# Optymalizacja kosztu działania aplikacji na urządzeniach mobilnych wspomaganych przez chmurę obliczeniową

mgr inż. Michał Nykiel

Promotor:
prof. dr hab. inż. Henryk Krawczyk

Promotor pomocniczy:
dr inż. Jerzy Proficz


Wydział Elektroniki,
Telekomunikacji i Informatyki

7 czerwca 2017

# Spis treści

# Wykaz oznaczeń

$A$ Model aplikacji w postaci acyklicznego grafu skierowanego. 40

$A_K$ Model kosztu przetwarzania aplikacji. 44

$B$ Przepustowość sieci. 44

$D$ Model aplikacji iteracyjnej w postaci acyklicznego grafu skierowanego. 63

$D_K$ Model kosztu przetwarzania aplikacji iteracyjnej. 69

$E$ Zbiór krawędzi oznaczających połączenia pomiędzy komponentami. 40

$E_{ext}$ Zbiór połączeń zewnętrznych (jeden z komponentów na chmurze drugi na urządzeniu). 42

$E_{int}$ Zbiór połączeń wewnętrznych (oba komponenty na chmurze lub urządzeniu). 41

$F$ Zbiór wszystkich operatorów w aplikacji. 63

$K$ Całkowity koszt wykonania aplikacji. 45, 46, 49

$K_c$ Koszt częściowy komponentów uruchomionych w chmurze. 46, 47, 49, 70

$K_e$ Koszt częściowy transmisji danych. 46, 47

$K_m$ Koszt częściowy komponentów uruchomionych na urządzeniu mobilnym. 46, 47, 49, 70

$N$ Zbiór wierzchołków oznaczających komponenty aplikacji. 40

$N_c$ Zbiór komponentów uruchomionych w chmurze obliczeniowej. 41

$N_m$ Zbiór komponentów uruchomionych na urządzeniu mobilnym. 41

$P_c$ Współczynnik wykorzystania energii (mocy) procesora w chmurze. 43, 45, 68

$P_e$ Współczynnik wykorzystania energii (mocy) podzespołów transmisji danych. 44, 45, 68

$P_m$ Współczynnik wykorzystania energii (mocy) procesora na urządzeniu mobilnym. 43, 45, 68

$R$ Ciąg rozdziałów operatorów aplikacji iteracyjnej. 65

$S$ Zbiór krawędzi oznaczających strumienie zdarzeń. 63

$V$ Zbiór wszystkich zdarzeń we wszystkich strumieniach. 63

$c_{max}$ Maksymalny dopuszczalny koszt wykonania w chmurze obliczeniowej. 47, 70

$f(s)$ Operator przetwarzania strumienia. 60

$h$ Liczba operatorów w modelu aplikacji. 72

$k_c(n)$ Koszt wykonania w chmurze obliczeniowej. 45

$k_e(e)$ Koszt transferu pomiędzy urządzeniem i chmurą. 45

$k_g(f)$ Koszt migracji operatora $f$. 69

$k_m(n)$ Koszt wykonania na urządzeniu mobilnym. 44

$m_{max}$ Maksymalny dopuszczalny koszt wykonania na urządzeniu mobilnym. 47, 70

$r_i$ Rozdział operatorów aplikacji w iteracji $i$. 65

$s_{in}$ Strumień zdarzeń wejściowych. 60

$s_{in}(n)$ Rozmiar danych wejściowych. 42

$s_{out}$ Strumień zdarzeń wyjściowych. 60

$s_{out}(n)$ Rozmiar danych wyjściowych. 42

$t_c(n)$ Czas wykonania w chmurze obliczeniowej. 42

$t_m(n)$ Czas wykonania na urządzeniu mobilnym. 42

$z(n_i, n_j)$ Rozmiar danych przesyłanych pomiędzy komponentami $n_i$ i $n_j$. 43

# Wykaz skrótów

**API** ang. application programming interface. 19, 25

**AR** ang. augmented reality. 32

**ARM** ang. Advanced RISC Machine. 18

**ART** ang. Android Runtime. 20

**BLE** ang. Bluetooth low energy. 16

**CaaS** ang. Communication as a Service. 27

**FRP** ang. Functional reactive programming. 58, 59, 87

**GFLOPS** ang. giga FLoating point Operations Per Second. 13

**GPS** ang. Global Positioning System. 14, 32

**GSM** ang. Global System for Mobile Communications. 15

**HAL** ang. hardware abstraction layer. 19

**IaaS** ang. Infrastructure as a Service. 26

**IoT** ang. Internet of Things. 8, 12, 15–17, 30

**IPC** ang. inter-process communication. 20

**KB** kilobajt, $2^{10}$ bajtów. 12

**Kbps** kilobit na sekundę. 13

**LCD** ang. liquid crystal display. 12

**MaaS** ang. Monitoring as a Service. 27

**MB** megabajt. 13

**Mbps** megabit na sekundę. 13

**MCC** ang. Mobile Cloud Computing. 29, 31

**MHz** megaherc. 12, 13

**MOFF** ang. Mobile Offloading Framework. 85

**mWh** miliwatogodzina. 14

**NFC** ang. Near Field Communication. 14, 16

**P2P** ang. Peer to Peer. 31

**PaaS** ang. Platform as a Service. 26, 27

**PDA** ang. Personal Digital Assistant. 12, 13

**RAM** ang. random-access memory. 12

**REST** ang. Representational State Transfer. 29

**RISC** ang. reduced instruction set computing. 18

**ROM** ang. read-only memory. 12

**SaaS** ang. Software as a Service. 27

**SDK** ang. Software Development Kit. 34

**SOAP** ang. Simple Object Access Protocol. 29

**SoC** ang. System-on-a-chip. 18

**SSH** ang. secure shell. 23

**USB** ang. Universal Serial Bus. 18

**VPS** ang. Virtual Private Server. 23

# Rozdział 1

# Wprowadzenie

Rozwój usług IT oraz Internetu Rzeczy (IoT (*ang. Internet of Things*)) doprowadził do pojawienia się nowej koncepcji wykorzystania zasobów i systemów informatycznych w różnego typu organizacjach. Zamiast gromadzenia i utrzymywania infrastruktury informatycznej wewnątrz organizacji wykreowano dogodne warunku wydzierżawiania tego typu zasobów u firm specjalistycznych zajmujących się ich utrzymaniem. Takie podejście jest mniej kosztowne dla danej organizacji, gdyż koszty utrzymania i modernizacji infrastruktury informatycznej rozkładają się na większą liczbę firm zainteresowanych takimi usługami. Co więcej dana organizacja może skoncentrować się na swojej konkretnej działalności pomijając kosztowne problemy dotyczące często obcej jej tematyki. Takie rozwiązanie dotyczy tzw. chmury obliczeniowej, której wykorzystanie staje się coraz popularniejsze.

Zasoby informatyczne mogą być wydzierżawiane na trzech poziomach: infrastruktury (IaaS), platformy (PaaS) oraz oprogramowania (SaaS). Prezentowana praca doktorska dotyczy najwyższej warstwy, a szczególnym przedmiotem zainteresowania są aplikacje interaktywne, tzn. otwarte na bezpośrednią interakcję z użytkownikiem posługującym się urządzeniem mobilnym, takich jak tablet czy smartfon. Współczesne aplikacje charakteryzują się również architekturą warstwową, gdzie interakcje z użytkownikiem wspomaga odpowiedni interfejs aplikacji (GUI). Funkcje realizowane przez aplikację stanowią jej logikę biznesową, zaś dostęp do danych zapewniają odpowiednie systemy baz danych. Przy współpracy chmury obliczeniowej z konkretnym użytkownikiem pojawia się istotny problem alokacji komponentów aplikacji na urządzenie mobilne oraz chmurę obliczeniową. Naturalnym podejściem jest ulokowanie interfejsu aplikacji na urządzeniu mobilnym, zaś logiki biznesowej i danych w chmurze obliczeniowej. Inne skrajne przypadki alokacji dotyczą umieszczenia

całej aplikacji w chmurze, bądź całej aplikacji na urządzeniu mobilnym. Zakładając, że obliczenia realizowane w chmurze są znaczenie szybsze niż na urządzeniu mobilnym oraz czasy komunikacji i przesyłu danych między urządzeniem mobilnym i chmurą obliczeniową mogą przewyższyć czas wykonania obliczeń niektórych z komponentów logiki biznesowej, powstaje problem optymalizacji alokacji komponentów aplikacji na urządzeniu mobilnym i chmurze obliczeniowej. Jest to główne zadanie rozpatrywane w niniejszej rozprawie doktorskiej.

Rozwiązanie tego problemu wymagało wprowadzenia modelu aplikacji interaktywnych opartego na paradygmacie programowania funkcyjno-reaktywnego. Podstawą takiego podejścia jest reagowanie na pojawiające się zdarzenia. Z kolei architekturę komponentową aplikacji przedstawiono za pomocą grafu ważonego, w którym wierzchołki reprezentują obliczenia, zaś krawędzie wymianę danych pomiędzy komponentami. Rozpatrzono dwa główne typy modeli aplikacji - statyczny, w którym wagi są stałe w czasie realizowanych obliczeń oraz model dynamiczny, gdzie wagi wierzchołków i krawędzi mogą się zmieniać w każdej iteracji obliczeń. Przypadek aplikacji statycznej odpowiada więc jednej iteracji dla aplikacji dynamicznej. Każda iteracja aplikacji interaktywnej odpowiada obliczeniom typu request-response, w których zakłada się stabilność parametrów dotyczących wykonywanych obliczeń.

Rozpatrzmy aplikacje monitorowania zadanego obszaru geograficznego (np. parkingu) oraz interaktywną grę w szachy pomiędzy graczem i systemem komputerowym. W przypadku pierwszym algorytm monitorowania pozostaje stały, okresowo dokonuje się odczytu z kamer i na tej podstawie określa się sytuację na parkingu. W przypadku drugim po każdy ruchu gracza komputer analizuje stan szachownicy i wyznacza najlepsze posunięcie. W pierwszym i drugim przypadku przyjmujemy stałą wielkość danych (stan parkingu, stan szachownicy). Zmieniać się może czas wykonania aplikacji w zależności od sytuacji na parkingu czy stanu na szachownicy, bowiem ich analiza może wymagać wykorzystania różnych algorytmów lub przeszukania większej przestrzeni rozwiązań. Czasy transmisji są stałe w obu przypadkach, gdyż dotyczą przekazania opisu aktualnego stanu (obraz parkingu, stan szachownicy). Jednak przy rozdziale obliczeń na komputer monitorujący czy urządzenie mobilne i chmurę obliczeniową będą zmieniały się również czasy transmisji, gdyż przy analizie złożonych sytuacji będzie angażowana chmura lub zewnętrzny komputer monitorujący i tylko wówczas będzie konieczne przekazywanie stanu. Tak więc obie aplikacje są przykładami modelu dynamicznego. Tego typu aplikacje stają się coraz bardziej popularne, a ich liczba oraz praktyczne znaczenie będzie wzrastać

wraz z coraz szerszym wykorzystaniem technologii mobilnych oraz IoT w nowych obszarach zastosowań.

Problem optymalizacji alokacji został sformułowany jako podział grafu reprezentującego aplikację na dwie części, tak aby sumaryczny koszt obliczeń i komunikacji był minimalny, przy dodatkowych ograniczeniach na koszty (np. zużycie energii) każdej z części. Z uwagi na złożoność kryterium optymalizacji rozpatrywany problem charakteryzuje się złożonością niewielomianową (NP). Wykazanie tego stwierdzenia stanowi potwierdzenie pierwszej tezy rozprawy doktorskiej.

W dalszych rozważaniach przyjęto, że realizacja funkcji aplikacji sprowadza się do wywołania odpowiednich usług IT (serwisów internetowych), które są dostępne w środowisku urządzenia mobilnego i chmury obliczeniowej. Wykonaniem aplikacji zarządza odpowiedni moduł – biblioteka obsługi strumieni zdarzeń, który może być częściowo alokowany na urządzeniu i chmurze obliczeniowej. Dla tak przyjętego rozproszonego modelu obliczeń zbudowano środowisko eksperymentalne do testowania opracowanych heurystycznych algorytmów alokacji aplikacji. Wykorzystano przy tym dostępną chmurę obliczeniową opartą na środowisku OpenStack oraz środowisko NodeJS i bibliotekę RxJS wspomagającą budowę aplikacji.

Dla interaktywnej aplikacji gry w szachy przeanalizowano różne warianty czasów obliczeń i czasów komunikacji oraz praktycznie potwierdzono drugą tezę rozprawy. Teza ta formułuje warunki określające zakres stosowalności opracowanych algorytmów alokacji aplikacji na urządzenie końcowe i chmurę obliczeniową. Opracowane środowisko badawcze jest na tyle uniwersalne, że może być wykorzystane do przetestowania dowolnego algorytmu iteracyjnego. Warto podkreślić, że dotychczas przeprowadzana analiza złożoności obliczeniowej była na ogół ograniczana do aplikacji statycznych. To oznacza, że algorytmy optymalizacji alokacji komponentów aplikacji zakładały stałość parametrów czasowych we wszystkich iteracjach.

W związku z potencjalnymi zastosowaniami opisywanego rozwiązania przy maksymalizacji wydajności oraz minimalizacji wykorzystania energii w urządzeniach bezprzewodowych o niskiej mocy obliczeniowej, przeprowadzone badania są istotne z punktu widzenia rozwoju Internetu Rzeczy (IoT), a jeszcze bardziej z punktu widzenia Internetu Wszystkiego (IoE).

W rozdziale 2 zaprezentowano przegląd problematyki współpracy urządzeń mobilnych z chmurą obliczeniową i sformułowano tezy rozprawy doktorskiej. Rozdział 3 poświęcono statycznemu modelowi współpracy, w którym parametry aplikacji nie ulegają zmianie podczas jej wykonania. Przy tym założeniu zaprezentowano efek-

tywny algorytm rozdziału komponentów aplikacji na urządzenie i chmurę obliczeniową. W rozdziale 4 zaprezentowano optymalny i suboptymalny (liniowy) algorytm rozdziału, który uwzględnia parametry aplikacji zmienne w każdej iteracji jej wykonania. W rozdziale 5 zaprezentowano środowisko testowania w celu przeprowadzenia badań eksperymentalnych. W badaniach wykazano możliwość zastosowania opracowanych algorytmów oraz frameworka zarządzającego w przykładowych aplikacjach - gra w szachy oraz aplikacja monitorująca parking. W uwagach końcowych podsumowano wyniki badań oraz zasygnalizowano dalsze kierunki badań.

# Rozdział 2

# Przegląd problematyki

Przedstawiono problematykę dotycząca urządzeń mobilnych, chmury obliczeniowej, IoT oraz współpracy tych technologii. Wyszczególnione i porównane zostały istniejące rozwiązania pozwalające na budowanie aplikacji mobilnych wykorzystujących chmurę obliczeniową, ze szczególnym uwzględnieniem sposobu optymalizacji takiej współpracy. Zaproponowano nowy kierunek modelowania aplikacji mobilnych oparty o paradygmat programowania funkcyjnego i strumienie danych. Zdefiniowano problemy do rozwiązania oraz sformułowano tezy rozprawy doktorskiej.

## 2.1 Urządzenia i aplikacje mobilne

Urządzenia mobilne to małe i przenośne urządzenia elektroniczne pozwalające na przetwarzanie danych, wykonywanie obliczeń oraz uruchamianie różnego typu aplikacji. Najczęściej są one dostosowane do trzymania w ręku, posiadają ekran dotykowy pozwalający na interakcję z urządzeniem, a czasami również klawiaturę. Większość urządzeń mobilnych posiada także bezprzewodowy dostęp do Internetu, za pomocą technologii WiFi lub sieci GSM [72].

### 2.1.1 Rozwój urządzeń mobilnych

Pierwsze urządzenia mobilne zostały wprowadzone na rynek w 1984 roku przez firmę Psion pod nazwą Organiser I. Były to urządzenia określane jazko PDA (*ang. Personal Digital Assistant*). Przypominały one rozbudowany kalkulator, posiadały 8-bitowy procesor Hitachi, taktowany z częstotliwością 0.9 MHz, 4KB pamięci ROM i 2KB pamięci RAM, oraz monochromatyczny wyświetlacz LCD pozwalający na wy-

świetlenie jednej linii tekstu. Użytkownicy mogli pisać proste programy w specjalnie zaprojektowanym dla tego urządzenia języku POPL [78].

W latach 90' na rynku pojawiło się wiele urządzeń typu PDA, m.in. Psion Series 3, Apple Newton i Palm Pilot. W 1996 roku Nokia zaprezentowała urządzenie 9000 Communicator, które było pierwszym połączeniem PDA i telefonu komórkowego i zapoczątkowało kategorię urządzeń określaną dzisiaj jako smartfon [13]. Było to jedno z pierwszych urządzeń mobilnych pozwalających na dostęp do Internetu umożliwiających przeglądanie stron internetowych i odbieranie wiadomości e-mail. Communicator posiadał procesor Intela w architekturze i386 taktowany częstotliwością 24 MHz oraz 8MB pamięci wewnętrznej. Pracował pod kontrolą systemu operacyjnego GEOS.

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 [8]. Dostęp do Internetu jest dzisiaj podstawową funkcją urządzeń mobilnych, a prędkość połączeń w sieciach GSM podwoiła się. W roku 2012 średnia prędkość pobierania danych wynosiła 526 Kbps, zaś w 2013 prawie 1,4 Mbps [18]. 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 zapotrzebowania użytkowników, którzy oczekują ciągłego dostępu do usług Internetowych, multimediów, sieci społecznościowych, itp.

Dominującym sposobem interakcji użytkownika z urządzeniem są obecnie ekrany dotykowe, które są w stanie rozpoznać nawet kilkanaście punktów dotyku, a także określić siłę nacisku [99]. Wyświetlacze posiadają aktualnie rozdzielczości na poziomie porównywalnym z monitorami komputerowymi przy kilkukrotnie mniejszej przekątnej, dzięki czemu pojedyncze piksele są praktycznie nierozróżnialne dla ludzkiego oka. Do renderowania obrazu w takiej rozdzielczości konieczne było wyposażenie urządzeń mobilnych w dedykowane układy graficzne, które często wydajnościowo zbliżają się już do swoich odpowiedników w laptopach czy wręcz komputerach stacjonarnych. Dobrym przykładem jest tablet Google Pixel C, wyposażony w układ NVIDIA Tegra X1 o wydajności 512 GFLOPS (*ang. giga FLoating point Operations Per Second*) [75].

Oprócz wzrostu wydajności podzespołów oraz sieci, dzisiejsze urządzenia mobilne zostały wzbogacone o wiele dodatkowych komponentów w celu rozszerzenia ich funkcjonalności. Większość urządzeń wyposażona jest w aparat i kamerę o wysokiej

rozdzielczości, moduł Bluetooth, GPS (*ang. Global Positioning System*), żyroskop, akcelerometr, kompas, moduł NFC (*ang. Near Field Communication*), coraz częściej spotykane są również czytniki linii papilarnych. Wszystkie te podzespoły sprawiają, że urządzenia mobilne posiadają ogromną ilość informacji o kontekście ich wykorzystania, np. gdzie znajduje się użytkownik, czy użytkownik jest w ruchu i jakie urządzenia znajdują się w pobliżu.

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 [85], co jest dużym wyzwaniem dla ich projektantów.

### 2.1.2 Klasyfikacja urządzeń mobilnych

Urządzenia mobilne można rozpatrywać w kontekście następujących charakterystyk [77]:

- rodzaj mobilności,
- fizyczny rozmiar,
- możliwości funkcjonalne urządzenia,
- zastosowania.

Większość urządzeń określanych jako mobilne to urządzenia przenośne, to znaczy są związane z mobilnym hostem, np. smartfon przenoszony przez człowieka. Istnieją również autonomiczne urządzenia mobilne, takie jak np. roboty i drony.

Popularna klasyfikacja urządzeń wg. rozmiaru opiera się na trzech klasach [103]:

- wielkości dłoni, np. smartfony, kamery, aparaty, zegarki,
- wielkości kartki papieru, np. tablety, laptopy,
- wielkości tablicy szkolnej, np. interaktywne tablice, telewizory.

Ze względu na możliwości urządzeń możemy wyróżnić następujące kategorie:

- przenośne komputery, np. laptopy, tablety, konsole,
- telefony i smartfony,

- urządzenia multimedialne, np. aparaty, kamery, odtwarzacze,
- urządzenia specjalizowane, IoT.

Przenośne komputery charakteryzują się największą mocą obliczeniową oraz możliwością uruchamiania różnego typu aplikacji. Posiadają też dość duże ekrany, zazwyczaj o przekątnej powyżej 10", a ich obsługa najczęściej odbywa się za pomocą pełnej klawiatury, myszki, gładzika, rysika lub innego fizycznego kontrolera. Smartfony również posiadają możliwość uruchamiania aplikacji, jednak mają niższą moc obliczeniową, mniejszy ekran oraz zazwyczaj jedynie ekran dotykowy lub klawiaturę numeryczną. Wszystkie smartfony, w przeciwieństwie do laptopów i tabletów, posiadają wbudowany moduł GSM, w związku z czym pozwalają na bardziej swobodny dostęp do Internetu. Urządzenia multimedialne zazwyczaj nie posiadają możliwości uruchamiania aplikacji, a obsługa odbywa się za pomocą przycisków dedykowanych dla danej funkcjonalności, np. przycisk rozpoczęcia nagrywania lub odtwarzania. Urządzenia specjalizowane posiadają ograniczony interfejs użytkownika albo nie posiadają go wcale, często mogą komunikować się jedynie z innymi urządzeniami za pomocą interfejsu stykowego lub bezprzewodowego. W ramach niniejszej pracy rozważane będą głównie urządzenia mobilne umożliwiające uruchamianie aplikacji o najbardziej ograniczonych zasobach, tj. smartfony, tablety oraz urządzenia IoT.

### 2.1.3 Internet Rzeczy

Internet rzeczy, zwany również Internetem przedmiotów lub po prostu IoT [108], jest określeniem na sieć połączonych ze sobą urządzeń, pojazdów czy nawet budynków, które są w stanie odbierać, przetwarzać i wysyłać dane. Jest to bardzo szeroka kategoria urządzeń elektronicznych, które najczęściej posiadają różnego rodzaju sensory, czujniki i kontrolery, a właściwie jedyną ich wspólną cechą jest łączność z siecią, przeważnie bezprzewodową [10].

