt komunikacji za pomocą dodatkowego operatora transferu przy przejściu strumienia pomiędzy środowiskami.

W celu dokonania optymalnego rozdziału konieczne jest estymowanie metryk operatorów oraz strumieni. W przypadku operatów mierzone są czasy wykonania dla każdego zdarzenia, a następnie uśrednianie po wszystkich iteracjach, w których operator otrzymał jakiekolwiek zdarzenie. Dla operatorów mapowania obliczenie czasu wykonania jest proste, ponieważ wystarczy obliczyć różnicę czasów w zdarzeniu wejściowym t_i i wyjściowym t'_i:

$$dt_i = t'_i - t_i \tag{16}$$

Analogiczne podejście można zastosować w przypadku operatora akumulacji i scalenia, ponieważ na każde zdarzenie wejściowe generują odpowiadające mu zdarzenie wyjściowe. W przypadku operatora filtrowania obliczenie czasu wykonania jest trudniejsze, ponieważ zdarzenia wyjściowe są generowane tylko w przypadku zdarzeń wejściowych przechodzących przez funkcję filtrującą. Aby możliwe było obliczenie różnicy czasu dla każdego zdarzenia, należy przekształcić operator filtrujący w taki sposób żeby generował zdarzenia puste:

$$f_{filter}(\langle (t_i, v_i), (t_{i+1}, v_{i+1}), \ldots \rangle) = \langle (t'_i, v_i) : g(v_i) > 0 \vee (t'_i, \oslash) : g(v_i) \leqslant 0 \rangle \tag{17}$$

Zdarzenia puste powinny być ignorowane na wejściu wszystkich operatorów.

Średni czas wykonania operatora wykorzystany jest do estymowania długości ścieżki krytycznej, będącej warunkiem optymalizacji kosztu. Jest również podstawą do estymowania średniego kosztu wykonania operatora, po przemnożeniu przez odpowiedni współczynnik kosztu λ. Współczynnik ten może być interpretowany w zależności od celu optymalizacji, np. do zoptymalizowania zużycia energii urządzenia mobilnego współczynnik kosztu wyznaczony będzie na podstawie energii pobieranej przez procesor urządzenia.

Koszt i czas transferu strumienia zdarzeń pomiędzy urządzeniem mobilnym i chmurą obliczeniową jest bezpośrednio zależny od przepustowości strumienia, tzn. rozmiaru wszystkich przesłanych zdarzeń w strumieniu uśrednionych po każdej iteracji. Biorąc pod uwagę, że transfer danych jest znaczący tylko w przypadku przejścia strumienia pomiędzy środowiskami urządzenia i chmury to można estymować jego czas i koszt jako czas i koszt wykonania odpowiadającego mu operatora transferu, upraszczając tym samym problemem optymalizacji jedynie do minimalizacji kosztów wykonania operatorów przy założonym czasie wykonania iteracji t_{max}.

Optymalizacja aplikacji w opisanym modelu polega na takim podziale grafu operatorów na urządzenie i chmurę obliczeniową aby suma kosztów (wag wierzchołków) była możliwie najniższa przy założonym czasie wykonania, czyli ścieżce krytycznej nie dłuższej niż t_{max}. W zależności od przyjętego celu optymalizacji, współczynnik kosztu λ może być różny dla chmury i urządzenia, a wykonanie operatora w jednym lub drugim środowisku może mieć niższy lub wyższy koszt.

Dla przykładu, gdy celem optymalizacji jest minimalizacja wykorzystania energii urządzenia mobilnego, bez względu na koszt wykorzystania chmury, współczynnik dla operatorów wykonywanych na urządzeniu będzie wynosił $\lambda > 0$, natomiast dla operatorów wykonywanych w chmurze będzie wynosił $\lambda = 0$. Przeniesienie wszystkich operatorów do chmury może jednak okazać się nieoptymalne ze względu na wysoki czas transferu wszystkich strumieni zdarzeń. W ogólności w przypadku optymalizacji energii urządzenia czas transferu w ramach każdej iteracji będzie odwrotnie proporcjonalny do kosztu wykonania iteracji.

Innymi słowy, przenosząc wykonanie do chmury obliczeniowej zmniejszamy koszt wykonania, ale wprowadzamy dodatkowy czas związany z transferem danych. Czas ten związany jest z wprowadzeniem operatora transferu lub dwóch operatorów transferu, jeśli strumień musi zostać przesłany z powrotem do urządzenia. Ogólny czas wykonania iteracji nie musi jednak wcale wzrosnąć, jeśli czas wykonania wprowadzonych operatorów jest mniejszy niż czas wykonania operatora przeniesionego.

Sytuacja gdy wszystkie obliczenia, czyli cała aplikacja, uruchomiona jest na chmurze, a urządzenie jedynie formułuje żądania i otrzymuje odpowiedzi jest jednym ze skrajnych scenariuszy optymalizacji. Taki scenariusz może być najlepszym rozwiązaniem optymalny, kiedy dane wejściowe i wyjściowe są niewielkie. W innym skrajnym przypadku, tzn. gdy mamy do czynienia z dużymi danymi wejściowymi lub wyjściowymi, może się okazać, że przenoszenie jakichkolwiek obliczeń do chmury jest nieopłacalne ze względu na czasy transferu.

W przypadku nieczystych operatorów, tzn. takich które posiadają wewnętrzny stan, przeniesienie wykonania pomiędzy dwoma środowiskami wiąże się również z migracją stanu. Dlatego konieczne jest estymowanie czasu i kosztu transferu stanu, który

zależy bezpośrednio od rozmiaru zawartych w nim danych. Widać tu pewną analogię do kosztów transferu strumienia zdarzeń, należy jednak zauważyć, że migracja stanu odbywać się będzie jednorazowo przy przeniesieniu, a co za tym idzie całkowity koszt migracji nie powinien być brany pod uwagę przy każdej iteracji, a raczej podzielony przez liczbę iteracji w ciągu całego cyklu życia aplikacji. Dokładne obliczenie liczby iteracji jest oczywiście niemożliwe w aplikacji interaktywnej, dlatego należy się posłużyć przewidywaną liczbą, uzyskaną np. przy poprzednim wykonaniu.

4. Statyczny algorytm optymalizacji

Podział aplikacji interaktywnej na dwie części sprowadza problem do podziału grafu (ang. *graph partitioning*), które optymalne rozwiązanie otrzymuje się przez odpowiednie wymianę wierzchołków (operatorów) i jest to problem o złożoności wielomianowej [7]. Jednak uwzględnienie optymalizacji wielokryterialnej (czas wykonania i koszt) wymaga podejścia heurystycznego.

Do optymalizacji grafu operatorów można zastosować np. algorytm genetyczny, analogicznie jak w przypadku aplikacji statycznych [10]. W tym celu dla danego przebiegu aplikacji, należy skonstruować graf składający się z połączonych ze sobą grafów odpowiadających poszczególnym iteracjom. Następnie należy podzielić zbiór operatorów F na dwa podzbiory: operatory uruchamiane w chmurze F_c i na urządzeniu F_m. Dodatkowo należy rozważyć zbiór krawędzi E_t pomiędzy podzbiorami F_c i F_m, tzn. transferu danych pomiędzy operatorami w różnych środowiskach. Funkcje c_c, c_m i c_t oznaczają odpowiednio koszty przetwarzania operatora w chmurze, na urządzeniu oraz koszt transferu danych.

$$F_c \subset F, F_m = F \setminus F_c \tag{18}$$

$$E_{cm} = \{(f_c, f_m) : (f_c, f_m) \in E \wedge f_c \in F_c \wedge f_m \in F_m\} \tag{19}$$

$$E_{mc} = \{(f_c, f_m) : (f_c, f_m) \in E \wedge f_c \in F_c \wedge f_m \in F_m\} \tag{20}$$

$$E_t = E_{cm} \cup E_{mc} \tag{21}$$

$$C = \sum_{f \in F_c} c_c(f) + \sum_{f \in F_m} c_m(f) + \sum_{e \in E_t} c_t(e) \tag{22}$$

Graf będący przedmiotem optymalizacji będzie posiadał $N * I$ wierzchołków, gdzie N to liczba operatorów w każdej iteracji, I to liczba iteracji. Optymalizacja tak dużego grafu wiązałaby się dużym narzutem obliczeniowym, który prawdopodobnie przewyższałby oszczędności wynikające z optymalizacji. Co więcej, taką optymalizację należałoby przeprowadzić dla każdego potencjalnego przebiegu aplikacji. Kolejnym problemem jest duża niepewność w estymacji kosztu i czasu wykonania operatorów, zależna od zachowania użytkownika. Po każdej zmianie wag wierzchołków konieczne byłoby przeprowadzanie ponownej optymalizacji, co teoretycznie może mieć miejsce w każdej iteracji.