W związku z wysoką specjalizacją tego typu urządzeń, nie posiadają one dużej mocy obliczeniowej, zasoby sprzętowe są zazwyczaj ograniczone do minimum. Urządzenia IoT są często mocno zintegrowane z usługami w chmurze obliczeniowej lub na innych zdalnych serwerach i służą jedynie jako warstwa inteligentnych czujników lub fizycznych kontrolerów dla złożonych systemów. Dobrym przykładem są tzw. systemy *smart home* [11], na które składa się sieć połączonych czujników ruchu, dymu, temperatury, kamer, kontrolerów światła czy termostatów, sterowanych za pomocą centralnej jednostki, która jest z kolei podłączona do internetu. Użytkownik ma

możliwość kontrolowania całego systemu poprzez odbieranie danych za pomocą innego urządzenia mobilnego, np. smartfona, tabletu lub po prostu komputera.

Do innych zastosowań IoT należą m.in.:

- inteligentny sprzęt AGD, np. lodówka podłączona do internetu i wyposażona w skaner kodów kreskowych,
- inteligentna przestrzeń medyczna, np. glukometr przesyłający pomiary bezpośrednio do lekarza,
- inteligentna kontrola ruchu, np. światła drogowe reagujące na aktualne natężenie ruchu,
- systemy wczesnego ostrzegania, np. czujniki wstrząsów sejsmicznych.

Eksperci szacują, że rynek IoT był warty 655,8 miliardów dolarów w 2014 roku, a do roku 2020 ma powiększyć się nawet do 1,7 biliona dolarów, a liczba dostępnych urządzeń przekroczy 50 miliardów [38]. Ogromne perspektywy finansowe oraz silne powiązanie z technologią chmury obliczeniowej doprowadziło do powstania wielu platform wspierających komunikację oraz budowę złożonych architektur IoT. Do najpopularniejszych należą:

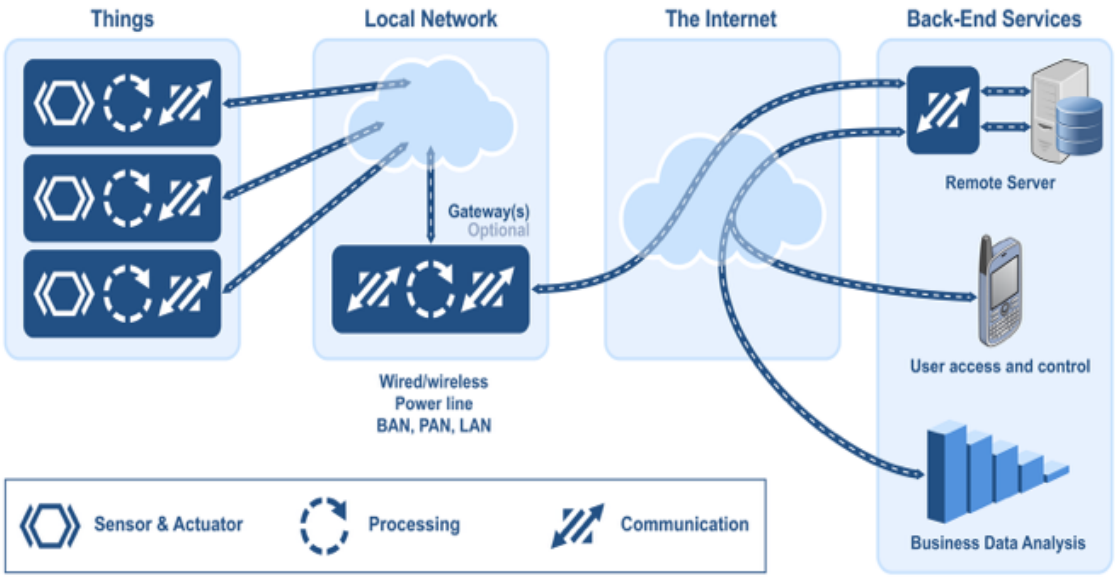
- Cisco Jasper [17],
- Amazon Web Services IoT [4],
- Microsoft Azure IoT Suite [70],
- Salesforce IoT Cloud [83].

Urządzenia IoT stymulują również rozwój pokrewnych technologii, głównie w dziedzinie komunikacji bezprzewodowej, ponieważ szybka sieć o niskim zużyciu energii jest kluczowa do działania wielu urządzeń w tego typu. Powszechnie wykorzystywane protokoły to:

- NFC - dla komunikacji zbliżeniowej, działającej w odległości do około 4cm [102],
- BLE (*ang. Bluetooth low energy*) - standard protokołu Bluetooth pozwalający urządzeniom pozostawać w trybie czuwania nawet do do roku na litowej baterii zegarkowej (CR2032) [33],
- ZigBee - technologia bazująca na protokole IEEE 802.15.4 i wykorzystująca kanał radiowy 2.4 GHz [46],

- WiFi-Direct - protokół oparty na WiFi, pozwalający jednak na komunikację peer-to-peer, czyli bez wykorzystania dodatkowego punktu dostępowego [5].

Aplikacje wykorzystujące urządzenia IoT są zazwyczaj w większości uruchamiane w chmurze lub na zewnętrznym komputerze. Na urządzeniu tego typu uruchomione jest jedynie oprogramowanie systemowe, różnego rodzaju sterowniki i proste programy przetwarzające dane, zwykle w sposób sekwencyjny. Można więc powiedzieć, że aplikacje IoT są aplikacjami rozproszonymi w środowisku heterogenicznym. Ogólny schemat architektury typowej aplikacji został przedstawiony na rysunku 2.1.



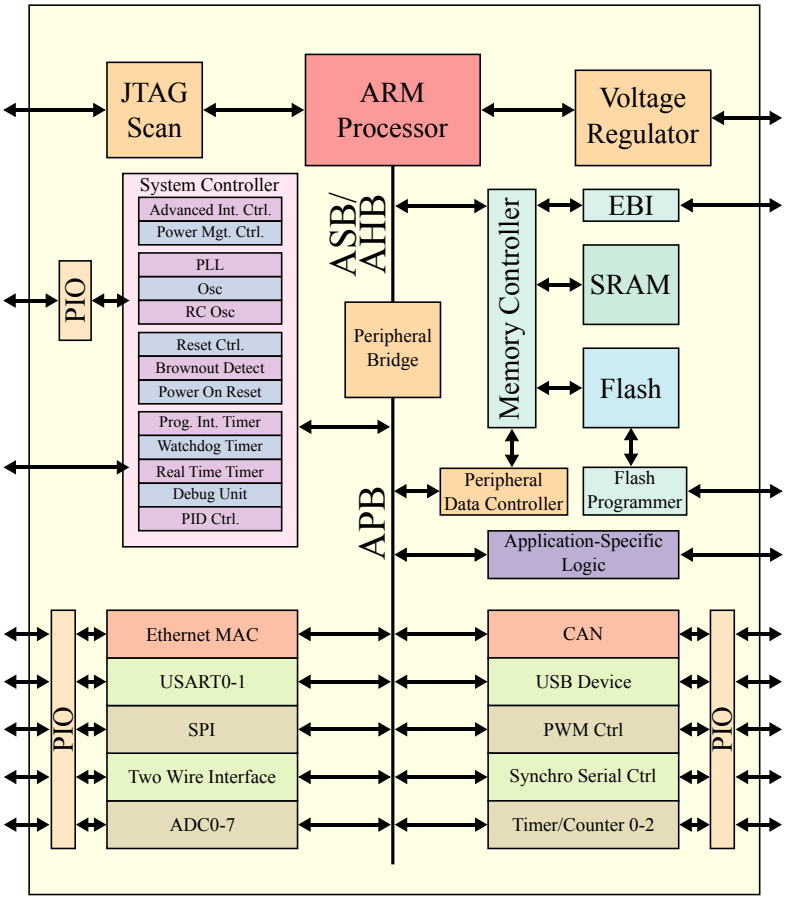

Rysunek 2.1: Architektura aplikacji IoT. Źródło: `https://www.micrium.com/`

Architektura aplikacji IoT zazwyczaj opiera się na asynchronicznych zdarzeniach, które przetwarzane są na wielu poziomach systemu. Na poziomie urządzeń odbywa się najprostsze przetwarzanie: konwersja sygnałów odbieranych przez czujniki i ich filtrowanie. W niektórych przypadkach mamy do czynienia z warstwą pośrednią, czyli urządzeniami o większej mocy obliczeniowej, które odpowiedzialne są za agregację zdarzeń i komunikację z chmurą. Właściwa analiza strumieni danych z dużej liczby urządzeń IoT odbywa się na zdalnych serwerach w chmurze obliczeniowej lub częściowo na urządzeniu użytkownika końcowego, który odbiera dane. W wielu przypadkach wykorzystuje się technologię SOA (*ang. Service Oriented Architecture*), która umożliwia przedstawienie funkcji urządzeń jako różnego typu usług dostępnych w środowisku obliczeniowym.

### 2.1.4 Architektura urządzeń mobilnych

Większość urządzeń mobilnych zbudowana jest w oparciu o układ scalony typu SoC (*ang. System-on-a-chip*). Jest to układ, w którym zintegrowane są wszystkie podzespoły urządzenia, typowo są to [7]:

- mikroprocesor,
- bloki pamięci RAM, ROM, FLASH,
- zegary i liczniki,
- kontrolery interfejsów, np. USB (*ang. Universal Serial Bus*), Ethernet,
- przetworniki analogowo-cyfrowe i cyfrowo-analogowe,
- regulatory napięcia i obwody zarządzania zasilaniem.




Rysunek 2.2: Schemat układu SoC. Źródło: `http://www.wikipedia.org`

Mikroprocesory w urządzeniach mobilnych są najczęściej zaprojektowane w ar-

chitekturze z rodziny ARM (*ang. Advanced RISC Machine*) [58], należących do architektur typu RISC (*ang. reduced instruction set computing*), tzn. o relatywnie małej liczbie rozkazów w porównaniu np. do architektury x86. Uproszczona architektura pozwala na redukcję liczby tranzystorów, co przekłada się na zmniejszenie kosztów, a także ograniczenie wydzielanego ciepła i zapotrzebowania na energię. Większość procesorów ARM wspiera 32-bitowe adresowanie, a architektura ARMv8-A także 64-bitowe. Schemat układu SoC opartego o architekturę ARM został przedstawiony na rys. 2.2.

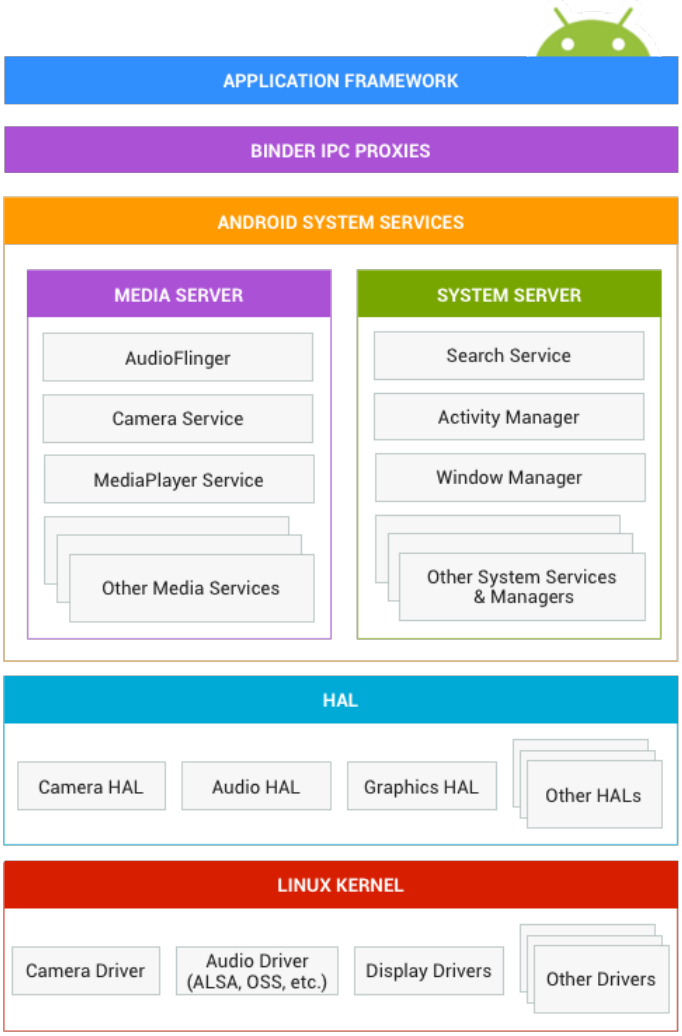
Wykorzystanie energii pozostaje jednym z największych problemów urządzeń mobilnych. W związku z powolnym rozwojem technologicznym baterii, producenci urządzeń oraz układów scalonych przywiązują szczególną uwagę do redukcji poboru mocy przez poszczególne podzespoły [65][95]. Z drugiej strony, optymalizacja wykorzystania energii jest równoważona przez ciągłe zwiększanie wydajności i mocy obliczeniowej urządzenia, w związku z czym nie obserwuje się znacznego wydłużenia czasu działania urządzeń mobilnych wykorzystujących baterie jako źródło zasilania.

### 2.1.5 System operacyjny urządzeń mobilnych

Najpopularniejszymi systemami operacyjnymi w urządzeniach mobilnych są Google Android (80,7% rynku), Apple iOS (17,7%) oraz Windows Phone (1,1%) [31]. W ramach niniejszej pracy rozważany będzie system Android ze względu na największy udział rynkowy oraz otwarty kod źródłowy.

System Android oparty jest o jądro Linuxa, zawiera jednak pewne zmiany, jak np. inny komponent zarządzania pamięcią oraz moduł oszczędzania energii oparty o tzw. *wakelock* [3]. Jądro jest również zwykle rozszerzone o moduły sterowników typowe dla danego urządzenia, np. sterowniki wyświetlacza, kamery, dźwięku. Pomiędzy usługami systemowymi a jądrem znajduje się warstwa HAL (*ang. hardware abstraction layer*) pozwalająca na komunikację z warstwą sprzętową - rys. 2.3.

Warstwę usług można podzielić na dwie grupy: usługi mediów oraz usługi systemu. Do usług mediów należą np. moduły pozwalające na odtwarzanie i nagrywanie wideo, rozpoznawanie głosu. Usługi systemowe to np. zarządzanie powiadomieniami, wyszukiwanie i dostęp do systemu plików. Warstwa usług pośredniczy pomiędzy sprzętem i jądrem systemu a poszczególnymi aplikacjami zbudowanymi w oparciu o framework aplikacyjny. Framework zawiera m.in. biblioteki API (*ang. application programming interface*) dostępne dla programistów budujących aplikacje dla użytkownika końcowego. Bardzo istotnym elementem systemu Android jest




Rysunek 2.3: Architektura systemu Android. Źródło: `http://www.android.com`

również moduł IPC (*ang. inter-process communication*), który umożliwia komunikację pomiędzy poszczególnymi procesami systemu, najczęściej pomiędzy aplikacją, a usługami systemowymi.

### 2.1.6 Klasyfikacja aplikacji mobilnych

Natywne aplikacje dla systemu Android, tzn. aplikacje uruchamianie bezpośrednio pod kontrolą systemu operacyjnego, mogą być pisane w języku Java oraz C/C++. Aplikacje są uruchamiane w środowisku ART (*ang. Android Runtime*), które w odróżnieniu od klasycznej maszyny wirtualnej Javy, kompiluje kod pośredni do kodu maszynowego już podczas instalacji, a nie uruchomienia. Oprócz aplikacji natyw-

nych popularne są również aplikacje webowe lub hybrydowe, uruchamianie w przeglądarce internetowej i oparte głównie o język JavaScript i HTML.

Aplikacje hybrydowe są szczególnie ciekawą koncepcją, ponieważ umożliwiają programowanie aplikacji w języku JavaScript, które działają na różnych systemach operacyjnych, zachowując jednak wydajność i możliwości aplikacji natywnych, dzięki zastosowaniu wydajnej warstwy abstrakcji zaimplementowanej w języku natywnym. W przeciwieństwie do zwykłych aplikacji webowych pozwalają na wykorzystanie wszystkich możliwości urządzenia, tzn. usług mediów i usług systemowych, oraz wprowadzają bardzo niewielki narzut wydajnościowy. W związku z tym aplikacje hybrydowe wydają się przyszłością w systemach mobilnych i tego typu aplikacje będą rozważane przy implementacji proponowanego szkieletu do współpracy urządzenia mobilnego z chmurą obliczeniową.

Aplikacje na urządzenia mobilne można podzielić na kategorie uwzględniając rodzaj wykorzystywanej architektury, poziom interaktywności oraz obszary zastosowań. Podział względem architektury wygląda następująco:

- aplikacje webowe - działające w przeglądarce, wymagające ciągłego dostępu do Internetu;
- aplikacje natywne - działające w natywnym środowisku systemu operacyjnego, mogą, ale nie muszą wymagać dostępu do Internetu;
- aplikacje hybrydowe - działające w oknie przeglądarki umieszczonym w aplikacji natywnej, mogą, ale nie muszą wymagać dostępu do Internetu;

Można wyróżnić następujące poziomy interaktywności aplikacji mobilnych:

- pasywne, np. odtwarzacz wideo, muzyki, aplikacje OCR,
- ograniczona interakcja, np. komunikatory, klient poczty e-mail, czytniki RSS, sieci społecznościowe,
- interaktywne, np. gry, edytory tekstu, symulacje 3D.

Aplikacje pasywne wymagają bardzo małej liczby interakcji użytkownika, charakteryzują się długotrwałym przetwarzaniem wprowadzonych danych lub długotrwałym odbieraniem danych przez użytkownika. Aplikacje o ograniczonej interakcji zazwyczaj realizują interakcje nie częściej niż raz na kilka sekund, pomiędzy którymi użytkownik odbiera i analizuje dostępne dane. W przypadku aplikacji w pełni interaktywnych interakcje zachodzą nawet kilka razy w ciągu sekundy, przy czym użytkownik odbiera i reaguje na prezentowane dane.

Do pięciu najpopularniejszych kategorii aplikacji z uwagi na obszary zastosowań należą kolejno:

- wszelkiego rodzaju gry, np. gry logiczne, zręcznościowe, sportowe czy przygodowe
- aplikacje rozrywkowe - związane najczęściej z multimediami (filmy, muzyka, zdjęcia) lub mediami społecznościowymi,
- aplikacje personalizacyjne - służące dostosowaniu urządzenia mobilnego do potrzeb użytkownika, np. zbiory tapet, dzwonków, itp.,
- aplikacje narzędziowe - ułatwiające pracę, np. kalendarze, programy do obsługi poczty e-mail,
- aplikacje edukacyjne - różnego rodzaju słowniki, encyklopedie w formie cyfrowej.

## 2.2 Chmura obliczeniowa

Termin *chmura obliczeniowa* lub po prostu *chmura* jest powszechnie używany w literaturze jako przykład wirtualizacji zasobów obliczeniowych. Najpopularniejsza definicja jest dość szeroka i opisuje chmurę obliczeniową jako model dostarczania powszechnego i praktycznego dostępu sieciowego do współdzielonej i konfigurowalnej puli zasobów komputerowych (sieci, serwerów, pamięci masowej, aplikacji i usług), która może być szybko dostarczona i zwolniona z minimalnym narzutem na zarządzanie lub działania ze strony usługodawcy [68].

W zależności od kontekstu chmura może być rozumiana jako model architektury, model dystrybucji oprogramowania, a także jako szeroko rozumiane zdalne zasoby sprzętowe [36]. Wspólnym elementem wszystkich definicji jest z pewnością idea dostarczania infrastruktury, platformy czy aplikacji przez zewnętrznego usługodawcę, w odróżnieniu od obsługi procesów obliczeniowych wewnątrz organizacji.

### 2.2.1 Motywacje rozwoju chmury

Początki modelu biznesowego, który dzisiaj można uznać za wczesną wersję modelu chmury obliczeniowej, sięgają 1957 roku, kiedy w literaturze pojawił się pomysł wynajmowania przez duże korporacje niewykorzystywanych zasobów obliczeniowych [67]. W latach 90', gdy Internet przeżywał bardzo szybki rozwój, zasoby te były sprzedawane najczęściej w postaci tzw. hostingu. Usługa ta pozwalała na

umieszczenie strony internetowej lub prostej aplikacji webowej na maszynach znajdujących się w serwerowniach usługodawcy w ramach za ustaloną miesięczną lub roczną opłatę. Usługodawcy hostingu oferowali najczęściej także dodatkowe usługi, takie jak kopie zapasowe, sprzedaż domen czy monitorowanie.

W przypadku bardziej zaawansowanych aplikacji webowych prosty hosting nie był wystarczający. Klienci, którzy wymagali specyficznego systemu operacyjnego, bibliotek lub po prostu dużej mocy obliczeniowej na wyłączność, korzystali z tzw. serwerów dedykowanych. Serwer dedykowany jest to po prostu fizyczna maszyna w całości wynajętą jednemu klientowi, ale w dalszym ciągu umieszczona w serwerowni usługodawcy. Z oczywistych względów, koszty serwerów dedykowanych były znacznie wyższe niż hostingu. Dla mniejszych firm, organizacji czy nawet osób prywatnych były to koszty zaporowe. Odpowiedzią na tę niszę były tzw. serwery VPS (*ang. Virtual Private Server*). Miały one znacznie większe możliwości w porównaniu do hostingu, umożliwiały uruchamianie własnego oprogramowania i dostęp przez SSH (*ang. secure shell*). Były też kilkukrotnie tańsze niż serwery dedykowane, ponieważ na jednej fizycznej maszynie uruchomionych było nawet do kilkudziesięciu serwerów VPS.

Wymagania rynku IT z czasem rosły i te trzy rodzaje usług przestały być wystarczające. Do nowych potrzeb rynku można zaliczyć:

- Skalowalność – szybki rozwój aplikacji webowych wymagał równie szybkiego rozszerzania liczby maszyn, na których te aplikacje działały. Klienci wymagali czasu uruchomienia nowego serwera liczonego w minutach, a nie w dniach czy godzinach. W związku z tym również koszty wynajmu maszyn powinny być liczone z dokładnością do minut.
- Elastyczność – klienci chcieli dobierać moc pojedynczych maszyn w zależności od potrzeb, a także np. migrować maszyny pomiędzy serwerowniami. Było to dość trudne do wykonania na istniejących fizycznych serwerach dedykowanych.
- Izolacja – rozwój aplikacji w kierunku usług i mikroserwisów stworzył potrzebę dostosowania infrastruktury do heterogenicznych architektur rozproszonych, będących w pewnym sensie rozszerzeniem istniejących wcześniej systemów gridowych [73]. Pojedyncza aplikacja często złożona jest z wielu usług działających w różnych środowiskach, wykorzystujących różne systemy operacyjne. Przeznaczenie do tego celu osobnych serwerów dedykowanych często bywa nieopłacalne, więc lokalizuje się wiele aplikacji w wirtualnym środowisku

rozproszonym. Dzięki mechanizmom izolacji użytkownik nie jest świadomy że dzieli zasoby z innymi użytkownikami.