5. Iteracyjny algorytm heurystyczny

Iteracyjny charakter modelu aplikacji interaktywnych pozwala na zastosowanie innego podejścia, to znaczy algorytmu zachłannego, który w każdej iteracji znajdzie

podział lokalnie optymalny przez przeniesienie maksymalnie jednego operatora pomiędzy środowiskami. Narzut algorytmu będzie bardzo niski, ponieważ sprowadza się on do znalezienia operatora ($operator_{max}$) o największej różnicy kosztów (dc_{max}) pomiędzy urządzeniem i chmurą obliczeniową przy spełnieniu warunku maksymalnego czasu przetwarzania (dt_{max}), a takie przeszukanie grafu operatorów ma złożoność liniową.

```
function OPTIMIZE(operators)
    dc_max ← 0
    operator_max ← null
    for all operators do
        if operator in cloud then
            dt ← mobileTime(operator) − cloudTime(operator)
            if t + dt ⩽ t_max then
                dc ← cloudCost(operator) − mobileCost(operator)
                if dc > dc_max then
                    dc_max ← dc
                    operator_max ← operator
                end if
            end if
        else
            dt ← cloudTime(operator) − mobileTime(operator)
            if t + dt ⩽ t_max then
                dc ← mobileCost(operator) − cloudCost(operator)
                if dc > dc_max then
                    dc_max ← dc
                    operator_max ← operator
                end if
            end if
        end if
    end for
    return max_operator
end function
```

Na powyższym listingu został przedstawiony pseudokod algorytmu zwracającego identyfikator operatora, który powinien zostać przeniesiony, tzn. jeśli działa na urządzeniu to w następnej iteracji powinien działać w chmurze i na odwrót. Funkcje $mobileTime()$, $mobileCost()$, $cloudTime()$ i $cloudCost()$ zwracają wartości kosztów i czasów przetwarzania dla danego operatora na podstawie średnich uzyskanych z poprzednich wywołań operatora. Można przyjąć, że w pierwszej iteracji wszystkie czasy i koszty wynoszą 0. Jak wynika z pseudokodu algorytmu ma on złożoność liniową zależną od liczby operatorów.

W celu przebadania zachowania się algorytmu opracowano narzędzie symulujące współpracę urządzenia i chmury obliczeniowej. Wykorzystano biblioteki otwartego oprogramowania RxJS [21] oraz CycleJS [20], przeznaczone do implementacji aplikacji interaktywnych. Tabela 2 przedstawia średnie czasy (w milisekundach) przetwarzania operatorów na urządzeniu mobilnym i rozmiary komunikatów zdarzeń wejściowych i wyjściowych (w bajtach) dla przykładowej aplikacji gry w szachy przedstawionej na

Tabela 2

Dane dla badanych przebiegów gry.

Operator	**Średni czas (ms)**	**Średni rozmiar zdarzeń wejściowych (b)**	**Średni rozmiar zdarzeń wyjściowych (b)**
Analiza akcji	0,91	27,86	23
Sprawdzenie ruchu	0,45	581,86	8
Wyszukanie opt. ruchu	635,27	555,86	23
Wykonanie ruchu	0,46	582,25	551,45
Rysowanie planszy	0,87	556,26	4506,21

rysunku 2, dane zostały uśrednione z wszystkich badanych przebiegów gry. Można zauważyć, że operator wyszukania optymalnego ruchu ma największy wpływ na czas i koszt przetwarzania całej iteracji. Zakładając, że czas przetwarzania tego operatora w chmurze obliczeniowej będzie niższy niż na urządzeniu, jest to najlepszy kandydat do przeniesienia na chmurę. Duży rozmiar zdarzeń wejściowych i wyjściowych dla operatora rysowania planszy jest przesłanką, że przeniesienie całej aplikacji do środowiska chmury obliczeniowej nie jest optymalnym rozwiązaniem.

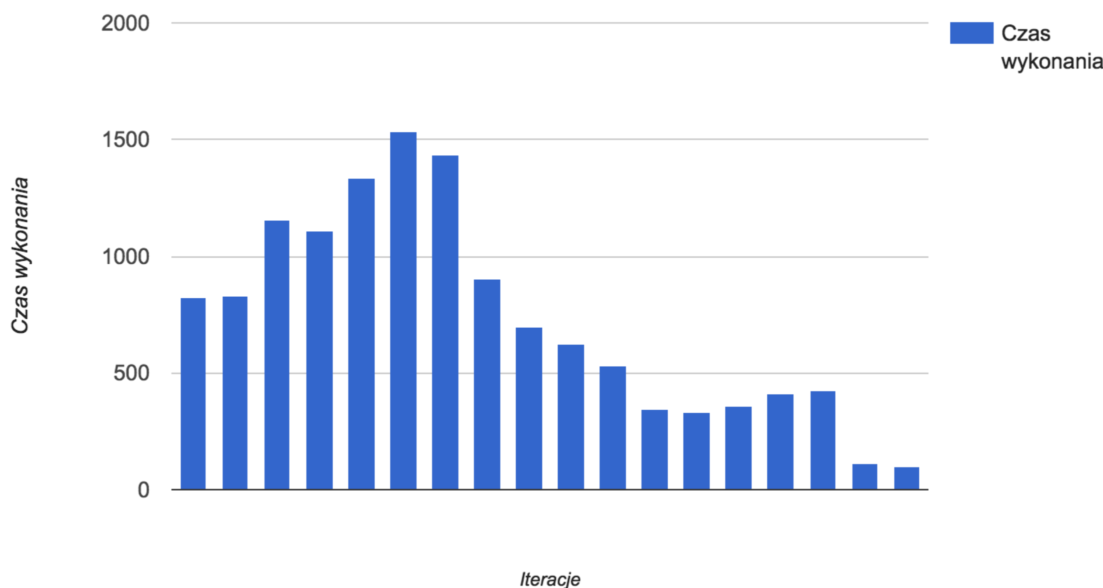
Konieczne jest również przeanalizowanie czasu wykonania operatora w czasie, tzn. w kolejnych iteracjach. Czasy przetwarzania zostały przedstawione na wykresie 5 i można zauważyć, że są one zmienne. W początkowej fazie gry wyszukanie optymalnego ruchu przebiega szybko, prawdopodobnie ze względu na mniejszą liczbę możliwych otwarć gry. W środkowej fazie czas przetwarzania jest najdłuższy ze względu na największą liczbę potencjalnych ruchów, które muszą zostać przeanalizowane. W końcowej fazie liczba figur, a więc i możliwych ruchów jest mniejsza, co przekłada się na bardzo szybkie działanie operatora.

Jeśli założymy, że t_{max} równe 1s jest akceptowalne dla użytkownika, to przeniesienie operatora do chmury jest konieczne tylko w środkowej fazie gry. Należy jednak mieć na uwadze przypadek kiedy czas przetwarzania operatora w kolejnych iteracjach waha się powyżej i poniżej t_{max}, co może prowadzić do częstego przenoszenia przetwarzania pomiędzy środowiskami. Jednym z rozwiązań tego problemu może być analiza kilku iteracji w przód. To potwierdza zasadność wymiany pojedynczego operatora pomiędzy urządzeniem a chmurą.

6. Wnioski

Wielokryterialna optymalizacja przetwarzania aplikacji mobilnej przez podział na urządzenie mobilne i chmurę obliczeniową ma trzy możliwe rozwiązania: przetwarzanie całej aplikacji w chmurze, przetwarzanie całej aplikacji na urządzeniu lub podział jej komponentów (operatorów) na te dwa środowiska. Jak pokazano na przykładzie aplikacji do gry w szachy, przeniesienie całego przetwarzania do chmury może nie być optymalne ze względu na wysoki koszt transmisji danych. Z kolei przy uruchomieniu aplikacji tylko na urządzeniu pojedyncza iteracja może trwać zbyt długo dla użytkownika. Zaproponowano więc heurystyczny algorytm rozlokowania operatorów.