- Bezpieczeństwo – duża presja na zapewnienie bezpieczeństwa danych i stabilności dostępu do usług w nowoczesnych systemach informatycznych stworzył potrzebę izolacji fragmentów sieci i pamięci masowej nawet w ramach jednej aplikacji. Dzięki wirtualizacji znacznie ogranicza się niebezpieczeństwo wycieku danych po włamaniu na jedną z maszyn, a także ryzyko zagłodzenia usług np. przez ataki DDoS (*Distributed Denial of Service*). Fizyczna izolacja tych zasobów, podobnie jak w przypadku serwerów, była bardzo droga lub po prostu niemożliwa.
- Poziomy zarządzania – w zależności od zastosowań klienci chcieli mieć pełen dostęp do maszyny jak w przypadku serwerów dedykowanych albo jedynie uruchamiać aplikację, pozostawiając zarządzanie maszynami usługodawcy. Pojawiło się także zapotrzebowanie na usługi w pełni zarządzane – np. usługa przechowywania danych lub usługa przetwarzania dużych zbiorów danych (Hadoop [104] czy Spark [43]). Często wymagane jest również łączenie tych powyższych podejść dla jednego użytkownika.

Warstwa usług i aplikacji zarządzanych w całości przez usługodawcę, uruchamianych na zdalnych serwerach i globalnie dostępnych przez Internet stała się z obecnie lepszą alternatywą dla praktycznie każdego rodzaju klasycznych aplikacji desktopowych: edytorów tekstów, programów graficznych czy księgowych. Takie aplikacje dostępne w chmurze zyskują coraz większą popularność przede wszystkim ze względu na łatwość ich zarządzania. Instalowanie, tworzenie kopii zapasowych czy aktualizacja nie wymaga żadnych czynności ze strony klienta. Zapewnienie przy tym skalowalności obliczeń w chmurze – często nawet dla tysięcy czy nawet milionów użytkowników – prowadzi do znacznego obniżenia kosztów. Czyni to chmurę obliczeniową bardzo atrakcyjną do wykorzystania w dużych organizacjach czy firmach.

### 2.2.2 Charakterystyka chmury obliczeniowej

W literaturze najczęściej jest mowa o pięciu najważniejszych charakterystykach chmury obliczeniowej [68]:

- Dostępność na żądanie – Klient może uruchomić lub zatrzymać usługi, takie jak serwery czy magazyny danych, bez interakcji międzyludzkiej. Najczę-

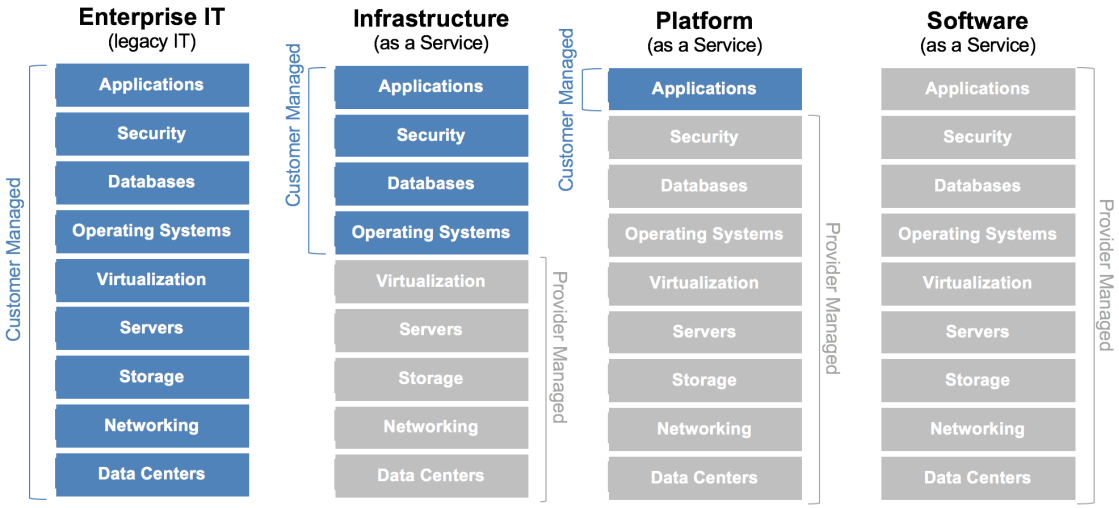
ściej może być to wykonywane w sposób automatyczny, za pomocą odpowiednich API.

- Dostępność przez sieć – Usługi dostępne są przez sieć, za pomocą standardowych protokołów, takich jak HTTP, SOAP czy REST. Dzięki temu mogą być wykorzystywane w szerokim zakresie aplikacji, zarówno na stacjach roboczych, jak również na smartfonach.
- Agregacja zasobów – Usługodawca agreguje zasoby, które są następnie oferowanie wielu klientom na żądanie. Wielu klientów może korzystać z tego samego zasobu (np. fizycznego serwera) w tym samym czasie, jednak ich środowisko jest w pełni izolowane (np. za pomocą wirtualizacji). Klient ma poczucie niezależności od lokalizacji, ponieważ nie ma kontroli ani wiedzy nad tym który dokładnie zasób zostanie mu przydzielony. Jest jednak możliwość wybrania lokalizacji na wyższym poziomie abstrakcji, np. na poziomie kraju, miasta lub serwerowni.
- Elastyczność – Przydzielone zasoby mogą być w zależności od potrzeb dynamicznie poszerzane i pomniejszane, w pewnych przypadkach w pełni automatycznie, w celu zapewnienia skalowalności obliczeń. Z punktu widzenia klienta zasoby chmury są wręcz nieograniczone i mogą być wykorzystane w dowolnej ilości oraz w dowolnym czasie.
- Monitorowanie – Systemy w chmurze automatycznie mierzą i optymalizują wykorzystanie zasobów, np. poprzez migrację i agregację mniej wymagających usług. Usługodawca zapewnia pełne dane dotyczące wykorzystania zasobów przez klientów, co może umożliwić im zadbanie o efektywność wykorzystania zasobów.

### 2.2.3 Modele chmury obliczeniowej

W ramach chmury obliczeniowej można wyróżnić trzy najpopularniejsze modele zarządzania różniące się zakresem działań dotyczących klienta oraz dostawcy chmury. Modele te bazują na rozwiązaniu warstwowym, gdzie każda warstwa oferuje pewien zakres usług. Rysunek 2.4 przedstawia porównanie klasycznego modelu zarządzania infrastrukturą i oprogramowaniem z trzema modelami chmury obliczeniowej.

- IaaS (*ang. Infrastructure as a Service*) – Podstawowy model oferowania usług w chmurze, który polega na dostarczaniu infrastruktury w postaci usług. Przy-




Rysunek 2.4: Porównanie trzech modeli chmury obliczeniowej i klasycznego modelu zarządzania. Źródło: `http://floris.vanenter.nl/`

kładem usługi może być np. uruchomienie maszyny wirtualnej. Dzięki wirtualizacji klient nie musi znać szczegółów na temat fizycznych urządzeń odpowiedzialnych za dostarczenie zasobów obliczeniowych. Oprócz zasobów obliczeniowych usługodawca najczęściej oferuje również gamę usług dodatkowych, np. repozytorium obrazów maszyn wirtualnych, pamięć masową, obiektową bazę danych, zapory sieciowe, usługi równoważenia obciążenia, adresy IP, wirtualne sieci i predefiniowane pakiety oprogramowania.

- PaaS (*ang. Platform as a Service*) – Model ten oferuje usługi dla twórców aplikacji. Usługodawca dostarcza odpowiednie biblioteki programistyczne do budowy aplikacji oraz środowisko uruchomienia aplikacji [55]. W odróżnieniu od modelu IaaS, programista nie musi zarządzać konkretnymi zasobami obliczeniowymi. W ramach środowisk uruchomienia aplikacji dostarczane są także dodatkowe usługi, jak np. automatyczne skalowanie, mechanizmy failover, systemy monitorowania i logowanie błędów aplikacji.
- SaaS (*ang. Software as a Service*) – W tym modelu użytkownik otrzymuje dostęp bezpośrednio do aplikacji. Usługodawca zarządza infrastrukturą, platformą uruchomieniową i oprogramowaniem. Model ten czasami nazywany jest *oprogramowaniem na żądanie* i zazwyczaj rozliczny jest w modelu pay-per-use. Dostęp do aplikacji odbywa się najczęściej przez przeglądarkę internetową i nie wymaga instalowania dodatkowego oprogramowania. Jednym z najczęściej

wymienianych problemów tego modelu jest brak pełnej kontroli użytkownika nad danymi, które przechowuje usługodawca.

Istnieją także inne modele chmury obliczeniowej, które powstały jako specjalistyczne zastosowania w ramach jednego z trzech głównych modeli lub pewnej ich kombinacji [80]. Przykładem jest np. CaaS (*ang. Communication as a Service*), który polega na dostarczaniu usług komunikacyjnych, takich jak poczta e-mail, wideokonferencje czy komunikatory internetowe. Innym modelem, który początkowo był częścią PaaS jest MaaS (*ang. Monitoring as a Service*) – oferuje różnego rodzaju usługi monitorowania infrastruktury i aplikacji, a także logowanie i analizę błędów.

### 2.2.4 Aplikacje użytkowe

Popularnym modelem budowy aplikacji użytkowych opartych o chmurę obliczeniową jest model trójwarstwowy. Wyróżniamy w nim następujące warstwy:

1. Warstwa danych - obsługuje żądania dostępu do danych, np. bazy danych, usługi big data, zasoby sieciowe, pamięć masową.

2. Warstwa logiki - opisuję logikę biznesową zachowania aplikacji, zarządza transakcjami, obsługuję uwierzytelnianie użytkownika oraz integruje różne usługi platformy chmury obliczeniowej.

3. Warstwa prezentacji - interfejs (GUI) dostępny dla końcowego użytkownika aplikacji, najczęściej dostarczany w postaci aplikacji webowej.

Warstwa danych jest szczególnie istotna w aplikacjach działających w modelu chmury obliczeniowej, ponieważ dane są zazwyczaj wysoce rozproszone na wiele maszyn, czasami pomiędzy serwerami zlokalizowanymi w geograficznie odległych regionach. Zadaniem tej warstwy jest zapewnienie integracji tak aby rozproszenie danych było przezroczyste dla logiki biznesowej. Popularnym rozwiązaniem stosowanym w chmurach obliczeniowych jest zapewnienie usług dostępu do danych przez protokoły usług sieciowych, np. REST. Ze względów wydajnościowych warstwa danych często jest silnie związana z warstwą logiki, ponieważ wykonuje część obliczeń na dużych zbiorach danych (big data). Do takich operacji należą m.in. algorytmy mapowania i redukcji (map-reduce).

Warstwa logiki biznesowej również w pełni wykorzystuje zasoby chmury, przez zastosowanie rozproszonego i współbieżnego przetwarzania żądań. Asynchroniczne

metody przetwarzania danych stosowane w nowoczesnych językach i serwerach aplikacyjnych pozwalają na uzyskanie wysokiej skalowalności aplikacji i obsługę tysięcy żądań na sekundę. Typowo, warstwa logiki biznesowej wymaga największych zasobów obliczeniowych, wyjątkiem są aplikacje z dziedziny big data, gdzie ciężar obliczeń został przeniesiony do warstwy danych.

Interfejs użytkownika, czyli warstwa prezentacji w aplikacjach w chmurze obliczeniowej zazwyczaj realizowana jest w koncepcji tzw. cienkiego klienta. Najpopularniejszą formą dystrybucji jest aplikacja webowa, dzięki czemu dostęp do aplikacji użytkowej możliwy jest przez przeglądarkę, z dowolnego systemu operacyjnego w tym z urządzeń mobilnych. Wiąże się to jednak z koniecznością zapewnienia ciągłego dostępu do internetu.

W ramach każdej z warstw można dokonać dalszego podziału na komponenty, czyli grupy ściśle ze sobą związanych funkcji i usług, które posiadają wyspecyfikowany interfejs wymiany danych. Każdy komponent aplikacji powinien być w stanie działać niezależnie od pozostałych, a co za tym idzie powinien być możliwy do wyizolowania. Ma to duże znaczenie w przypadku automatycznego testowania aplikacji, monitorowania wydajności i skalowania przez rozproszenie komponentów na wiele węzłów obliczeniowych. Implementacja komponentów może być różna i zależy od konkretnego zastosowania. Może to być np. klasa w językach obiektowych lub mikrousługa w przypadku aplikacji opartych na architekturze SOA.

Aplikacje mobilne implementowane są również jako aplikacje trójwarstwowe. W przypadku aplikacji wykorzystujących chmurę obliczeniową warstwa logiki i warstwa danych są najczęściej umieszczone w chmurze, a interfejs użytkownika zlokalizowany jest na urządzeniu mobilnym. W ogólności komponenty i dane mogą być dowolnie rozdzielone pomiędzy chmurą i urządzeniem na etapie projektowania aplikacji. Naturalnym podziałem danych jest umieszczenie w chmurze danych z zewnętrznych źródeł i danych współdzielonych, natomiast lokalny stan aplikacji powinien znajdować się na urządzeniu. Z kolei podział logiki biznesowej nie jest już tak oczywisty. Z uwagi na to, że chmura obliczeniowa zapewnia wysoki poziom abstrakcji dostępu do danych w postaci usług sieciowych, komponenty przetwarzające dane mogą być z powodzeniem uruchomione w chmurze, jak również na urządzeniu mobilnym.

## 2.3 Współpraca urządzeń mobilnych z chmurą

Integracja pomiędzy urządzeniami mobilnymi i chmurą obliczeniową, zwana popularnie MCC (*ang. Mobile Cloud Computing*), jest rozwiązaniem popularnym i nieskomplikowanym w założeniach, opierającym się głównie na wymianie komunikatów przez ustandaryzowane protokoły typu SOAP (*ang. Simple Object Access Protocol*) i REST (*ang. Representational State Transfer*). Jednocześnie połączenie aplikacji mobilnej z usługami w chmurze wymaga dość dużego narzutu implementacyjnego: obsługi wyżej wymienionych protokołów, rozwiązywania problemów z dostępnością sieci, obsługi sytuacji wyjątkowych, synchronizacji danych, itp. Dodatkowym problemem jest konieczność zaprojektowania zupełnie nowej architektury aplikacji oraz zadecydowanie, które fragmenty aplikacji powinny zostać zaimplementowane jako usługi zdalne. Powstała potrzeba opracowania rozwiązań wspierających projektowanie i implementację takich aplikacji mobilnych.

### 2.3.1 Problemy integracji

Mimo szybkiego rozwoju urządzeń mobilnych, nie zawsze spełniają one wymagania użytkowników, szczególnie w specjalistycznych zastosowaniach. Ponieważ postęp w tej dziedzinie zbiegł się w czasie z popularyzacją idei chmury obliczeniowej, naturalnym wydawało się rozszerzenie możliwości urządzeń o praktycznie nieograniczoną moc obliczeniową w chmurze [27].

Ograniczone możliwości przetwarzania na urządzeniach mobilnych spowodowane ich wolniejszymi podzespołami i niewielką pojemnością pamięci są największym wyzwaniem dla programistów aplikacji mobilnych. Użytkownicy oczekują aplikacji o możliwościach obliczeniowych porównywalnych z komputerami stacjonarnymi lub laptopami, natomiast możliwości tego typu urządzeń są zazwyczaj kilkukrotnie mniejsze. Kolejnym problemem są ograniczenia mobilnych systemów operacyjnych, które nie zostały przystosowane do obsługi dużej liczby procesów, a raczej jednej aplikacji działającej na pierwszym planie [1]. Wszelkie zadania działające w tle, do których przywykli użytkownicy komputerów osobistych, najczęściej są wstrzymywane lub usypiane przez system w celu przydzielenia większej ilości zasobów do aktualnie używanej aplikacji lub oszczędzania energii.

Aplikacje oparte na zdalnych usługach w chmurze obliczeniowej stają się aktualnie najpopularniejszym rozwiązaniem w wielu dziedzinach, więc nie ulega wątpliwości, że zmiany te dotykają również bardzo młodego i dynamicznego rynku usług

mobilnych. Według badań, już ponad 240 milionów firm używa rozwiązań w chmurze za pomocą urządzeń mobilnych. Przychody w tym sektorze szacuje się na ponad 5 miliardów dolarów rocznie [23]. Rozwój integracji pomiędzy chmurą obliczeniową i urządzeniami mobilnymi nie byłby możliwy bez znacznego postępu w rozwiązaniach dotyczących wirtualizacji, przepustowości sieci bezprzewodowych i rozwoju infrastruktury dostawców chmury [59]. Mimo tego postępu, wiele problemów pozostaje nierozwiązanych. Do najpoważniejszych należy m.in. niestabilność sieci komórkowej, która może zmieniać swoją przepustowość i opóźnienie w zależności od zakłóceń środowiska, takich jak pogoda, budynki, a nawet prędkość przemieszczania się [62].

Heterogeniczność w środowisku urządzeń mobilnych również jest wyzwaniem dla programistów. Urządzenia, mogą nie tylko różnić się w znaczny sposób mocą obliczeniową i dostępnością pamięci, ale także architekturą mikroukładów, szczególnie jeśli uwzględnimy urządzenia IoT. Na rynku popularnych jest kilka mobilnych systemów operacyjnych, które są ze sobą praktycznie niekompatybilne i wymuszają na twórcach aplikacji implementację kilku rozwiązań lub wykorzystywanie dodatkowych bibliotek zapewniających niezbędną warstwę abstrakcji – co naturalnie wiąże się z pewnym narzutem na wydajność, jak również możliwości wykorzystania aplikacji [84].

Biorąc pod uwagę stale rosnące zapotrzebowanie aplikacji na moc obliczeniową, ograniczoną pojemność baterii urządzeń mobilnych oraz bardzo zróżnicowaną przepustowość sieci, ogromnym wyzwaniem jest znalezienie odpowiedniego podziału między przetwarzaniem w chmurze i na urządzeniu mobilnym. Dodatkowo parametry urządzenia (poziom baterii, obciążenie) i sieci mogą zmieniać się dynamicznie w czasie życia aplikacji. W związku z tym zaproponowany w tej rozprawie doktorskiej szkielet aplikacji oraz moduł zarządzający pozwalający na integrację urządzenia mobilnego z chmurą będzie brał pod uwagę następujące parametry: czas przetwarzania, wykorzystanie energii i czas komunikacji oraz ich zmienność podczas działania aplikacji.

### 2.3.2 Przykłady stosowanych rozwiązań

Jednym z najpopularniejszych zastosowań smartfonów i tabletów jest odbieranie różnego rodzaju treści multimedialnych. Nie dziwi więc fakt, że w literaturze często spotykane są rozwiązania pozwalające na optymalizację właśnie tego typu aplikacji i usług. Podstawowym przykładem aplikacji multimedialnej jest odbieranie treści z zdalnego serwera znajdującego się w chmurze [60], ale można rozważać również

transmisję wideo pomiędzy wieloma użytkownikami urządzeń mobilnych [30]. Do bardziej zaawansowanych przykładów można zaliczyć aplikacje przetwarzające strumienie multimedialne, np. rozpoznające obiekty [92]. Ciekawym zastosowaniem, które szczególnie dobrze wpasowuje się w charakterystykę urządzeń mobilnych jest transmisja z wielu źródeł jednocześnie do chmury, a następnie łączenie nagrań w jeden strumień z wielu perspektyw [106]. Aplikacje multimedialne zasadniczo wymagają dużej przepustowości sieci oraz mocy obliczeniowej, jeśli strumienie mają być w jakiś sposób przetwarzane lub analizowane, więc integracja z chmurą obliczeniową z jednej strony wydaje się bardzo trafnym rozwiązaniem, ponieważ pozwala na znaczne zwiększenie mocy obliczeniowej, ale jednocześnie transfer dużej ilości danych na znaczną odległość jest sporym wyzwaniem [57].

Rozwiązania dla aplikacji multimedialnych skupiają się głównie wokół optymalizacji przesyłania danych. Należą do nich tzw. cloudlety, czyli serwery pośredniczące w komunikacji z chmurą, ale które znajdują się fizycznie dużo bliżej użytkownika [26]. Innym popularnym rozwiązaniem jest adaptacja jakości strumienia do potrzeb urządzenia mobilnego, która najczęściej polega na przetwarzaniu i kompresji multimediów w chmurze w czasie rzeczywistym [101][14], także przy wykorzystaniu kontekstu samego strumienia [52]. Zaproponowano również zupełnie nowe architektury dla przetwarzania aplikacji multimedialnych w chmurze obliczeniowej [87][41][109], przesyłanie strumieni w technologii P2P (*ang. Peer to Peer*) [90], a także kodeki dostosowane do potrzeb urządzeń mobilnych [47].

Podejścia do optymalizacji transferu danych o szerszych zastosowaniach skupiają się na dopasowaniu jakości usług do kontekstu przesyłanych danych, dzięki czemu można ograniczyć wykorzystanie procesora urządzenia mobilnego, a także zmniejszyć użycie niezbędnej energii [15]. Istnieją także rozwiązania umożliwiające zrównoleglenie przetwarzania strumieni danych w aplikacjach mobilnych [105].

Innym ciekawym zastosowaniem dla MCC poruszanym w literaturze jest *mobile gaming*, czyli wykorzystanie chmury obliczeniowej do wspomagania różnego rodzaju gier na urządzenia mobilne. Rozwiązania skupiają się na renderowaniu scen 3D w chmurze i przesyłaniu jej do urządzenia w postaci strumienia video, ponieważ w przypadku większości gier to właśnie renderowanie grafiki jest najbardziej obciążającym procesem. Proponuje się m.in. metodę dynamicznej optymalizację renderowania w zależności od stopnia złożoności sceny [100], a także opracowanie kodeka obrazu zoptymalizowane pod kątem gier [91].

Kolejnym zastosowaniem, silnie związanym z multimediami, jest tzw. rzeczywi-

stość rozszerzona lub AR (*ang. augmented reality*). Są to aplikacje łączące świat rzeczywisty z obrazami generowanymi komputerowo, zazwyczaj poprzez nałożenie obiektów na obraz z kamery urządzenia mobilnego. Wymaga to złożonego przetwarzania obrazu, śledzenia rzeczywistych obiektów w strumieniu wideo oraz renderowania obiektów wirtualnych. Chmura obliczeniowa nadaje się do wykorzystania właściwie na wszystkich etapach działania takich aplikacji [42].

Urządzenia mobilne ze względu na swoją przenośność oraz różnego rodzaju czujniki i sensory, takie jak GPS czy żyroskop, często wykorzystywane są do śledzenia i lokalizacji użytkownika. Również w tym przypadku wykorzystanie chmury obliczeniowej pozwala na dostarczenie funkcjonalności, które inaczej nie byłyby osiągalne. Np. w przypadku próby określenia lokalizacji wewnątrz budynku nadajnik GPS jest praktycznie bezużyteczny i koniecznie jest wykorzystanie innych rozwiązań, takich jak analiza otoczenia za pomocą kamery [6]. Także w przypadku monitorowania zachowań grupy osób w trudnych warunkach, np. strażaków czy policjantów, chmura staje się niezbędna do synchronizacji danych [81]. Kolejną grupą aplikacji wymagającą monitorowania użytkownika urządzenia są aplikacje związane z ochroną zdrowia [79]. W tym przypadku wymagana jest dużo niezawodność działania.

### 2.3.3 Istniejące rozwiązania

W literaturze zostało zaproponowanych wiele rozwiązań integrujących urządzenia mobilne z chmurą obliczeniową [45]. Różnią się one zakresem integracji i celem optymalizacji, niektóre z nich zostały zaprojektowane w celu zmaksymalizowania wydajności przetwarzania, inne z kolei, nastawione są głównie na minimalizację zużycia energii. Część rozwiązań wymaga ręcznej optymalizacji aplikacji, przez podzielenie jej na komponenty wykonywane w chmurze i na urządzeniu już na etapie implementacji. Istnieje również kilka propozycji, które biorą pod uwagę optymalizację wielokryterialną. Często do wdrożenia większości rozwiązań wymagane są znaczne zmiany w systemie operacyjnym urządzenia. Najbardziej popularne metody zostały przedstawione w tabeli 2.1.

CloneCloud [16] jest modelem opartym o klonowanie systemu operacyjnego urządzenia mobilnego na maszynie wirtualnej działającej na zdalnym serwerze. Kiedy złożone obliczeniowo zadanie jest uruchomione na urządzeniu mobilnym proces zostaje wstrzymany, a stan przesyłany jest do sklonowanego urządzenia w chmurze. Specjalny moduł działający na urządzeniu decyduje w czasie wykonania czy przetwarzanie powinno zostać przeniesione. Migracja jest w pełni automatyczna i nie

Tabela 2.1: Porównanie dostępnych rozwiązań

| | **Automatyczna optymalizacja** | **Zmiany w aplikacji** | **Zmiany w systemie** | **Inne** |
|---|---|---|---|---|
| CloneCloud [16] | wydajność | brak | znaczne | - |
| Weblets [107] | wielokryterialna | znaczne | brak | wymaga projektowania aplikacji w specyficznej architekturze |
| $\mu$Cloud [66] | brak | znaczne | brak | - |
| Cloudlet [86] | brak | brak | znaczne | wymaga serwera z niskim czasem odpowiedzi (prywatnej chmury) |
| eXCloud [64] | brak | brak | brak | - |
| MAUI [19] | wielokryterialna | niewielkie | brak | bazuje na przestarzałej wersji systemu operacyjnego |
| ThinkAir [51] | energia lub wydajność | niewielkie | brak | dedykowany kompilator |
| Cuckoo [44] | brak | niewielkie | brak | - |

wymaga żadnych zmian w kodzie aplikacji. Wymagana jednak poważnych zmian w systemie operacyjnym urządzenia mobilnego, a konkretnie specjalnie przygotowanej implementacji maszyny wirtualnej Dalvik [40] w systemie Android. Architektura CloneCloud pozwala jedynie na optymalizację wydajności przetwarzanie i nie bierze pod uwagę zużycia energii lub kosztu transferu stanu aplikacji. Ponadto klonowanie całego systemu operacyjnego, wraz z wszystkimi danymi, jest bardzo złożonym procesem i wymaga szybkiego transferu danych do chmury. W praktyce takie warunki są trudne do osiągnięcia, szczególnie biorąc pod uwagę koszt transmisji danych w sieci GSM. Innym problemem jest kwestia bezpieczeństwa i ochrony danych - na chmurze znajduje się pełny klon urządzenia, który po przejęciu przez osoby nieupoważnione może zostać wykorzystany nawet do przejęcia kontroli nad urządzeniem. W związku z powyższymi problemami, rozwiązanie CloneCloud nie wydaje się najlepszą propozycją do współpracy urządzenia mobilnego z chmurą.

Kolejny model integracji urządzenia mobilnego i chmury zakłada, że aplikacja zbudowana jest z luźno powiązanych ze sobą komponentów - webletów [107]. Aplikacja zbudowana w ten sposób określana jest jako *aplikacja elastyczna* i wspiera trzy wzorce architektury: replikację, podział i agregację. Wzorzec replikacji polega

na zrównolegleniu wykonania webletu na urządzeniu mobilnym i w chmurze, dzięki czemu czas wykonania może być krótszy. Wzorzec podziału umożliwia uruchomienie innej implementacji webletu na chmurze obliczeniowej, która jest rozszerzeniem implementacji uruchamianej na urządzeniu. Pozwala to na wprowadzenie dodatkowych funkcjonalności, które nie mogłyby być zrealizowane bez udziału chmury. Przyjęty wzorzec agregacji polega na uruchomieniu kilku webletów w chmurze, które obsługują różne usługi, jak np. komunikatory czy sieci społecznościowe, a następnie zagregowane wyniki przesyłane są do urządzenia mobilnego. Proponowana optymalizacja aplikacji, czyli uruchomienie odpowiednich webletów w chmurze, przeprowadzona jest w czasie wykonania, wykorzystując model kosztu, który bazuje na wielu parametrach. Niestety autorzy nie podają żadnych szczegółów implementacji algorytmu optymalizacji, a przykładowa aplikacja wykorzystuje jedynie predefiniowaną konfigurację. W opisywanym modelu aplikacja musi zostać zaimplementowana od podstaw z wykorzystaniem dostarczonego SDK (*ang. Software Development Kit*).

W architekturze $\mu$Cloud [66] aplikacja mobilna zbudowana jest z wielu heterogenicznych komponentów. Implementacja aplikacji sprowadza się do orkiestracji istniejących komponentów w ramach *grafu wykonania*. Optymalizacja aplikacji musi być wykonana a priori przez programistę, a autorzy nie proponują żadnej metody optymalizacji w czasie wykonania co jest największą wadą tego rozwiązania. Ponadto aplikacja musi być implementowana od postaw w opisanym modelu komponentów.

Koncepcja Cloudlet'ów [86] bazuje na migracji całego systemu operacyjnego urządzanie mobilnego na zdalny serwer. Kiedy użytkownik uruchamia zadanie złożone obliczeniowo, maszyna wirtualna na urządzeniu mobilnym jest wstrzymywana i transferowana do chmury. Istniejące aplikacje mogą być migrowane bez żadnych zmian co jest największą zaletą tego rozwiązania. Zmiany muszą jednak zostać wprowadzone w systemie operacyjnym urządzenia, a ponadto transfer całej maszyny wirtualnej, czy nawet niewielkiej nakładki zawierającej jedynie zmiany jak proponują autorzy, wymaga bardzo szybkiego połączenia o niskim opóźnieniu transmisji danych. Cloudlety są rozwiązaniem bardzo podobnym do CloneCloud, jednak zakładają jeszcze głębszą integrację pomiędzy urządzeniem mobilnym i chmurą obliczeniową. W praktyce urządzenie działa jako cienki klient i w odróżnieniu od CloneCloud zawsze wymaga uruchomionej maszyny wirtualnej.

Model eXCloud (*ang. extensible cloud*) [64] bazuje na migracji ramek stosu maszyny wirtualnej Javy do chmury. Specjalizowany pre-procesor modyfikuje bajt-kod aplikacji przed załadowaniem jej do pamięci. Dzięki temu możliwe jest przechwyty-

wanie wywołań poszczególnych funkcji i migracja ich do chmury obliczeniowej, na której uruchomiona jest kopia aplikacji. Koncepcja ta jest ciekawa, ponieważ nie wymaga żadnych zmian w aplikacji czy systemie operacyjnym urządzenia. Niestety eXCloud zakłada migrację jedynie w przypadku braku wystarczających zasobów (np. pamięci) na urządzeniu i nie uwzględnia migracji w celu zwiększenia wydajności przetwarzania lub minimalizacji kosztów wykonania.

MAUI [19] to rozwiązanie pozwalające na optymalizację czasu wykonania i wykorzystania energii przez przekazanie wykonania pojedynczych metod do chmury obliczeniowej. Programista musi oznaczyć metody obecne w aplikacji przeznaczone do optymalizacji w kodzie źródłowym, a specjalny moduł optymalizacji decyduje w czasie wykonania aplikacji czy uruchomić je w chmurze obliczeniowej. W chmurze działa specjalnie skompilowana wersja aplikacji, która pozwala na odebranie danych z urządzenia mobilnego i przekazanie ich do oznaczonych metod. MAUI wymaga względnie niewielkich modyfikacji w kodzie aplikacji co jest jedną z największych zalet tego rozwiązania. Niestety oparte jest ono o nieaktualną już wersję frameworka .NET wykorzystywaną w systemie Windows Mobile 6.5, który przestał być rozwijany.

ThinkAir [51] wykorzystuje emulator systemu Android, który pozwala na uruchamianie aplikacji mobilnych na architekturze x86. Decyzja o tym czy wykonanie powinno zostać przeniesione do chmury obliczeniowej opiera się o aktualne dane dotyczące wykorzystania energii oraz przewidywanego czasu przetwarzania. W chmurze działa kopia aplikacji, która łączy się z aplikacją na urządzeniu mobilnym. Gdy wykonanie metody zostaje przeniesione do chmury, wszystkie obiekty przekazywane jako argumenty są serializowane i przesyłane przez `ObjectInputStream` dostępny w standardowej bibliotece Javy. Analogicznie do danych wejściowych, dane wyjściowe przesyłane są z powrotem do urządzenia przez `ObjectOutputStream`. Jednym z głównych problemów tego rozwiązania jest to, że koszt transferu danych i prędkość sieci nie jest brana pod uwagę przy optymalizacji. Ponadto aplikacja musi być kompilowana za pomocą dedykowanego kompilatora, który nie zawsze jest zgodny z aktualną wersją systemu Android i może nie wspierać niektórych funkcji lub wręcz nie działać na najnowszych urządzeniach.

Cuckoo [44] jest bardzo prostym rozwiązaniem, które w założeniach miało być łatwe do wykorzystania przez programistów. Aplikacja zawiera dwie implementacje wszystkich złożonych obliczeniowo funkcji: jedną do wykonania lokalnie na urządzeniu i drugą do wykonania w chmurze. Cuckoo nie wymaga do działania żadnych

zmian w systemie operacyjnym, ale decyzja o uruchomieniu przetwarzania w chmurze opiera się jedynie na dostępności zdalnej maszyny i nie uwzględnia optymalizacji wydajności, wykorzystania energii czy kosztów transferu. W praktyce, gdy urządzenie mobilne ma połączenie z chmurą obliczeniową wszystkie zaimplementowane funkcje uruchamiane są zdalnie, podobnie jak usługi sieciowe. Gdy nie ma połączenia, aplikacja wykorzystuje lokalne implementacje funkcji.

Opisane powyżej rozwiązania można podzielić na trzy grupy, według poziomów abstrakcji na których odbywa się współpraca z chmurą obliczeniową:

- system operacyjny: CloneCloud, Cloudlet,
- maszyna wirtualna Javy/.NET: eXCloud, MAUI,
- komponenty aplikacji: Weblets, $\mu$Cloud, ThinkAir, Cuckoo.

Można również dokonać podziału rozwiązań ze względu na funkcje automatycznej optymalizacji:

- wielokryterialna: Weblets, MAUI,
- jednokryterialna: CloneCloud, ThinkAir,
- brak optymalizacji: $\mu$Cloud, Cloudlet, eXCloud, Cuckoo.

Najbardziej pożądane jest więc rozwiązanie, które będzie oferować integrację na poziomie aplikacji oraz optymalizację automatyczną. Integracja na poziomie komponentów aplikacji konieczna jest ze względów praktycznych: aplikacja musi być łatwa do zainstalowania przez użytkownika, a wszelkie zmiany w systemie operacyjnym i maszynie wirtualnej utrudniają instalację lub nawet ją uniemożliwiają np. ze względu na blokowanie zapisu na partycji systemowej przez producentów urządzeń.

Jedynym z dostępnych rozwiązań, które mieści się w obu kategoriach są Weblety. Niestety, ze względu na ograniczenia nałożone na możliwą architekturę aplikacji, mają one dość niewielkie możliwości zastosowania w praktyce. Analiza pozostałych rozwiązań prowadzi do wniosku, że większość opisywanych modeli i architektur wykorzystywanych przy współpracy urządzeń mobilnych z chmurą obliczeniową wymaga podobnych problemów do rozwiązania:

1. implementacja takiej aplikacji zmusza do wykorzystania zupełnie nowej architektury, wzorców lub kompilatora, co w większości przypadków oznacza poważne zmiany lub nawet tworzenie aplikacji na nowo,

2. proponowane rozwiązanie nie przewiduje dokonania optymalizacji współpracy w czasie wykonania aplikacji lub stosowana optymalizacja uwzględnia tylko jeden parametr, np. wydajność albo wykorzystanie energii, a nie oba parametry jednocześnie,

3. wdrożenie takiej aplikacji wymaga poważnych zmian w systemie operacyjnym urządzenia, co utrudnia implementację w praktycznych zastosowaniach bez wsparcia od producenta systemu operacyjnego.

W związku z powyższymi wnioskami wynikającymi z przeglądu istniejących rozwiązań zasadne jest opracowanie nowego wzorca umożliwiającego optymalizację współpracy aplikacji mobilnych zintegrowanych z chmurą obliczeniową. Oryginalność proponowanego rozwiązania polega na zastosowaniu dynamicznej optymalizacji, w której algorytm rozdziału uwzględnia zmieniające się parametry wykonania w czasie działania aplikacji. Ponadto, moduł zarządzający dostosowany jest do najnowszych sposobów implementacji aplikacji mobilnych, tzw. aplikacji hybrydowych, w których logika aplikacji implementowana jest w języku JavaScript, a instrukcje tłumaczone są na kod natywny dla danej platformy mobilnej. Dzięki temu aplikacje są niezależne od systemu operacyjnego i nie wymagają wprowadzania w nim żadnych zmian. Jednocześnie modyfikacje samej aplikacji są minimalne i pozwalają na podział aplikacji na chmurę obliczeniową i urządzenie mobilne z dokładnością do pojedynczego komponentu.

## 2.4 Tezy rozprawy

Rozpatrujemy aplikacje złożone z kilku komponentów, w których czasy wykonania przynajmniej części komponentów na chmurze obliczeniowej są znacznie krótsze niż na urządzeniu mobilnym. Ponadto czasy komunikacji między urządzeniem a chmurą obliczeniową muszą być mniejsze niż różnica czasów wykonania tych komponentów w chmurze i na urządzeniu. Co więcej rozpatrywane aplikacje charakteryzują się wysoką złożonością obliczeniową.

Dodatkowo przyjęto założenie, że zasoby chmury obliczeniowej są nieograniczone, a aplikacje korzystające równocześnie z jej zasobów nie wpływają na siebie nawzajem, inaczej są obliczeniowo niezależne. Biorąc pod uwagę aktualne możliwości chmur obliczeniowych, w tym wirtualizacji i izolacji procesów, założenie to jest uzasadnione.

Uwzględniając powyższe ograniczenia, zaproponowano szkielet aplikacji (framework) oraz moduł zarządzający umożliwiający optymalizację wykonania aplikacji mobilnych zintegrowanych z chmurą obliczeniową. Szkielet ten określa strukturę aplikacji oraz mechanizm jej działania, a moduł zarządzający dostarcza zestaw bibliotek do wykonania, monitorowania i rozdziału komponentów aplikacji. Jako komponenty rozumie się wszystkie funkcje obsługujące logikę aplikacji, a także różne usługi przeznaczone do obsługi żądań, np. odwołanie się do kontrolerów, baz danych, obsługa wyjątków, uwierzytelnienie użytkownika. W rozprawie przeprowadzono szereg badań, które pozwoliły wykazać następujące tezy:

1. Dokładny algorytm rozdziału aplikacji na urządzenie mobilne i chmurę obliczeniową przy optymalizacji z ograniczeniami (minimalizacja całkowitego kosztu wykonania i ograniczenie na koszty wykonania każdego z komponentów aplikacji) należy do problemów klasy złożoności NP.

2. Proponowany heurystyczny algorytm rozdziału aplikacji na urządzenie mobilne i chmurę obliczeniową posiada złożoność wielomianową, a w przypadku iteracyjnego przenoszenia pojedynczego komponentu pomiędzy urządzeniem a chmurą obliczeniową – złożoność liniową.

# Rozdział 3

# Statyczny model współpracy

Rozpatrzono współpracę urządzeń mobilnych z chmurą obliczeniową przy założeniu stałych, tzn. nie zmieniających się w czasie, parametrów wykonania aplikacji. Przyjęto wielowymiarowe kryterium optymalizacji takiej współpracy oraz zaproponowano algorytm określający jaka część aplikacji powinna być uruchomiona w chmurze, a jaka na urządzeniu mobilnym. Zaproponowano również framework zarządzający wykonaniem takiej aplikacji.

## 3.1 Model komponentowy aplikacji

Zaproponowano nowy statyczny model aplikacji mobilnych współpracujących z chmurą, który powinien być elastyczny i pozwalać na implementację dowolnego typu aplikacji. Programista powinien móc wykorzystać dowolne wzorce architektury (patrz rozdział 2.2.4) i nie być zmuszony do implementacji aplikacji tylko w jeden określony sposób. Ponadto proponowane rozwiązanie powinno być zintegrowane z istniejącym systemem operacyjnym urządzenia mobilnego oraz jego platformą wytwarzania.

Optymalizacja rozdziału aplikacji powinna być określona bezpośrednio przed uruchomieniem aplikacji i powinna uwzględniać parametry urządzenia, sieci i dostępność chmury obliczeniowej. Algorytm rozdziału powinien odnosić się do uogólnionego modelu aplikacji i uwzględniać koszt wykonania aplikacji oraz transferu danych pomiędzy urządzeniem i chmurą, a także ograniczenia na koszty częściowe, tzn. koszt części aplikacji wykonywanej w chmurze i części wykonywanej na urządzeniu.

Jak podano w rozdziale 2.3.3, wykorzystanie modelu w praktycznych zastosowaniach nie powinno wymagać żadnych zmian w systemie operacyjnym urządzenia. Wszystkie dostarczone narzędzia, kompilatory czy interfejsy muszą być zintegrowane

z aplikacją i systemem w nieinwazyjny sposób, w postaci plug-inów, rozszerzeń lub bibliotek, które bedą nadal działać w przypadku aktualizacji środowiska wykonania.

W celu zbudowania uniwersalnego modelu aplikacji mobilnej należy rozważyć różne typy przetwarzania danych w aplikacji. Można łatwo wyróżnić trzy podstawowe klasy:

1. Przetwarzanie sekwencyjne – przepływ danych w aplikacji jest sekwencyjny, wyjście z jednego komponentu jest przekazywanie na wejście dokładnie jednego innego komponentu. W danej chwili może być wykonywany tylko jeden komponent, pozostałe czekają w kolejce na zakończenie przetwarzania.

2. Przetwarzanie współbieżne – wyjście z jednego komponentu może być przekazane do wielu innych komponentów. Komponenty mogą przetwarzać dane w tym samym czasie, w efekcie realizując przetwarzanie równoległe.

3. Przetwarzanie rozproszone – komponenty aplikacji mogą być wykonywane na różnych maszynach. Przepływ danych może być zamodelowany jako dowolny acykliczny graf skierowany.

Ponieważ modele przetwarzania sekwencyjnego i współbieżnego są specjalnymi przypadkami modelu przetwarzania rozproszonego, ten ostatni może być wykorzystany jako uogólniony model aplikacji. W dalszej części rozdziału sprecyzowano model aplikacji opartej na przetwarzaniu rozproszonym, sposób funkcjonowania komponentów aplikacji i mechanizm przekazywania danych pomiędzy nimi.

### 3.1.1 Komponentowy model aplikacji

Każda nietrywialna aplikacja składa się z wielu komponentów przetwarzających dane. W ogólności, stosując model rozproszonego przetwarzania danych, mogą być one połączone w formie acyklicznego grafu skierowanego $A$, w którym wierzchołki $N$ oznaczają komponenty aplikacji, a krawędzie $E$ przepływ danych pomiędzy nimi.

$$A = (N, E) \tag{3.1}$$

$$E = \{(n_i, n_j) : n_i \in N \wedge n_j \in N \wedge n_i \neq n_j\} \tag{3.2}$$

W ramach opisywanego modelu można wyróżnić cztery typy danych związane z każdym komponentem:

- Dane wejściowe - dane, które są dostarczane do komponentu przez użytkownika lub pośrednio przez inny komponent.
- Dane wyjściowe - dane, które są rezultatem przetwarzania komponentu.
- Stan komponentu - dane wewnętrzne komponentu, nie są współdzielone z innymi komponentami.
- Dane współdzielone - dane dostępne w postaci usługi, która może być wywołana przez dowolny komponent, zarówno w chmurze, jak i na urządzeniu.

W opisywanym modelu podstawową funkcją komponentu jest przetwarzanie danych wejściowych na dane wyjściowe. Zarządzanie stanem i wywoływanie zewnętrznych usług to wewnętrzne operacje komponentu, które nie są istotne z punktu widzenia optymalizacji rozdziału aplikacji. Czas i koszt związany z dostępem do współdzielonych danych powinien być jednak uwzględniony w czasie i koszcie przetwarzania komponentu.

## 3.2 Problem optymalnego rozdziału aplikacji

Problem optymalnego rozdziału aplikacji w przedstawionym modelu polega na wskazaniu komponentów, które będą wykonywane w chmurze obliczeniowej i na urządzeniu mobilnym. Przy założeniu, że parametry wykonania komponentów nie zmieniają się w czasie działania aplikacji, optymalizacja może zostać wykonana jednorazowo, bezpośrednio przed uruchomieniem aplikacji. Do przeprowadzania tego typu optymalizacji niezbędne jest określenie parametrów wykonania komponentów i funkcji kosztu, która będzie minimalizowana.