W rozważaniach przyjęto, że urządzenie i chmura są tylko do dyspozycji wy-



Rys. 5. Zmiana czasu przetwarzania operatora

konywanej aplikacji. W rzeczywistości może się ich wykonywać wiele, co ma istotny wpływ na czasy przetwarzania i transmisji danych. W związku z tym dalszy rozwój metody powinien uwzględniać takie sytuacje.

LITERATURA

1. Bauer H., Goh Y., Schlink S., Thomas C.: The supercomputer in your pocket. MkKinsey on Semiconductors (Autumn), 14–27, 2012.
2. Chun B., Ihm, S., Maniatis P., Naik M., Patti A.: Clonecloud: elastic execution between mobile device and cloud. Proceedings of the sixth conference on Computer systems, 301–314, 2011.
3. Cisco, Cisco Visual Networking Index: Global mobile data traffic forecast update, 2013–2018. 2014.
4. Cuervo E., Balasubramanian A., Cho D., Wolman A., Saroiu S., Chandra R., Bahl P.: MAUI: making smartphones last longer with code offload. Proceedings of the 8th international conference on Mobile systems, applications, and services, 49–62, 2010.
5. Czaplicki E.: Elm: Concurrent FRP for Functional GUIs. Senior thesis, Harvard University, 2012.
6. Etzion O., Niblett P.: Event processing in action. Manning Publications Co., 2010.
7. Hao, J.X., Orlin J.B.: A faster algorithm for finding the minimum cut in a directed graph. Journal of Algorithms 17.3, 424–446, 1994.

8. Kemp R., Palmer N., Kielmann T., Bal H.: Cuckoo: a computation offloading framework for smartphones. Mobile Computing, Applications, and Services, 59–79, Springer, 2010.
9. Kosta S., Aucinas A., Hui P., Mortier R., Zhang X.: Unleashing the power of mobile cloud computing using thinkair. 2011.
10. Krawczyk H., Nykiel M., Proficz J.: Mobile Offloading Framework: Solution for Optimizing Mobile Applications Using Cloud Computing. Computer Networks, 293–305, Springer International Publishing, 2015.
11. Liberty J., Betts P., Turalski S.: Programming Reactive Extensions and LINQ. Springer, 2011.
12. Chessgames Services LLC: Chess Statistics. http://www.chessgames.com/chessstats.html, 2016.
13. Ma R.KK., Lam K.T., Wang C.: excloud: Transparent runtime support for scaling mobile applications in cloud. Cloud and Service Computing (CSC), 2011 International Conference on, 103–110, 2011.
14. March V., Gu Y., Leonardi E., Goh G., Kirchberg M., Lee B.S.: μcloud: towards a new paradigm of rich mobile applications. Procedia Computer Science, volume 5, 618–624, 2011.
15. Nilsson H., Courtney A., Peterson J.: Functional reactive programming, continued. Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, 51–64, 2002.
16. Pathak A., Hu C., Zhang M., Bahl P., Wang Y.: Enabling automatic offloading of resource-intensive smartphone applications. 2011.
17. Satyanarayanan M.: Mobile computing: the next decade. Proceedings of the 1st ACM workshop on mobile cloud computing & services: social networks and beyond, 2010.
18. Satyanarayanan M., Bahl P., Caceres R., Davies N.: The case for vm-based cloudlets in mobile computing. Pervasive Computing, IEEE, volume 8, number 4, 14–23, 2009.
19. Shannon C.: Programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, volume 41, number 314, 256–275, 1950.
20. Staltz A.: Cycle.js - a functional and reactive JavaScript framework for cleaner code. http://cycle.js.org/, 2016.
21. Microsoft Open Technologies: The Reactive Extensions for JavaScript. https://github.com/Reactive-Extensions/RxJS, 2016.
22. Zhang X., Kunjithapatham A., Jeong S., Gibbs S.: Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. Mobile Networks and Applications, volume 16, number 3, 270–284, 2011.