Na komponentowym modelu aplikacji $A = (N, E)$ można zdefiniować rozdział $(N_c, N_m)$ komponentów na komponenty uruchomione w chmurze $N_c$ i komponenty uruchomione na urządzeniu $N_m$;

$$N = N_m \cup N_c \tag{3.3}$$

$$N_m \cap N_c = \emptyset \tag{3.4}$$

Przy danym podziale komponentów $N_m$ i $N_c$ możemy określić dwa podzbiory połączeń pomiędzy komponentami:

- Połączenia wewnętrzne $E_{int}$ - połączenia pomiędzy komponentami alokowa-

nymi w tym samym środowisku, tzn. oba komponenty na urządzeniu mobilnym lub oba komponenty w chmurze,

- Połączenia zewnętrzne $E_{ext}$ - połączenia pomiędzy komponentami alokowanymi w różnych środowiskach, tzn. jeden z komponentów na urządzeniu mobilnym, a drugi w chmurze.

W pewnych przypadkach komponent może być ograniczony do wykonania jedynie na urządzeniu mobilnym, np. ze względu na konieczność dostępu do specyficznych urządzeń wejścia/wyjścia lub ściśle określonych funkcji systemu operacyjnego. Jest również możliwe, że niektóre komponenty muszą być wykonane w chmurze obliczeniowej, np. ze względu na konieczność synchronizacji danych z wieloma użytkownikami aplikacji. Zakłada się, że tego typu komponenty są na stałe przypisane do środowiska wykonania i nie są brane pod uwagę przy optymalizacji rozdziału.

## 3.3 Parametry wykonania komponentów

Dla każdego z komponentów można empirycznie wyznaczyć następujące mierzalne parametry:

- Czas wykonania - czas potrzebny do przetworzenia danych wejściowych na dane wyjściowe, liczony jako czas wykorzystania pojedynczego rdzenia procesora, tzn. wykorzystanie dwóch rdzeni przez 1s jest równoważne z wykorzystaniem jednego rdzenia przez 2s. Ponieważ dany komponent może przetwarzać dane z różną wydajnością w zależności od środowiska, wyróżniamy dwie funkcje czasu:
  - $t_m(n)$ - czas wykonania komponentu $n$ na urządzeniu mobilnym ($m$),
  - $t_c(n)$ - czas wykonania komponentu $n$ w chmurze obliczeniowej ($c$).
- Rozmiar danych wejściowych $s_{in}(n)$ - rozmiar danych przesłanych do komponentu $n$ liczony w bajtach.
- Rozmiar danych wyjściowych $s_{out}(n)$ - rozmiar danych wysłanych z komponentu $n$ liczony w bajtach.

Pozostałe parametry wykonania, takie jak wykorzystanie pamięci RAM czy dostęp do pamięci masowej, nie są istotne z punktu widzenia przyjętej optymalizacji.

Różnica w wydajności tych komponentów odzwierciedlona jest w czasie wykorzystania procesora, a wykorzystanie energii w typowych aplikacjach użytkowych jest pomijalne w stosunku do wykorzystania CPU i transferu danych przez sieć [12].

### 3.3.1 Funkcja kosztu

Koszt $k_x(n)$ wykonania pojedynczego komponentu $n$, gdzie $x = c$ dla chmury lub $x = m$ dla urządzenia mobilnego, jest proporcjonalny do czasu wykorzystania procesora w chmurze $t_c(n)$ lub na urządzeniu $t_m(n)$:

$$k_m(n) = P_m * t_m(n) \tag{3.5}$$

$$k_c(n) = P_c * t_c(n) \tag{3.6}$$

Parametry $P_m$ oraz $P_c$ to odpowiednio współczynniki poboru energii (mocy) w chmurze i na urządzeniu mobilnym. W ramach przyjętej optymalizacji aplikacji są to wartości stałe i równe dla wszystkich komponentów.

Zazwyczaj pobór energii elektrycznej danego urządzenia jest proporcjonalny do jego mocy obliczeniowej, natomiast czas wykonania obliczeń jest odwrotnie proporcjonalny. Dokładne wartości zależą oczywiście od charakterystyki konkretnych podzespołów, głównie procesora, ale także np. akceleratorów graficznych, jeśli są wykorzystywane. W związku z tym koszty wykonania komponentu na chmurze i urządzeniu mobilnym są w ogólności trudne do przewidzenia i mogą być zupełnie różne dla różnych urządzeń i chmur obliczeniowych. Jedynym praktycznym sposobem określenia parametrów wykonania jest doświadczalne zmierzenie czasu i rozmiaru danych. W rozdziale 5 został szczegółowo opisany moduł zarządzający, którego zadaniem jest m.in. mierzenie parametrów poszczególnych komponentów. W niniejszym rozdziale przyjęto założenie, że parametry wykonania komponentów są znane.

W celu obliczenia całkowitego kosztu wykonania należy również uwzględnić koszt komunikacji. Ponieważ każdy komponent może wysyłać i odbierać dane do i od wielu innych komponentów, oznaczmy rozmiar danych przesyłanych pomiędzy komponentami $n_i$ i $n_j$ jako $z(n_i, n_j)$, który pozostaje w następującej zależności z danymi wejściowymi i wyjściowymi:

$$z_{out}(n_i) \geq z(n_i, n_j) \leq z_{in}(n_j) \tag{3.7}$$

Na podstawie rozmiaru danych przesyłanych pomiędzy komponentami i przepustowości sieci $B$ wyrażonej w bajtach na sekundę, można wyznaczyć czas transferu $t_e$ dla połączenia $e = \{n_i, n_j\}$ pomiędzy komponentami $n_i$ i $n_j$:

$$t_e(e) = \frac{z(n_i, n_j)}{B} \tag{3.8}$$

Koszt transferu, analogicznie jak w przypadku kosztu wykonania komponentu, jest proporcjonalny do czasu i poboru energii podzespołów $P_e$ wykorzystywanych przy transmisji:

$$k_e(e) = P_e * \frac{z(n_i, n_j)}{B} = P_e * t_e(e) \tag{3.9}$$

Dla uproszczenia można założyć, że czas i koszt transferu pomiędzy dwoma komponentami wykonywanymi w tym samym środowisku jest pomijany i równy zero:

$$E_{int} : \{e = \{n_i, n_j\} : n_i \in N_c \wedge n_j \in N_c\} \tag{3.10}$$

$$E_{ext} = E \setminus E_{int} \tag{3.11}$$

$$\forall e \in E_{int} : t_e(e) = 0 \tag{3.12}$$

$$\forall e \in E_{int} : k_e(e) = 0 \tag{3.13}$$

W statycznym modelu współpracy współczynnik $P_e$ jest przez cały czas działania aplikacji i równy dla wszystkich połączeń. Dla uproszczenia obliczenia kosztu przyjęto, że dotyczy on wyłącznie energii wykorzystywanej przez urządzenie mobilne, tzn. moduły WiFi oraz GSM.

W oparciu o model aplikacji $A = (N, E)$ oraz opisane parametry komponentów można zdefiniować model kosztu $A_K$:

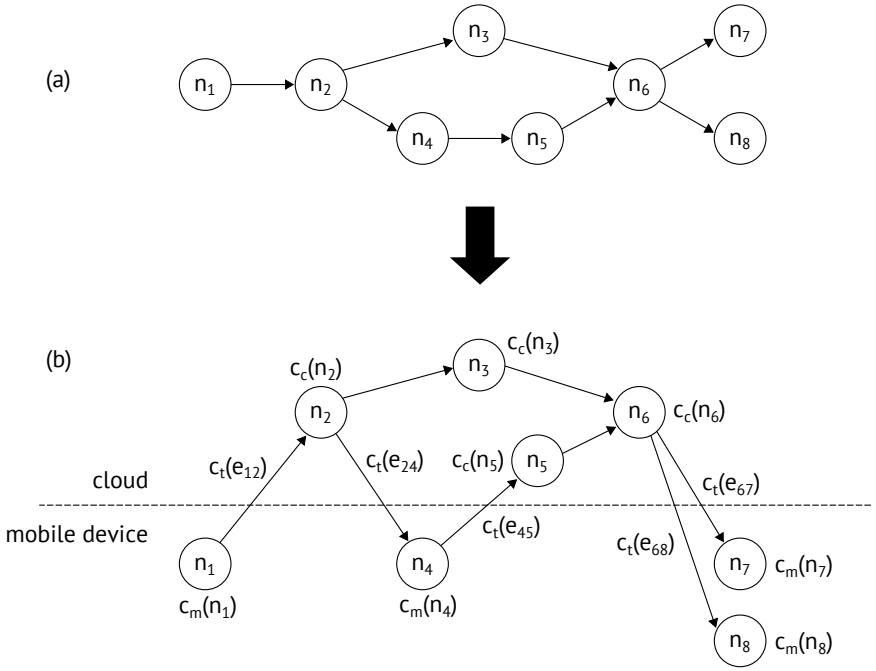
$$A_K = (N, E, P) \tag{3.14}$$

$$P = (k_m, k_c, k_e) \tag{3.15}$$

$$k_m : N \to \mathbb{R}, k_c : N \to \mathbb{R}, k_e : E \to \mathbb{R} \tag{3.16}$$

Model kosztu rozszerza wcześniej opisany komponentowy model aplikacji o funkcje kosztu: dwie funkcje nad zbiorem komponentów i jedną funkcję nad zbiorem krawędzi:

- $k_m(n)$ – koszt wykonania komponentu $n$ na urządzeniu mobilnym,




Rysunek 3.1: Przykład modelu aplikacji (a) i modelu kosztu (b).

- $k_c(n)$ – koszt wykonania komponentu $n$ w chmurze obliczeniowej,
- $k_e(e)$ – kosz transferu krawędzi $e = (n_i, n_j)$, od komponentu $n_i$ do komponentu $n_j$.

Na rysunku 3.1 przedstawiony został przykładowy model aplikacji i odpowiadający mu model kosztu przetwarzania dla podziału $N_c = \{n_2, n_3, n_5, n_6\}$ i $N_m = \{n_1, n_4, n_7, n_8\}$.

Dla danego podziału komponentów $(N_m, N_c)$ można zdefiniować całkowity koszt $K$ wykonania aplikacji:

$$K(N_c, N_m) = \sum_{n \in N_c} k_c(n) + \sum_{n \in N_m} k_m(n) + \sum_{e \in E_{ext}} k_e(e) \tag{3.17}$$

Ponieważ parametry $P_c$, $P_m$ i $P_e$ są równe dla wszystkich komponentów można wyłączyć je z sumy:

$$K(N_c, N_m) = P_c \sum_{n \in N_c} t_c(n) + P_m \sum_{n \in N_m} t_m(n) + P_e \sum_{e \in E_{ext}} t_e(e) \tag{3.18}$$

Tak opisana funkcja kosztu określa całkowitą energię potrzebną do wykonania aplikacji. Warto zauważyć, że funkcję kosztu można przekształcić do uogólnionej

Tabela 3.1: Różne wersje uogólnionej funkcji kosztu

| | $\alpha$ | $\beta$ | $\gamma$ |
|---|---|---|---|
| Całkowite wykorzystanie energii | $> 0$ | $> 0$ | $> 0$ |
| Wykorzystanie energii urządzenia | $> 0$ | $= 0$ | $> 0$ |
| Wykorzystanie energii chmury | $= 0$ | $> 0$ | $= 0$ |
| Koszt pieniężny | $= 0$ | $> 0$ | $> 0$ |
| Koszt transferu danych | $= 0$ | $= 0$ | $> 0$ |

funkcji kosztu $K_g$ przez zamianę współczynników opisujących wykorzystanie energii chmury i urządzenia, na abstrakcyjne współczynniki $\alpha$, $\beta$ i $\gamma$:

$$K_g(N_c, N_m) = \alpha \sum_{n \in N_c} t_c(n) + \beta \sum_{n \in N_m} t_m(n) + \gamma \sum_{e \in E_{ext}} t_e(e) \tag{3.19}$$

Tak zdefiniowana funkcja kosztu pozwala na dopasowanie modelu kosztu do różnych celów optymalizacji [56]. Tabela 3.1 przedstawia przykłady różnych celów optymalizacji, które mogą być osiągnięte przez przypisanie odpowiednich wartości współczynników. Przykładowo, możliwe jest zamodelowanie kosztu pieniężnego, gdzie rozważany jest koszt transferu danych w sieci GSM i kosztów obsługi infrastruktury chmury obliczeniowej. Dokładne wartości współczynników zależeć będą od konkretnych cen wymienionych zasobów. W ramach rozprawy doktorskiej rozważany będzie model kosztu z funkcją kosztu określającą całkowite wykorzystanie energii, ponieważ ma on największe zastosowanie z praktycznego punktu widzenia, jak wynika z rozdziału 2.1.

W ramach funkcji kosztu $K$ można wyróżnić trzy składniki, które same w sobie również mają zastosowanie w optymalizacji: koszt komponentów uruchomionych na chmurze $K_c$, koszt komponentów uruchomionych na urządzeniu $K_m$ oraz koszt komunikacji $K_e$:

$$K_c(N_c) = P_c \sum_{n \in N_c} t_c(n) \tag{3.20}$$

$$K_m(N_m) = P_m \sum_{n \in N_m} t_m(n) \tag{3.21}$$

$$K_e(N_c, N_m) = P_t \sum_{e \in E_{ext}} t_e(e) \tag{3.22}$$

$$K(N_c, N_m) = K_c(N_c) + K_m(N_m) + K_e(N_c, N_m) \tag{3.23}$$

Funkcje częściowego kosztu $K_c$, $K_m$ i $K_e$ mogą być wykorzystane jako dodatkowe kryteria w optymalizacji aplikacji. Typowym problemem w aplikacjach mobilnych jest wykorzystanie energii urządzenia, a więc koszt częściowy urządzenia mobilnego $K_m$ powinien zostać zminimalizowany lub nie przekraczać pewnego określonego progu $c_{max}$. Koszt wykorzystania chmury obliczeniowej $K_c$ również może być istotnym parametrem, szczególnie jeśli rozważymy wiele klientów mobilnych korzystających z tej samej chmury. Mając na uwadze zastosowania praktyczne, częściowe koszty $K_m$ oraz $K_c$ będą dodatkowymi kryteriami optymalizacji rozważanymi w niniejszej rozprawie.

## 3.4 Kryteria i algorytmy rozdziału aplikacji

Optymalizacja polega na podziale zbioru komponentów $N$ na podzbiory komponentów, które powinny zostać wykonane w chmurze obliczeniowej $N_c$ i na urządzeniu $N_m$. Ponieważ podzbiór $N_m$ jest dopełnieniem zbioru $N_c$, można powiedzieć, że celem optymalizacji jest znalezienie takiego podzbioru $N_c$ z wszystkich możliwych podzbiorów komponentów $\mathbb{P}(N)$, że funkcja kosztu $C(N_c, N \setminus N_c)$ jest minimalna i jednocześnie koszt częściowy urządzenia mobilnego $K_m$ nie przekracza założonego kosztu maksymalnego $m_{max}$, a koszt częściowy $K_c$ chmury obliczeniowej nie przekracza założonego kosztu maksymalnego $c_{max}$:

$$\begin{aligned} \min_{N_c \in \mathbb{P}(N)} \quad & K(N_c, N \setminus N_c) \\ & K_m(N \setminus N_c) \leqslant m_{max} \\ & K_c(N_c) \leqslant c_{max} \end{aligned} \tag{3.24}$$

## 3.5 Algorytm rozdziału komponentów

Algorytm optymalizacji powinien wyznaczyć optymalny rozdział komponentów pomiędzy urządzenie mobilne i chmurę obliczeniową dla podanego na wejściu modelu kosztu. Można wyróżnić specjalny przypadek problemu, w którym dopuszczalne są dowolne koszty częściowe na urządzeniu mobilnym i chmurze obliczeniowej.

### 3.5.1 Minimalizacja kosztu

Przy założeniu, że $c_{max} = \infty$ oraz $m_{max} = \infty$ otrzymujemy uproszczony, jednokryterialny problem optymalizacji. Dla danego grafu $A = (N, E)$ problem rozdziału

można zredukować do problemu minimalnego rozcięcia grafu [35], konstruując graf $G = (N', E', w)$, gdzie $N \in N'$, $E \in E'$, a $w : N \to \mathbb{R}$, w następujący sposób:

1. Należy utworzyć zbiór wierzchołków $N'$ przez dodanie wierzchołki $n_c$ i $n_m$ do zbioru wierzchołków $N$.
2. Dla każdego wierzchołka $n_i$ w $N$ należy utworzyć krawędź $e = (n_i, n_c) \in E'$ o wadze $w(e) = k_c(n_i)$.
3. Dla każdego wierzchołka $n_i$ w $N$ należy utworzyć krawędź $e = (n_i, n_m) \in E'$ o wadze $w(e) = k_m(n_i)$.
4. Dla każdej krawędzi $(n_i, n_j) \in E$ należy utworzyć krawędź $e = (n_i, n_j) \in E'$ o wadze $w(e) = k_e(n_i, n_j)$.

W tak skonstruowanym grafie $G$ należy znaleźć minimalne rozcięcie, takie że wierzchołki $n_c$ i $n_m$ znajdują się po przeciwnych stronach rozcięcia. Suma wag krawędzi rozcinających jest równa funkcji kosztu $K$. Problem ten należy do klasy $P$ i może być rozwiązany w czasie wielomianowym [94]:

$$O(|V||E| + |V|^2 \log |V|) \tag{3.25}$$

Pseudokod algorytmu został przedstawiony na poniższym listingu.

**function** MinimumCutPhase(G,n,w)
  $A \leftarrow \{n\}$
  **while** $A \neq N$ **do**
    *add to A the most tightly connected vertex*
  **end while**
  *store the cut of the phase and shrink G by merging the two vertices added last*
**end function**

**function** MinimumCut(G,N,w)
  $n \in N$
  **while** $|N| > 1$ **do**
    MinimumCutPhase(G,w,n)
    **if** *the cut of the phase is lighter than the current minimum cut* **then**
      *store the cut of the phase as the current minimum cut*

**end if**
**end while**
**end function**

### 3.5.2 Minimalizacja kosztu z ograniczeniami

W ogólności opisany problem optymalizacji z uwzględnieniem kosztu całkowitego $K$ oraz częściowych kosztów $K_m$ i $K_c$ należy do klasy problemów NP. Dowód polega na redukcji problemu minimalnej bisekcji grafu na równe części, zdefiniowanego w następujący sposób:

Niech $G(N, E)$ będzie grafem, gdzie $N$ to zbiór jego wierzchołków, a $E$ zbiór krawędzi. Należy znaleźć podział $N$ na dwa rozłączne, jednakowo liczne podzbiory $A$ i $B$ tak, by liczba krawędzi między wierzchołkami z podzbioru $A$ i $B$, oznaczona przez $T$, była jak najmniejsza.

Redukcję do problemu optymalizacji kosztu aplikacji z ograniczeniami przeprowadzamy przyjmując, że podzbiór $A = N_c$, a podzbiór $B = N_m$. Załóżmy następujące wartości dla czasów wykonania i transferu:

$$\forall n \in N : k_c(n) = 1 \tag{3.26}$$

$$\forall n \in N : k_m(n) = 1 \tag{3.27}$$

$$\forall e \in E : k_e(e) = 1 \tag{3.28}$$

Przyjmijmy również następujące ograniczenia dla kosztów częściowych:

$$c_{max} = \frac{|N|}{2} \tag{3.29}$$

$$m_{max} = \frac{|N|}{2} \tag{3.30}$$

Łatwo zauważyć, że gdy koszty wykonania danego komponentu są równe na urządzeniu mobilnym i w chmurze to bez względu na podział aplikacji suma kosztów $K_c$ i $K_m$ jest taka sama i równa $|K|$, a więc koszt całkowity można wyrazić jako:

$$K(N_c, N_m) = \sum_{n \in N_c} k_c(n) + \sum_{n \in N_m} k_m(n) + \sum_{e \in E_{ext}} k_e(e) = |V| + \sum_{e \in E_{ext}} k_e(e) = |V| + K_t \tag{3.31}$$

Ponieważ założyliśmy jednostkowy koszt transferu dla każdej krawędzi to koszt częściowy $K_t$ jest równy liczbie krawędzi między wierzchołkami z podzbioru $A$ i $B$, a więc:

$$\min_{A,B} T = \min_{N_c,N_m} K(N_c, N_m) - |N| \tag{3.32}$$

Z ograniczeń nałożonych na koszty częściowe $K_c$ i $K_m$ wynika, że żaden z nich nie może być większy niż $\frac{|N|}{2}$ co przy jednostkowych kosztach dla każdego komponentu oznacza, że podzbiory $N_c$ oraz $N_m$, a więc i $A$ oraz $B$ muszą być równoliczne. Rozwiązanie problemu minimalnej bisekcji grafu na równe części wymaga więc rozwiązania problemu optymalizacji kosztu aplikacji z ograniczeniami, co dowodzi że przedstawiony problem optymalizacji jest NP-trudny, chyba że P=NP. Tym samym dowodzi to pierwszej tezy rozprawy.

### 3.5.3 Algorytm dokładny

W celu znalezienia rozdziału zbioru komponentów $N$ aplikacji, tak aby koszt wykonania $K$ był możliwie najmniejszy przy ograniczeniu kosztów częściowych $K_c$ i $K_m$, można sprawdzić wszystkie możliwe podziały zbioru komponentów na dwa podzbiory $N_c$ i $N_m$. Liczba wszystkich możliwości jest równa liczności zbioru potęgowego $|\mathcal{P}(N)| = 2^{|N|}$, a złożoność algorytmu dokładnego (Dokładny Rozdział Komponentów, DRK) przedstawionego na poniższym listingu wynosi $O(n2^n)$.

**function** DokładnyRozdziałKomponentów($N, E, P, c_{max}, m_{max}$)
  $K_{min} \leftarrow \inf$, $N_c^{opt} \leftarrow \emptyset$
  **for all** $N_c \in \mathcal{P}(N)$ **do**
    $K_c \leftarrow 0$, $K_m \leftarrow 0$, $K_e \leftarrow 0$
    **for all** $n \in N_c$ **do**
      $K_c \leftarrow K_c + k_c(n)$
    **end for**
    **for all** $n \in N \setminus N_c$ **do**
      $K_m \leftarrow K_m + k_m(n)$
    **end for**
    **for all** $e \in E_{ext}$ **do**
      $K_e \leftarrow K_e + k_e(e)$
    **end for**

```
        K ← K_c + K_m + K_e
        if K_c ≤ c_max and K_m ≤ m_max and K < K_min then
            K_min ← K
            N_c^opt ← N_c
        end if
    end for
    return (N_c^opt, N \ N_c^opt)
end function
```

### 3.5.4 Algorytm heurystyczny

Model aplikacji jest z reguły grafem rzadkim, w którym liczba krawędzi jest mniejsza niż kwadrat liczby wierzchołków. W problemach optymalizacyjnych dotyczących takich grafów dobrze sprawdzają się algorytmy heurystyczne, np. algorytm genetyczny [71]. W literaturze można znaleźć przykłady dobrych wyników wykorzystania takich algorytmów do rozwiązywania problemów podziału grafu [37] [49].

Algorytm genetyczny polega na generowaniu kolejnych populacji osobników w iteracyjny sposób. Każda iteracja składa się z kilku etapów: losowania, selekcji, krzyżowania i mutacji. W przypadku optymalizacji aplikacji w modelu komponentowym, pojedynczym osobnikiem będzie pewien podzbiór komponentów przeznaczony do uruchomienia w chmurze. Podzbiór komponentów uruchomionych na urządzeniu będzie zawsze dopełnieniem do zbioru wszystkich komponentów.

W celu opisania heurystycznego algorytmu optymalizacji zdefiniujmy najpierw funkcję generującą losowy podzbiór komponentów, które zostaną uruchomione w chmurze:

```
function LosowyPodzbiór(N)
    N_c ← ∅
    for all n ∈ N do
        r ← random from 0 to 1
        if r < 0.5 then
            add n to N_c
        end if
    end for
```

**return** $N_C$
**end function**

Kolejnym etapem jest selekcja najlepszych podzbiorów z populacji, tzn. takich dla których koszt $K$ wykonania aplikacji jest najmniejszy i jednocześnie koszty częściowe $K_c$ i $K_m$ nie przekraczają wartości maksymalnych $c_{max}$ i $m_{max}$. W tym celu zastosowany zostanie algorytm selekcji turniejowej, który zwraca najlepszy podzbiór komponentów spośród pewnej próbki o rozmiarze $s_{size}$. Zasada działania jest analogiczna do algorytmu dokładnego, ale wybór zbioru ogranicza się jedynie do zdefiniowanej próbki:

**function** SelekcjaTurniejowa($Population, N, E, P, c_{max}, m_{max}$)
  $K_{min} \leftarrow \inf$, $N_c^{best} \leftarrow \emptyset$
  **for** $i$ to $s_{size}$ **do**
    $N_c \leftarrow$ *random from Population*
    $K_c \leftarrow 0$, $K_m \leftarrow 0$, $K_e \leftarrow 0$
    **for all** $n \in N_c$ **do**
      $K_c \leftarrow K_c + k_c(n)$
    **end for**
    **for all** $n \in N \setminus N_c$ **do**
      $K_m \leftarrow K_m + k_m(n)$
    **end for**
    **for all** $e \in E_{ext}$ **do**
      $K_e \leftarrow K_e + k_e(e)$
    **end for**
    $K \leftarrow K_c + K_m + K_e$
    **if** $K_c \leq c_{max}$ **and** $K_m \leq m_{max}$ **and** $K < K_{min}$ **then**
      $K_{min} \leftarrow K$
      $N_c^{best} \leftarrow N_c$
    **end if**
  **end for**
  **return** $N_c^{best}$
**end function**

Proces krzyżowania polega na utworzeniu nowego podzbioru komponentów z

dwóch wcześniej wybranych. Odbywa się to przez losowe dodawanie komponentu z jednego lub drugiego podzbioru:

**function** Krzyżowanie($N_c^A, N_c^B, N$)
  $N_c \leftarrow \emptyset$
  **for all** $n \in N$ **do**
    $r \leftarrow$ *random from 0 to 1*
    **if** $r < 0.5$ **then**
      **if** $n \in N_c^A$ **then**
        *add* $n$ *to* $N_c$
      **end if**
    **else**
      **if** $n \in N_c^B$ **then**
        *add* $n$ *to* $N_c$
      **end if**
    **end if**
  **end for**
  **return** $N_c$
**end function**

Ostatnim etapem jest mutacja, która w przypadku zbioru komponentów polegać będzie na przeniesieniu jednego losowo wybranego komponentu pomiędzy zbiorem $N_c$ i $N_m$:

**function** Mutacja($N_c, N$)
  $n \leftarrow$ *random from* $N$
  **if** $n \in N_c$ **then**
    *remove* $n$ *from* $N_c$
  **else**
    *add* $n$ *to* $N_c$
  **end if**
  **return** $N_c$
**end function**

Proponowany algorytm genetyczny wykonuje *it* iteracji, w każdej z nich genero-

wana jest populacja składająca się z $s$ podzbiorów komponentów. Każdy podzbiór jest przechodzi proces mutacji z prawdopodobieństwem $mut$. Pseudokod heurystycznego algorytmu optymalizacji (Genetyczny Rozdział Komponentów, GRK) rozdziału komponentów został przedstawiony na listingu:

**function** GentycznyRozdziałKomponentów($N, E, P, c_{max}, m_{max}, it, s, mut$)
  $Population \leftarrow \emptyset$
  **for** $i = 1$ to $s$ **do**
    $Population \leftarrow$ LosowyPodzbiór($N$)
  **end for**
  **for** $i = 1$ to $it$ **do**
    $NewPopulation \leftarrow \emptyset$
    **for** $j = 1$ to $s$ **do**
      $N_c^A \leftarrow$ SelekcjaTurniejowa($Population, N, E, P, c_{max}, m_{max}$)
      $N_c^B \leftarrow$ SelekcjaTurniejowa($Population, N, E, P, c_{max}, m_{max}$)
      $N_c \leftarrow$ Krzyżowanie($N_c^A, N_c^B, N$)
      $r \leftarrow$ *random from 0 to 1*
      **if** $r < mut$ **then**
        Mutacja($N_c, N$)
      **end if**
      *add* $N_c$ *to NewPopulation*
    **end for**
    $Population \leftarrow NewPopulation$
  **end for**
  **return** $N_c$ *from Population where* $K(N_c, N \setminus N_c)$ *is minimum*
    **and** $K_c(N_c) \leq c_{max}$ **and** $K_m(N \setminus N_c) \leq m_{max}$
**end function**

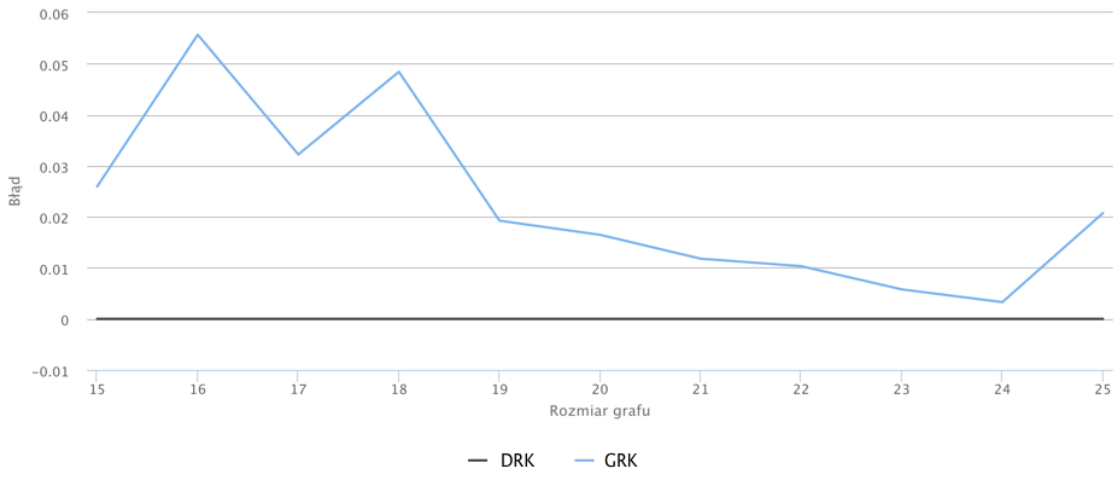
## 3.6 Porównanie jakości algorytmu heurystycznego

Zaproponowany heurystyczny algorytm genetyczny GRK został przetestowany na losowo generowanych grafach o różnym rozmiarze i o różnej gęstości. Przebadano grafy o rozmiarach $|N|$ od 15 do 25 wierzchołków, które reprezentują aplikacje o średniej złożoności (od 15 do 25 komponentów). Gęstość grafu, wyrażona jako stosunek

maksymalnego stopnia wierzchołka do rozmiaru grafu wynosiła od 0.1 (maksymalnie $\frac{|N|}{10}$ krawędzi w wierzchołku) do 1 (maksymalnie $|N|$ krawędzi w wierzchołku). Wyniki dla każdej kombinacji rozmiaru i gęstości grafu zostały uśrednione z 20 wygenerowanych grafów, tzn. grafów z losowo połączonymi wierzchołkami i o losowych wagach wierzchołków i krawędzi.

Algorytm heurystyczny GRK został porównany do algorytmu dokładnego DRK pod względem błędu optymalizacji i czasu wykonania. Błąd optymalizacji wyrażony jest jako różnica pomiędzy kosztem podziału suboptymalnego i optymalnego podzielona przez koszt optymalny. Przyjęte parametry algorytmu genetycznego to liczba iteracji $it = 4|N|$, rozmiar populacji $s = 4|N|$ i szansa na mutację równa $mut = 0.05$.

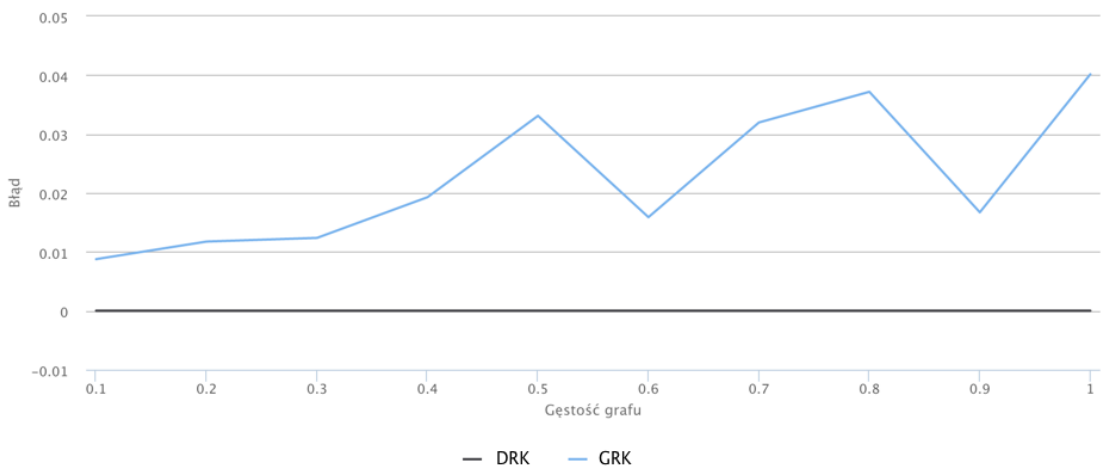
Na rysunku 3.2 przedstawiony został wykres błędu w funkcji rozmiaru grafu. Jak widać zaproponowany algorytm genetyczny daje w każdym wypadku błąd optymalizacji poniżej 6% i zmniejsza się on wraz ze wzrostem rozmiaru grafu.



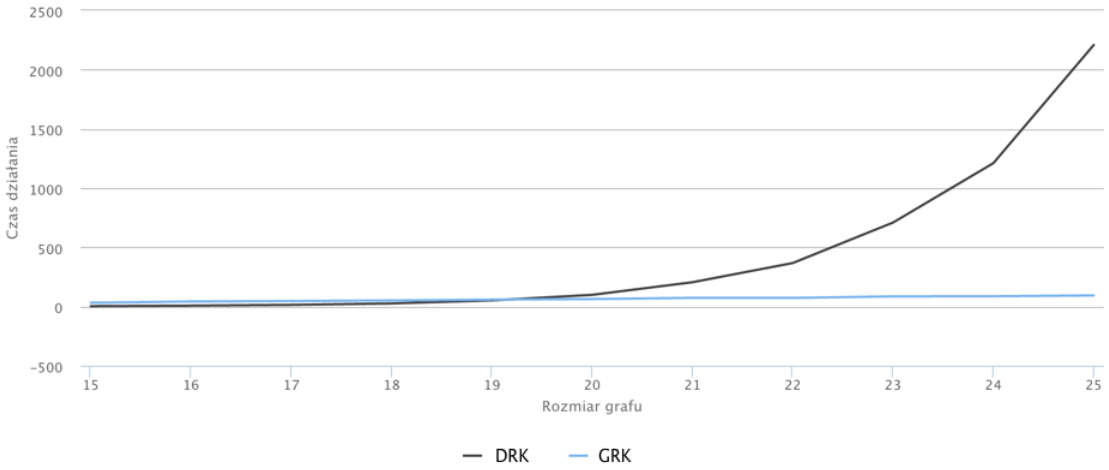

Rysunek 3.2: Błąd optymalizacji dla różnych rozmiarów grafu

Rysunek 3.3 przedstawia wykres błędu optymalizacji dla grafów o różnej gęstości. Zgodnie z przewidywaniami, można zauważyć tutaj odwrotną zależność – dla gęstych grafów błąd optymalizacji jest większy. Mimo wszystko nie przekracza on 4% kosztu optymalnego.

Porównanie czasów wykonania dokładnego algorytmu optymalizacji z algorytmem genetycznym zostało przedstawione na wykresie 3.4. Powyżej 19 wierzchołków czas wykonania algorytmu dokładnego zauważalnie rośnie, a przy 23 wierzchołkach jest już o rząd wielkości wyższy niż czas algorytmu heurystycznego.




Rysunek 3.3: Błąd optymalizacji dla różnej gęstości grafu




Rysunek 3.4: Czas optymalizacji dla różnych rozmiarów grafu

Przedstawione badania teoretycznie potwierdzają tezę, że zaproponowany genetyczny algorytm heurystyczny z powodzeniem może być stosowany do statycznej optymalizacji rozdziału aplikacji nawet o stosunkowo dużej złożoności (powyżej 20 komponentów) i gęstości (średni stopień wierzchołka ponad 10).

# Rozdział 4

# Dynamiczny model współpracy

Rozważono model aplikacji interaktywnych, których parametry zmieniają się w czasie w sposób niedeterministyczny m.in. na skutek interakcji użytkownika. Zaproponowano dynamiczną optymalizację rozdziału wykonywaną podczas całego okresu działania aplikacji. Przyjęto, że okres ten składa się z kolejno wykonywanych iteracji. Podobnie jak w modelu statycznym założono kryterium optymalizacji kosztu z ograniczeniami kosztów częściowych oraz zaproponowano iteracyjny algorytm heurystyczny określający, który komponent aplikacji w danej iteracji powinien zostać przeniesiony do chmury obliczeniowej.

## 4.1 Model aplikacji interaktywnej

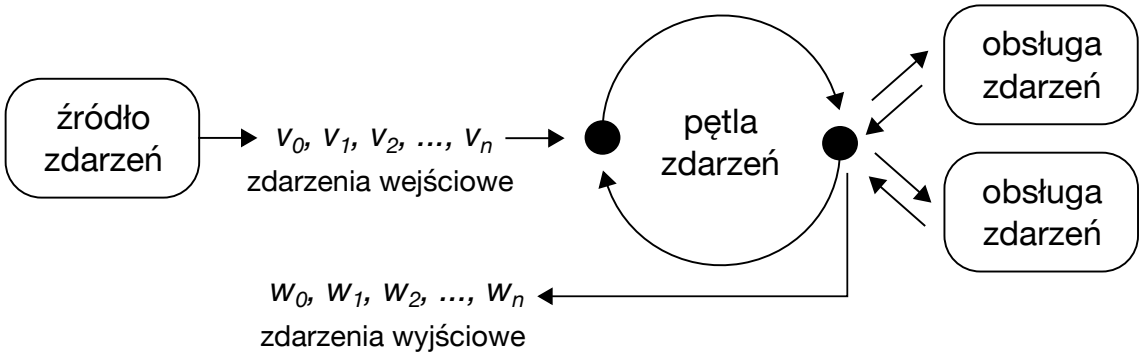
Aplikacje dla urządzeń mobilnych to w ogromnej większości aplikacje z graficznym interfejsem użytkownika poprzez który może on oddziaływać na przebieg działania aplikacji. Są to więc aplikacje interaktywne, takie jak gry, serwisy społecznościowe czy edytory dokumentów. Użytkownik wykonuje akcje w ciągu całego czasu życia programu, wprowadza dane i zmienia parametry wewnętrzne na ogół w niedeterministyczny sposób. Optymalizacja tego typu aplikacji jest znacznie większym wyzwaniem niż aplikacji statycznych, gdzie z góry znane są wszystkie dane wejściowe, tym samym też ścieżka wykonania. W aplikacjach interaktywnych przepływ sterowania z reguły nie jest możliwy do przewidzenia, gdyż zależy od bieżącej decyzji użytkownika.

Aplikacje interaktywne z graficznym interfejsem użytkownika wykorzystują asynchroniczne zdarzenia, takie jak kliknięcie myszą czy naciśnięcie klawisza przez użytkownika, ale również różne zdarzenia czasowe i komunikaty sieciowe. W przypadku

urządzeń mobilnych są to oprócz dotknięć ekranu, również zmiana orientacji urządzenia czy inne zdarzenia sygnalizowane przez pozostałe sensory wbudowane w to urządzenie.

W odpowiedzi na tego typu zdarzenia aplikacja przedstawia użytkownikowi wyniki przetwarzania danych prezentowane w sposób graficzny. Obsługa zdarzeń i renderowanie interfejsu są zazwyczaj realizowane za pomocą kolejki i pętli zdarzeń (ang. *event loop*), która cyklicznie pobiera i przetwarza zdarzenia $v_i$. We współczesnych 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 $w_i$ powodujące faktycznie renderowanie odpowiedzi na interfejsie użytkownika. Schemat działania pętli zdarzeń został przedstawiony na rys 4.1.




Rysunek 4.1: Pętla zdarzeń

W dalszych rozważaniach uwzględniony 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 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 [74]. 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 [20], a także biblioteki i frameworki dla popularnych języków takich jak C# (Reactive Extensions [61]) oraz JavaScript (RxJS [97] oraz Cycle.js [93]). Wprowadzają one warstwę abstrakcji do obsługi asynchronicznych zdarzeń i nowe konstrukcje programistyczne: strumienie oraz operatory służące do przetwarzania strumieni. Język JavaScript został wykorzystany przy modelowaniu aplikacji w rozdziale 5.

### 4.1.1 Strumienie zdarzeń

Asynchroniczne zdarzenia można modelować jako strumienie, czyli sekwencje zdarzeń następujących w czasie [25]. Strumień zdarzeń $s$ można interpretować jako ciąg wartości $v_i$. Wartość $v_i$ może być dowolna, ciąg zdarzeń może być nieskończony, jak również pusty.

$$s = \langle v_i, v_{i+1}, v_{i+2}, \dots \rangle \tag{4.1}$$

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.:

$$\begin{aligned} s_{touch} = \langle (307, 204), (521, 149), (122, 501), \dots \rangle \\ x \in [0, x_{max}], y \in [0, y_{max}] \end{aligned} \tag{4.2}$$

Innymi przykładami strumieni wejściowych zdarzeń są sygnały generowane przez 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 wartościami zdarzenia są identyfikatory elementów:

$$s_{action} = \langle (home\_button), (back\_button), (text\_input), \dots \rangle \tag{4.3}$$

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. Można go opisać za pomocą sekwencji macierzy, gdzie wiersze i kolumny odpowiadają współrzędnym ekranu urządzenia.

$$s_{frames} = \langle \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_i , \begin{bmatrix} p_{1,1} & \cdots & p_{1,w} \\ \vdots & \ddots & \vdots \\ p_{h,1} & \cdots & p_{h,w} \end{bmatrix}_{i+1} , \dots \rangle \tag{4.4}$$

Istotną cechą wszystkich strumieni zdarzeń jest to, że są one niemutowalne: wartość $v_i$ w strumieniu jest stała przez cały cykl życia aplikacji, tzn. od momentu jej uruchomienia do zatrzymania procesu.

### 4.1.2 Operatory

Strumienie wejściowe $s_{in}$ przetwarzane są na strumienie wyjściowe $s_{out}$ za pomocą funkcji $f(s)$, zwanych operatorami. W ogólności można zamodelować całą aplikację interaktywną jako jeden operator przetwarzający strumień wejściowy $s_{in}$ na strumień wyjściowy $s_{out}$:

$$s_{out} = f(s_{in}) \tag{4.5}$$

W praktyce w aplikacji wyróżnia się wiele prostszych operatorów, jednak wszystkie mają taki sam interfejs, a więc przetwarzają pewne strumienie wejściowe na strumienie wyjściowe. Wyjątkiem są operatory scalenia $f_{merge}$, które łączą wiele strumieni w jeden:

$$s_{out} = f_{merge}(s_0, s_1, \dots, s_n) \tag{4.6}$$

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 $w$, bez względu na liczbę poprzednich wywołań:

$$f(\langle v_i, v_{i+1}, \dots \rangle) = \langle w_i, w_{i+1}, \dots \rangle : v_i = v_j \implies w_i = w_j \\ i \neq j \wedge i, j = 0, 1, 2, \dots \tag{4.7}$$

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 brak wewnętrznego stanu, dzięki czemu można przenosić wykonanie operatora pomiędzy środowiskami bez konieczności migracji stanu pamięci.

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

$$f_{map}(\langle v_i, v_{i+1}, \dots \rangle) = \langle g(v_i), g(v_{i+1}), \dots \rangle \tag{4.8}$$

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

$$f_{filter}(\langle v_i, v_{i+1}, \dots \rangle) = \langle v_i : g(v_i) > 0 \rangle \tag{4.9}$$

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

$$f_{scan}(\langle v_0, v_i, v_{i+1}, \dots \rangle) = \langle w_i : w_o = h(v_0, 0) \wedge w_i = h(v_i, h(v_{i-1})) \rangle \tag{4.10}$$

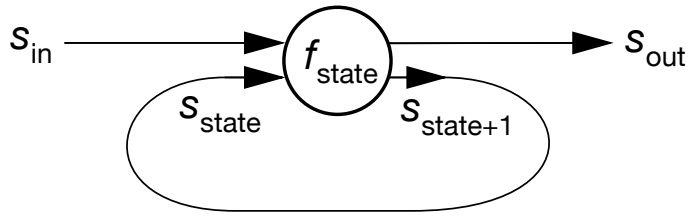
Operator akumulacji generuje strumień wyjściowy na podstawie ostatniej wartości ze strumienia wejściowego i poprzedniej wartości strumienia wyjściowego. Przy-

kładowym zastosowaniem tego operatora jest sumowanie wartości wszystkich zdarzeń.

Stan operatora można również zamodelować jako strumień zdarzeń $s_{state} = \langle state_1, state_2, \dots \rangle$. W tym wypadku stanowy operator $f_{state}$ otrzymuje na wejściu parę strumieni $\{s_{in}, s_{state}\}$ i generuje parę strumieni $\{s_{out}, s_{state+1}\}$:

$$f_{state}(s_{in}, s_{state}) = \{s_{out}, s_{state+1}\} \tag{4.11}$$

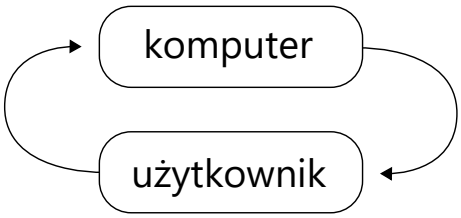
Strumień $s_{state+1}$ jest strumieniem stanu $s_{state}$ rozszerzonym o nowy stan otrzymany po wykonaniu funkcji operatora. Można więc założyć, że operator stanowy jest w pewnym sensie funkcją rekurencyjną. Model operatora stanowego ze strumieniem stanu został przedstawiony na rysunku 4.2.




Rysunek 4.2: Strumień stanu

### 4.1.3 Iteracyjne aplikacje interaktywne

Interakcja pomiędzy aplikacją i użytkownikiem jest (ang. *human-computer interaction, HCI*) to dwustronny proces, w którym obie strony produkują i konsumują informacje, jak przedstawiono na rysunku 4.3. 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.




Rysunek 4.3: Interakcja użytkownik-komputer

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 zachodzących przerwań, pojawiających się sygnałów zegarowych lub zdarzeń, współdziałanie sytemu operacyjnego z aplikacją można zamodelować jako cykl przetwarzania strumieni zdarzeń. Jeden taki cykl nazywa się iteracją aplikacji interaktywnej.

Strumienie zdarzeń przekazywane są do aplikacji, która przetwarza je zgodnie z góry określonym acyklicznym grafem skierowanym $D$, w którym zbiór wierzchołków $F$ oznacza operatory, a zbiór krawędzi $S$ oznacza strumienie pomiędzy nimi:

$$D = (F, S) \tag{4.12}$$

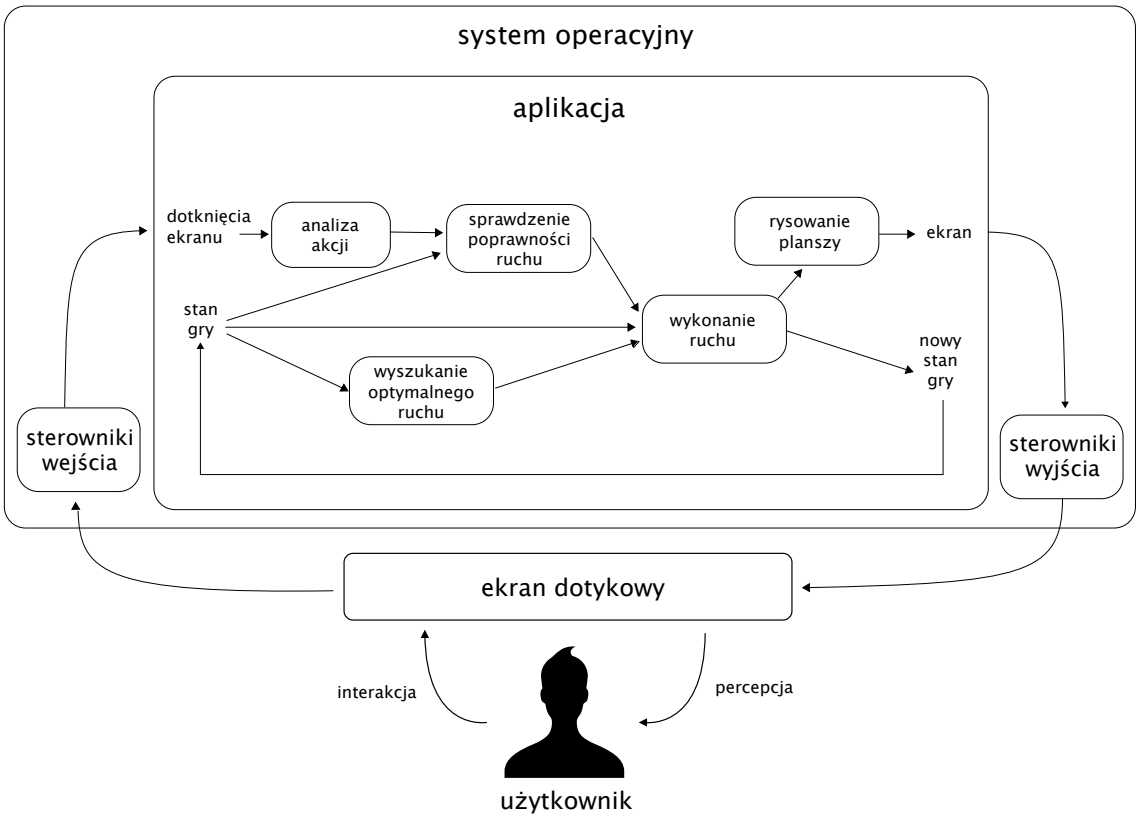
$$h = |F| \tag{4.13}$$

$$S = \{(f_x, f_y) : f_x \in F \wedge f_y \in F \wedge f_x \neq f_y\} \tag{4.14}$$

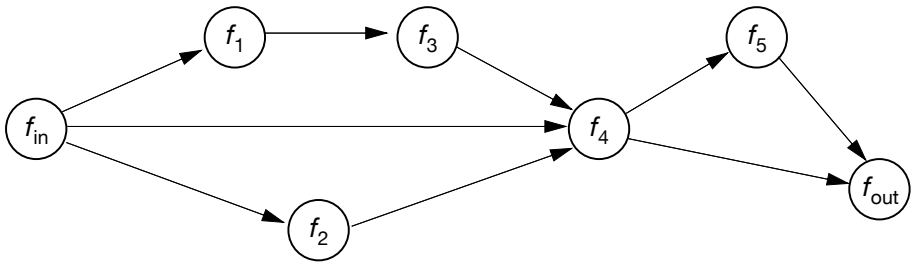
Każdemu strumieniowi $s \in S$ przyporządkowany jest ciąg zdarzeń $\langle v_i, v_{i+1}, v_{i+2}, \ldots \rangle$ przesyłanych w tym strumieniu, a $V$ to zbiór wszystkich zdarzeń we wszystkich strumieniach.

Dzięki koncepcji strumieni oraz powtarzającemu się cyklowi przepływu zdarzeń pomiędzy systemem i aplikacją zamodelowany został asynchroniczny i niedeterministyczny przepływ sterowania [54]. Dla przykładu na rysunku 4.4a przedstawiony został model architektury gry w szachy na urządzeniu mobilnym i graf odpowiadający każdej iteracji aplikacji. Wierzchołki $f_{in}$ i $f_{out}$ reprezentują sterowniki wejścia i wyjścia, natomiast wierzchołki od $f_1$ do $f_5$ to operatory implementujące logikę przetwarzania aplikacji.

Aplikacja posiada jeden strumień wejściowy – dotknięcia ekranu. Należy zauważyć, że aplikacja nie posiada żadnego globalnego stanu gry, jest on natomiast wewnątrz aplikacji w postaci cyklicznego strumienia (oznaczonego przerywaną linią). Aplikacja została podzielona na $h = 5$ operatorów. Dotknięcia ekranu zostają mapowane na akcje (*analiza akcji*), tzn. ruchy figur na szachownicy. Ze strumienia ruchów wybrane zostają tylko dozwolone ruchy (*sprawdzenie poprawności ruchu*) i




(a) architektura aplikacji




(b) graf aplikacji

Rysunek 4.4: Architektura komponentowa aplikacji do gry w szachy (a) i odpowiadający jej graf pojedynczej iteracji (b)

po scaleniu z aktualnym stanem gry generują nowy stan (*wykonanie ruchu*). Co drugi stan gry wykonany zostaje ruch przeciwnika, tzn. komputera (*wyszukanie optymalnego ruchu*). W tym celu, na podstawie strumienia stanu wyszukany zostaje optymalny ruch, który przekazywany jest w postaci strumienia do omawianego już operatora wykonania ruchu. Liczba iteracji w tym przypadku zależy od przebiegu

gry, od poziomu zaawansowania graczy i nie jest przewidywalna na początku gry. Na koniec aktualny stan gry jest rysowany w formie szachownicy na ekranie (*rysowanie planszy*).

Możliwe jest dalsze dzielenie operatorów na mniejsze i w rezultacie zwiększenie granularności grafu. Szczególnie operator *wyszukanie optymalnego ruchu* wydaje się dość złożoną funkcją i mógłby być dobrym kandydatem do podziału. Jednak ze względu na zastosowaną implementację algorytmu sztucznej inteligencji, która wyszukuje optymalny ruch (wariant algorytmu Minimax [50]), podział skutkowałby otrzymaniem wielu ściśle połączonych ze sobą operatorów z dużą ilością komunikacji pomiędzy nimi, co prawdopodobnie obniżyłoby wydajność aplikacji.

## 4.2 Problem rozdziału aplikacji interaktywnej

Optymalizacja interaktywnej aplikacji mobilnej w modelu opisanym w rozdziale 4.1 polega na rozdziale zbioru operatorów $F$ pomiędzy urządzenie i w chmurę obliczeniową. Koncepcja rozdziału aplikacji została przedstawiona na rysunku 4.5. Jednak w odróżnieniu od modelu statycznego, w modelu dynamicznym optymalny rozdział może różnić się dla każdej iteracji, dlatego optymalizacja polega na wyznaczeniu ciągu rozdziałów $R$ o długości $l$ równej liczbie iteracji aplikacji, w którym każdy element $r_i$ odpowiada rozdziałowi operatorów $\{F_c^i, F_m^i\}$ w danej iteracji:

$$r_i \in R, i = 1, 2, \ldots, l \tag{4.15}$$

$$r_i = \{F_c^i, F_m^i\} \tag{4.16}$$

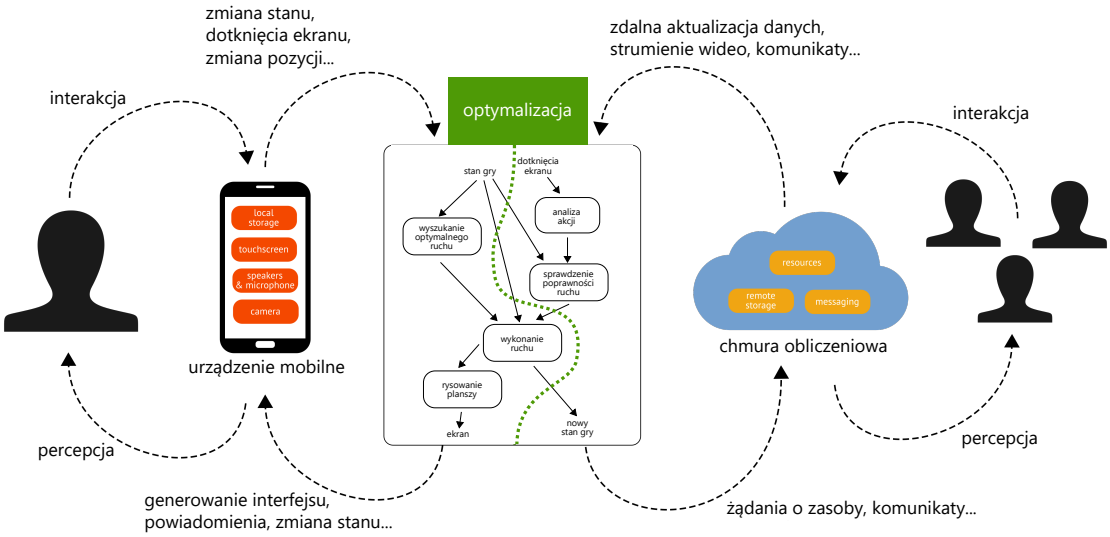
$$R = (\{F_c^1, F_m^1\}, \{F_c^2, F_m^2\}, \ldots, \{F_c^l, F_m^l\}) \tag{4.17}$$

Można założyć, że dla pojedynczej iteracji operatory są równoważne komponentom w aplikacji statycznej, opisanej w rozdziale 3. Możemy więc analogicznie określić zbiór strumieni wewnętrznych $S_{int}^i$ i zewnętrznych $S_{ext}^i$ dla danego rozdziału $r_i$ w iteracji $i$. Jeśli założymy, że strumień $s_y^x$ to strumień generowany przez operator $f_x$ i przesyłany na wejście operatora $f_y$ to:

$$S_{int}^i : \{s_y^x : (f_x \in F_c^i \wedge f_y \in F_c^i) \vee (f_x \in F_m^i \wedge f_y \in F_m^i)\} \tag{4.18}$$

$$S_{ext}^i = S \setminus S_{int}^i \tag{4.19}$$

Możemy również określić zbiór zdarzeń $V_{ext}^i$ przesyłanych pomiędzy chmurą ob-




Rysunek 4.5: Koncepcja optymalizacji rozdziału komponentów aplikacji

liczeniową i urządzeniem w iteracji $i$ oraz zbiór zdarzeń $V_{int}^i$ przesyłanych tylko w ramach jednego środowiska:

$$v_i \in s \wedge s \in S_{int}^i \implies v_i \in V_{int}^i \tag{4.20}$$

$$v_i \in s \wedge s \in S_{ext}^i \implies v_i \in V_{ext}^i \tag{4.21}$$

Należy zauważyć, że ze względu na możliwość innego rozdziału operatorów $r_i$ w każdej iteracji, zbiór strumieni wewnętrznych $S_{int}^i$ i zewnętrznych $S_{ext}^i$ oraz zbiór zdarzeń przesyłanych wewnętrznie $V_{int}^i$ i zewnętrznie $V_{ext}^i$ również może się różnić.

## 4.3 Parametry wykonania operatorów

W celu dokonania optymalnego rozdziału konieczne jest zmierzenie parametrów wykonania operatorów oraz strumieni. Z punktu widzenia pojedynczej iteracji parametry te odpowiadają parametrom wykonania komponentów w statycznym modelu aplikacji:

- Czas wykonania - czas potrzebny do przetworzenia pojedynczego zdarzenia $v_i$ w danej iteracji. Dla operatorów mapowania obliczenie czasu wykonania jest proste, ponieważ wystarczy obliczyć różnicę czasów w zdarzeniu wejściowym

$t_i$ i wyjściowym $t'_i$:

$$dt_i = t'_i - t_i \tag{4.22}$$

Analogiczne podejście można zastosować w przypadku operatora akumulacji i scalenia, ponieważ na każde zdarzenie wejściowe generują odpowiadające mu zdarzenie wyjściowe. W przypadku operatora filtrowania $f_{filter}$ 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, które ignorowane są na wejściu pozostałych operatorów:

$$f_{filter}(\langle v_i, v_{i+1}, \dots \rangle) = \langle v_i : g(v_i) > 0 \vee \oslash : g(v_i) \leq 0 \rangle \tag{4.23}$$

Ponieważ każdy z operatorów może być uruchomiony w chmurze obliczeniowej lub na urządzeniu mobilnym, a czasy przetwarzania danego zdarzenia mogą się różnić w zależności od środowiska uruchomienia, wyróżniamy dwie funkcje związane z czasem wykonania operatora:

  - $t_m(f, i)$ - czas wykonania operatora $f$ na urządzeniu mobilnym w iteracji $i$
  - $t_c(f, i)$ - czas wykonania operatora $f$ w chmurze w iteracji $i$

- Rozmiar zdarzenia w strumieniu $z(v_i)$ - rozmiar danych zdarzenia $v_i$ w strumieniu $s$ liczony w bajtach. Ponieważ zdarzenia mogą mieć dowolne wartości, rozmiar danych może się znacznie różnić w kolejnych iteracjach. Może być również zerowy, np. w przypadku wspomnianego wyżej operatora filtrowania $f_{filter}$, który generuje zdarzenia puste jeśli nie spełniają one warunku funkcji filtrującej.

### 4.3.1 Funkcja kosztu

Koszty wykonania operatora $f$ w iteracji $i$ w chmurze $k_c(f, i)$ i na urządzeniu mobilnym $k_m(f, i)$ są proporcjonalne do czasu przetwarzania zdarzenia $v_i$:

$$k_m(f, i) = P_m * t_m(f, i) \tag{4.24}$$

$$k_c(f, i) = P_c * t_c(f, i) \tag{4.25}$$

Parametry $P_m$ oraz $P_c$ to, analogicznie jak w przypadku modelu statycznego, współczynniki poboru energii w chmurze i na urządzeniu mobilnym. Również w modelu dynamicznym zakładamy, że są one stałe i równe dla wszystkich komponentów, ponieważ związane są ze sprzętowymi parametrami środowiska wykonania, które nie ulega modyfikacji w czasie działania aplikacji.

Całkowity koszt wykonania danego operatora to suma kosztów wszystkich iteracji:

$$K_m(f) = \sum_{i=1}^{l} k_m(f, i) \tag{4.26}$$

$$K_c(f) = \sum_{i=1}^{l} k_c(f, i) \tag{4.27}$$

Koszt transferu strumienia zdarzeń pomiędzy urządzeniem mobilnym i chmurą obliczeniową jest proporcjonalny do przepustowości strumienia, tzn. rozmiaru wszystkich przesłanych zdarzeń w strumieniu, oraz odwrotnie proporcjonalny do przepustowości sieci. Dla pojedynczej iteracji $i$ możemy obliczyć czas przesyłania zdarzenia $t_v(v_i)$:

$$t_v(v_i) = \frac{z(v_i)}{B} \tag{4.28}$$

Podobnie jak w przypadku modelu statycznego, koszt transferu $k_v$ zdarzenia $v_i$ pomiędzy operatorami uruchomionymi w danej iteracji w tym samym środowisku jest zerowy:

$$\forall v_i \in V_{int}^i : t_v(v_i) = 0, k_v(v_i) = 0 \tag{4.29}$$

Natomiast w przypadku przesyłania zdarzenia pomiędzy chmurą obliczeniową i urządzeniem mobilnym jest on zależny od czasu $t_v(v_i)$ i współczynnika poboru energii podzespołów $P_e$ wykorzystywanych przy transmisji:

$$k_v(v_i) = P_e * t_v(v_i) \tag{4.30}$$

Koszt transferu strumienia to suma kosztów transferu wszystkich zdarzeń we wszystkich iteracjach:

$$K_s(s) = \sum_{v_i \in s} k_v(v_i) \tag{4.31}$$

Należy uwzględnić również koszt migracji stanu operatora $k_g(f)$. W przypadku operatorów bezstanowych rozmiar stanu $z(state_i) = 0$. Oczywiście koszt migracji jest niezerowy tylko w przypadku, gdy wykonanie operatora $f$ ma zostać przeniesione do innego środowiska, tzn. w iteracji $i-1$ był uruchomiony w chmurze, a w iteracji $i$ zostanie uruchomiony na urządzeniu mobilnym lub odwrotnie.

$$k_g(f,i) = \begin{cases} P_e * \frac{z(state_i)}{B}, & \text{jeśli } (f \in F_c^i \wedge f \in F_m^{i-1}) \vee (f \in F_m^i \wedge f \in F_c^{i-1}) \\ 0, & \text{jeśli } (f \in F_c^i \wedge f \in F_c^{i-1}) \vee (f \in F_m^i \wedge f \in F_m^{i-1}) \end{cases} \tag{4.32}$$

Całkowity koszt migracji operatora $f$ to suma migracji w każdej iteracji:

$$K_g(f) = \sum_{i=1}^{l} k_g(f,i) \tag{4.33}$$

Uwzględniając opisane powyżej funkcje kosztu dla operatorów oraz strumieni można rozszerzyć model aplikacji interaktywnej $D$ do modelu kosztu $D_K$:

$$D_K = (F, S, W) \tag{4.34}$$

$$W = (k_m, k_c, k_v, k_g) \tag{4.35}$$

$$k_m : F \to \mathbb{R}, k_c : F \to \mathbb{R}, k_g : F \to \mathbb{R}, k_v : V \to \mathbb{R} \tag{4.36}$$

Koszt wykonania pojedynczej iteracji $i$ dla danego rozdziału $r_i$ analogicznie jak w aplikacji statycznej jest sumą kosztów wykonania wszystkich operatorów $f \in F$ i przesłania wszystkich zdarzeń $v \in V_{ext}^i$, ale w odróżnieniu od aplikacji uwzględnia również koszt migracji operatorów $k_g(f)$:

$$K(r_i) = \sum_{f \in F_c^i} k_c(f) + \sum_{f \in F_m^i} k_m(f) + \sum_{v \in V_{ext}^i} k_v(v) + \sum_{f \in F} k_g(f) \tag{4.37}$$

Całkowity koszt wykonania aplikacji rozumiany jest jako suma kosztów wykonania wszystkich iteracji $K(r_i)$, gdzie $i = 1, 2, \ldots, l$:

$$K(R) = \sum_{i=1}^{l} K(r_i) = \sum_{i=1}^{l} K(\{F_c^i, F_m^i\}) \tag{4.38}$$

Koszt całkowity można podzielić na trzy składniki: całkowity koszt urządzenia mobilnego $K_m(R)$, całkowity koszt chmury obliczeniowej $K_c(R)$ oraz całkowity koszt komunikacji $K_e(R)$:

$$K(R) = K_c(R) + K_m(R) + K_e(R) \tag{4.39}$$

Przyjęto, że w całkowitym koszcie komunikacji $K_e$ uwzględniony jest całkowity koszt transmisji strumieni $K_s$ oraz całkowity koszt migracji $K_g$, a zatem:

$$K_e(R) = K_s(R) + K_g(R) \tag{4.40}$$

## 4.4 Kryteria i algorytmy rozdziału

Dla pojedynczej iteracji analogicznie jak w przypadku modelu statycznego, celem optymalizacji jest minimalizacja kosztu wykonania aplikacji $K$ przy jednoczesnym ograniczeniu kosztu częściowego urządzenia mobilnego $K_m$ do założonego kosztu maksymalnego $m_{max}$ i ograniczeniu kosztu częściowego chmury obliczeniowej $K_c$ do założonego kosztu maksymalnego $c_{max}$.

Ciąg rozdziałów $R_{opt}$ jest optymalny jeśli:

$$K(R_{opt}) = min \tag{4.41}$$

$$K_m(R_{opt}) \leqslant K_m^{max} \tag{4.42}$$

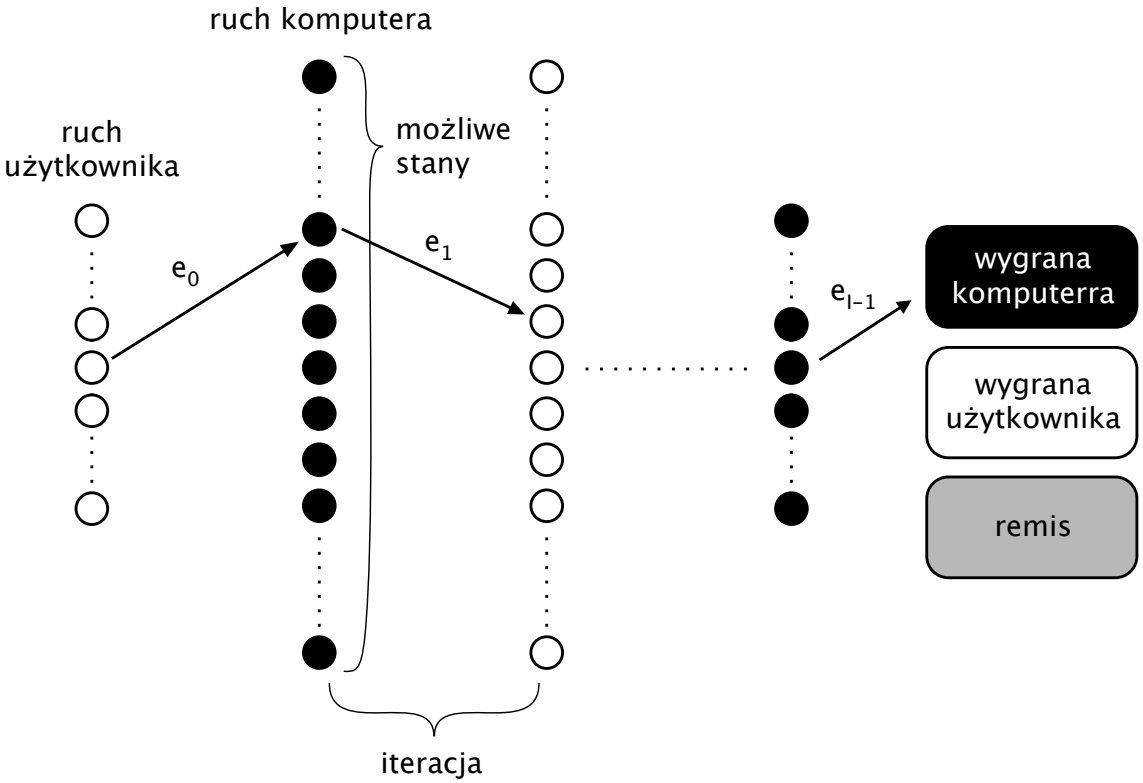
$$K_c(R_{opt}) \leqslant K_c^{max} \tag{4.43}$$

### 4.4.1 Możliwa przestrzeń rozwiązań

Należy zauważyć, że w ogólności koszt 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, a więc używając analogicznego podejścia jak w statycznym modelu aplikacji, 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 stanów gry po zaledwie dwóch ruchach. Ostatecznie przebieg gry kończy się na jednym z trzech możliwych stanów: wygrana białych, wygrana czarnych lub remis. Średnia liczba ruchów w partii, a więc również iteracji w aplikacji do gry w szachy, wynosi około 40 ruchów w przypadku rozgrywki zaawansowanych graczy [63]. Rysunek 4.6 przedstawia ilustrację potencjalnych przebiegów i stanów gry.




Rysunek 4.6: Możliwe stany gry w szachy

Mimo, że w każdym z potencjalnych stanów graf $D$ jest taki sam, czasy i koszty wykonania operatorów, a także koszty komunikacji mogą się różnić. W przypadku omawianej gry w szachy obliczenie optymalnego ruchu komputera w dalszej części gry może trwać dłużej niż w początkowej fazie, a co za tym idzie optymalny rozdział może się różnić w każdej z iteracji.

Liczbę wszystkich możliwych stanów gry $Q$ trwającej $l$ iteracji można obliczyć na podstawie liczby możliwości $q_i$ w $i$-tej iteracji:

$$Q = q_0 \cdot q_1 \cdot \ldots \cdot q_l \tag{4.44}$$

Średnia liczba ruchów w partii w praktyce wynosi około 40 ruchów w przypadku rozgrywki zaawansowanych graczy [63]. W każdym ruchu odbywają się dwie iteracje, posunięcie gracza białego i posunięcie gracza czarnego, a więc łącznie gra ma około $l = 80$ iteracji. Średnia liczba możliwości w danym posunięciu jest różna w zależności od liczby figur na planszy, ale przez większość gry wynosi około $\bar{q} = 30$ [89]. Można więc oszacować, że przeciętna liczba możliwych stanów gry w szachy wynosi:

$$Q \approx \bar{q}^l = 30^{80} \approx 10^{118} \tag{4.45}$$

Dla uproszczenia rozważmy sytuację, w której możemy ograniczyć potencjalne stany gry i uśrednić koszty operatorów we wszystkich stanach w danej iteracji, np. wychodząc z założenia, że w środkowej fazie gry koszt obliczenia ruchu jest zawsze wyższy, bez względu na faktyczny stan szachownicy. Z punktu widzenia rozdziału operatorów wszystkie stany są więc tożsame, a zatem możemy utworzyć model pełnego przebiegu sterowania w aplikacji powielając $l$ razy graf $D$.

Graf będący przedmiotem optymalizacji posiadałby $h * l$ wierzchołków, gdzie $h$ to liczba operatorów w każdej iteracji, a $l$ to liczba iteracji. Znalezienie optymalnego rozdziału tak dużego grafu wiązałoby się dużym narzutem obliczeniowym, który prawdopodobnie przewyższałby oszczędności wynikające z optymalizacji. Z tego powodu optymalizacja wykonania całkowitego kosztu aplikacji $K(R)$ wydaje się niepraktyczna i w związku z tym rozważać będziemy koszt wykonania pojedynczej iteracji $K(r_i)$.

## 4.5 Proponowany algorytm

W typowym scenariuszu, koszty wykonania operatorów w chmurze będą niższe niż na urządzeniu ze względu na wyższą wydajność węzłów w chmurze. Przeniesienie wszystkich operatorów do chmury może jednak okazać się nieoptymalne ze względu na wysoki koszt transferu wszystkich strumieni zdarzeń. W ogólności koszt transferu w ramach każdej iteracji będzie odwrotnie proporcjonalny do kosztu wykonania operatorów. Innymi słowy, przenosząc wykonanie obliczeń do chmury obliczeniowej zmniejszamy koszt wykonania, ale wprowadzamy dodatkowy koszt związany z transferem danych. Ogólny koszt wykonania iteracji nie musi jednak wcale wzrosnąć,

jeśli czas transferu jest mniejszy niż różnica czasów wykonania danego operatora w chmurze i na urządzeniu.

Należy również zwrócić uwagę na zależności pomiędzy operatorami. W sytuacji, gdy dany operator przesyła strumienie zdarzeń do wielu innych operatorów, może się okazać, że przeniesienie wykonania operatora do chmury ma sens tylko w przypadku przeniesienie wszystkich sąsiednich operatorów. Jeśli bierzemy pod uwagę jedynie pojedynczą iterację jest to więc analogiczny problem optymalizacyjny do problemu opisanego w rozdziale 3.

W przypadku aplikacji złożonej jedynie z operatorów bezstanowych całkowity koszt migracji $K_g$ jest zawsze równy zero. Koszty kolejnych iteracji są więc od siebie niezależne, a zatem znalezienie optymalnego rozdziału $r_i$ dla każdej iteracji będzie równocześnie optymalnym ciągiem rozdziałów $R_{opt}$.

W przypadku operatorów, które posiadają wewnętrzny stan, przeniesienie wykonania pomiędzy dwoma środowiskami wiąże się również z migracją stanu. Migracja stanu odbywać się będzie przy każdym przeniesieniu wykonania pomiędzy chmurą obliczeniową i urządzeniem, a co za tym idzie całkowity koszt iteracji jest zależny od rozdziału operatorów w poprzedniej iteracji, co dodatkowo utrudnia znalezienie optymalnego rozdziału. Przykładowo, przeniesienie wykonania danego operatora oraz migracja jego stanu może być opłacalna tylko jeśli aplikacja będzie miała wiele iteracji i całkowity zysk z przeniesienia operatora będzie większy niż koszt migracji.

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 optymalnym, 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.

### 4.5.1 Statyczny algorytm optymalizacji

Ponieważ rozdział pojedynczej iteracji można interpretować jak rozdział aplikacji w modelu statycznym, do optymalizacji grafu operatorów można zastosować algorytmy oparte o metody opisane w rozdziale 3.5, tzn. algorytm DRK (Dokładny Rozdział Komponentów) i GRK (Genetyczny Rozdział Komponentów).

Danymi wejściowymi dla Algorytm Dokładnego Rozdziału Operatorów (DRO) są zbiór operatorów $F$, zbiór strumieni $S$, funkcje kosztów $W$, maksymalny dopusz-

czalny koszt chmury obliczeniowej $c_{max}$, maksymalny dopuszczalny koszt urządzenia $m_{max}$, aktualne iteracja $i$ oraz rozdział operatorów w poprzedniej iteracji $r_{i-1}$. Główna różnica pomiędzy algorytmem DRO i algorytmem DRK to właśnie zależność od rozdziału w poprzedniej iteracji – jest ona konieczna do uwzględniania kosztów migracji. Dla iteracji $i = 1$ zakładamy, że rozdział poprzedniej iteracji $r_0$ jest pusty, a więc koszty migracji są zerowe.

**function** DokładnyRozdziałOperatorów($F, S, W, c_{max}, m_{max}, i, r_{i-1}$)
  $K_{min} \leftarrow \inf$, $r_i \leftarrow null$
  **for all** $F_c^i \in \mathcal{P}(F)$ **do**
    $F_m^i \leftarrow F \setminus F_c^i$
    $K_c \leftarrow 0$, $K_m \leftarrow 0$, $K_s \leftarrow 0, K_g \leftarrow 0$
    **for all** $f \in F_c^i$ **do**
      $K_c \leftarrow K_c + k_c(f, i)$
      **if** $f \in F_m^{i-1}$ **then**
        $K_g \leftarrow K_g + k_g(f, i)$
      **end if**
    **end for**
    **for all** $f \in F_m^i$ **do**
      $K_m \leftarrow K_m + k_m(f, i)$
      **if** $f \in F_c^{i-1}$ **then**
        $K_g \leftarrow K_g + k_g(f, i)$
      **end if**
    **end for**
    **for all** $v \in V_{ext}^i$ **do**
      $K_s \leftarrow K_s + k_v(v)$
    **end for**
    $K \leftarrow K_c + K_m + K_s + K_g$
    **if** $K_c \leq c_{max}$ **and** $K_m \leq m_{max}$ **and** $K < K_{min}$ **then**
      $K_{min} \leftarrow K$
      $r_i \leftarrow \{F_c, F_m\}$
    **end if**
  **end for**
  **return** $r_i$
**end function**

Złożoność prezentowanego algorytm DRO zależy od liczności zbioru potęgowego operatorów $|\mathcal{P}(F)| = 2^{|F|} = 2^h$, a więc wynosi $O(h2^h)$ dla każdej iteracji. Jeśli zastosujemy algorytm dla całej aplikacji trwającej $l$ iteracji całkowita złożoność wynosi $O(lh2^h)$.

W podobny sposób można dostosować algorytm GRK na potrzeby rozdziału iteracji w dynamicznym modelu aplikacji. Algorytm Genetycznego Rozdziału Operatorów (GRO) wymaga zastosowania innej funkcji selekcji, która uwzględnia koszt migracji:

**function** SelekcjaTurniejowa($Population, F, S, W, c_{max}, m_{max}, i, r_{i-1}$)
  $K_{min} \leftarrow \inf$, $r_i \leftarrow \emptyset$
  **for** $i$ to $s_{size}$ **do**
    $F_c^i \leftarrow$ *random from Population*
    $F_m^i \leftarrow F \setminus F_c^i$
    $K_c \leftarrow 0$, $K_m \leftarrow 0$, $K_s \leftarrow 0, K_g \leftarrow 0$
    **for all** $f \in F_c^i$ **do**
      $K_c \leftarrow K_c + k_c(f, i)$
      **if** $f \in F_m^{i-1}$ **then**
        $K_g \leftarrow K_g + k_g(f, i)$
      **end if**
    **end for**
    **for all** $f \in F_m^i$ **do**
      $K_m \leftarrow K_m + k_m(f, i)$
      **if** $f \in F_c^{i-1}$ **then**
        $K_g \leftarrow K_g + k_g(f, i)$
      **end if**
    **end for**
    **for all** $v \in V_{ext}^i$ **do**
      $K_s \leftarrow K_s + k_v(v)$
    **end for**
    $K \leftarrow K_c + K_m + K_s + K_g$
    **if** $K_c \leq c_{max}$ **and** $K_m \leq m_{max}$ **and** $K < K_{min}$ **then**
      $K_{min} \leftarrow K$
      $r_i \leftarrow \{F_c, F_m\}$

```
        end if
    end for
    return r_i
end function
```

Pozostałe funkcje wykorzystywane w algorytmie genetycznym pozostają niezmienione – jedyną różnicą jest to, że działają na zbiorach operatorów, a nie komponentów. Pełen kod algorytmu GRO wygląda więc następująco:

```
function GentycznyRozdziałOperatorów(F, S, W, c_max, m_max, i, r_{i-1}, it, s, mut)
    Population ← ∅
    for j = 1 to s do
        Population ← LosowyPodzbiór(F)
    end for
    for j = 1 to it do
        NewPopulation ← ∅
        for u = 1 to s do
            F_c^A ← SelekcjaTurniejowa(Population, F, S, W, c_max, m_max, i, r_{i-1})
            F_c^B ← SelekcjaTurniejowa(Population, F, S, W, c_max, m_max, i, r_{i-1})
            F_c^i ← Krzyżowanie(F_c^A, F_c^B, F)
            w ← random from 0 to 1
            if w < mut then
                Mutacja(F_c^i, F)
            end if
            F_m^i ← F \ F_c^i
            r_i = {F_c^i, F_m^i}
            add r_i to NewPopulation
        end for
        Population ← NewPopulation
    end for
    return r_i from Population where K(r_i) is minimum
        and K_c(r_i) ≤ c_max and K_m(r_i) ≤ m_max
end function
```

Analogicznie jak w przypadku algorytmu GRK, algorytm GRO wykonuje *it* ite-

racji, w każdej z nich generowana jest populacja składająca się z $s$ podzbiorów operatorów. Każdy podzbiór jest przechodzi proces mutacji z prawdopodobieństwem $mut$. Ponadto parametrami algorytmu jest aktualna iteracja $i$ oraz rozdział operatorów w poprzedniej iteracji $r_{i-1}$.

W rozdziale 3.6 pokazano, że do skutecznego działania algorytm genetyczny wymaga zwiększania liczby iteracji $it$ i rozmiaru populacji $s$ proporcjonalnie do liczby operatorów $h$. Przy takim założeniu algorytm GRO ma złożoność $O(h^3)$ dla jednej iteracji oraz $O(lh^3)$ dla $l$ iteracji aplikacji.

Jak widać nawet algorytm heurystyczny, który miał zastosowanie w statycznym modelu aplikacji, w przypadku modelu dynamicznego ma dużo większą złożoność obliczeniową ze względu na konieczność przeprowadzenia rozdziału w każdej iteracji. Można jednak rozważyć inną wersję tego algorytmu – stosowaną jednorazowo na uśrednionych kosztach w każdej iteracji. Algorytm Uśrednionego Rozdziału Operatorów (URO) wygląda następująco:

**function** UśrednionyRozdziałOperatorów($F, S, \bar{W}, c_{max}, m_{max}, it, s, mut$)
  $Population \leftarrow \emptyset$
  **for** $j = 1$ to $s$ **do**
    $Population \leftarrow$ LosowyPodzbiór($F$)
  **end for**
  **for** $j = 1$ to $it$ **do**
    $NewPopulation \leftarrow \emptyset$
    **for** $u = 1$ to $s$ **do**
      $F_c^A \leftarrow$ SelekcjaTurniejowa($Population, F, S, W, c_{max}, m_{max}$)
      $F_c^B \leftarrow$ SelekcjaTurniejowa($Population, F, S, W, c_{max}, m_{max}$)
      $F_c \leftarrow$ Krzyżowanie($F_c^A, F_c^B, F$)
      $w \leftarrow$ *random from 0 to 1*
      **if** $w < mut$ **then**
        Mutacja($F_c, F$)
      **end if**
      $F_m \leftarrow F \setminus F_c$
      $r = \{F_c, F_m\}$
      *add r to NewPopulation*
    **end for**
    $Population \leftarrow NewPopulation$

**end for**
**return** *r from Population where* $K(r)$ *is minimum*
**and** $K_c(r) \leq c_{max}$ **and** $K_m(r) \leq m_{max}$
**end function**

Algorytm URO nie jest zależny od iteracji $i$, a więc rozdział $r$ obliczany jest jednorazowo i stosowany w każdej iteracji. W związku z tym nie ma też konieczności uwzględniania kosztów migracji, ponieważ dany operator wykonywany jest na urządzeniu lub w chmurze obliczeniowej przez cały czas życia aplikacji. Złożoność obliczeniowa algorytmu URO dla $l$ iteracji jest zatem taka sama jak algorytmu GRO dla pojedynczej iteracji: $O(h^3)$.

Opisany algorytm wymaga podania uśrednionych funkcji kosztów $\bar{W}$ jako dane wejściowe. Uśrednione funkcje możemy obliczyć na podstawie funkcji kosztów $W = (k_m, k_c, k_s, k_g)$ w następujący sposób:

$$\bar{W} = (\bar{k_m}, \bar{k_c}, \bar{k_s}) \tag{4.46}$$

$$\bar{k_m}(f) = \sum_{i=1}^{l} \frac{k_m(f,i)}{l} \tag{4.47}$$

$$\bar{k_c}(f) = \sum_{i=1}^{l} \frac{k_c(f,i)}{l} \tag{4.48}$$

$$\bar{k_s}(s) = \sum_{i=1, v_i \in s}^{l} \frac{k_v(v_i)}{l} \tag{4.49}$$

Uśrednione koszty wykonania i transmisji po wszystkich iteracjach mogą mieć zastosowanie w przypadku aplikacji, w których nie ma dużej zmienności pomiędzy iteracjami, tzn. czasy wykonania operatorów dla kolejnych zdarzeń i wielkości danych w tych zdarzeniach są podobne.

## 4.6 Iteracyjny algorytm heurystyczny

Iteracyjne algorytmy alokacji były wcześniej z powodzeniem stosowane do optymalizacji aplikacji rozproszonych [22]. Iteracyjny charakter modelu aplikacji interaktywnej pozwala na zastosowanie dynamicznej metody alokacji, z wykorzystaniem algorytmu zachłannego, który w każdej iteracji znajdzie podział lokalnie optymalny przez

przeniesienie maksymalnie jednego operatora pomiędzy środowiskami [53]. Narzut algorytmu będzie bardzo niski, ponieważ sprowadza się on do znalezienia jednego operatora $f$, którego przeniesienie będzie skutkowało największą różnicą kosztów $\Delta k_{max}$ przy spełnieniu warunku maksymalnego kosztu przetwarzania na chmurze $c_{max}$ i urządzeniu $m_{max}$. Dla jednej iteracji takie przeszukanie grafu operatorów ma złożoność liniową $O(h)$, natomiast dla $l$ iteracji złożoność wynosi $O(lh)$.

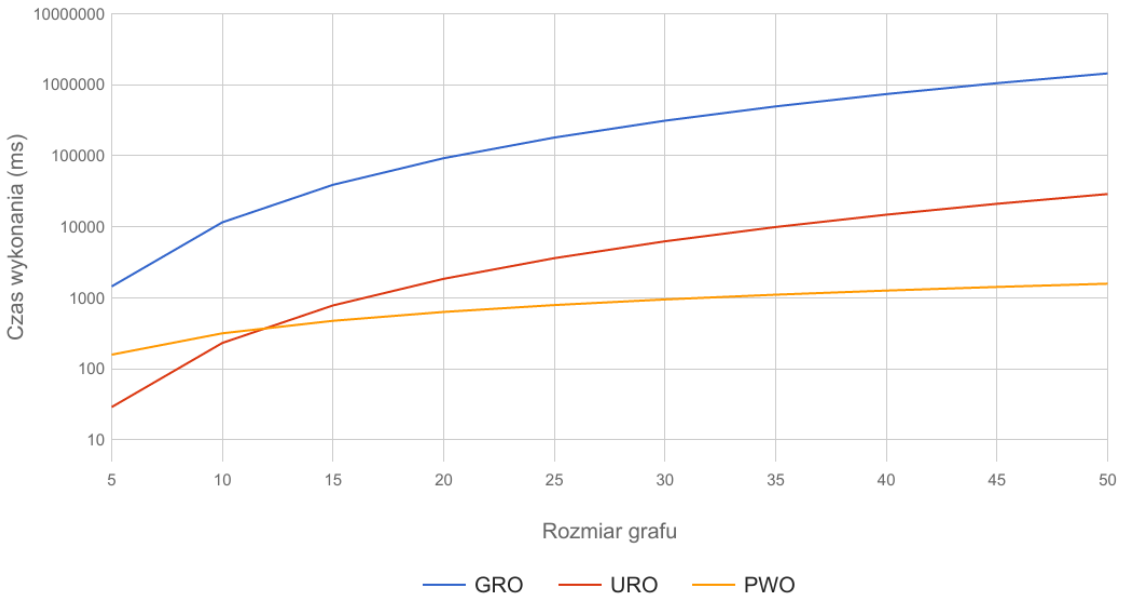
**function** PojedynczaWymianaOperatora($F, S, W, c_{max}, m_{max}, i, r_{i-1}, l$)
  $\Delta k_{max} \leftarrow 0$
  $r_i^{opt} \leftarrow null$
  **for all** $f \in F$ **do**
    **if** $f \in F_c^{i-1}$ **then**
      $F_c^i \leftarrow F_c^{i-1} \setminus f$
      $F_m^i \leftarrow F_m^{i-1} \cup f$
    **else**
      $F_c^i \leftarrow F_c^{i-1} \cup f$
      $F_m^i \leftarrow F_m^{i-1} \setminus f$
    **end if**
    $r_i = \{F_c^i, F_m^i\}$
    $\Delta k \leftarrow K(r_{i-1}) - K(r_i) - \frac{k_g(f,i)}{l-i}$
    **if** $\Delta k > \Delta k_{max}$ **and** $K_c(r_i) \leq c_{max}$ **and** $K_m(r_i) \leq m_{max}$ **then**
      $\Delta k_{max} \leftarrow \Delta k$
      $r_i^{opt} \leftarrow r_i$
    **end if**
  **end for**
  **return** $r_i^{opt}$
**end function**

Różnica kosztów $\Delta k$ po przeniesieniu operatora $f$ uwzględnia koszt migracji $k_g$ tego operatora podzielony przez liczbę pozostałych do wykonania iteracji $l-i$. Oznacza to, że przeniesienie danego operatora ma sens tylko wtedy, kiedy potencjalna redukcja kosztu we wszystkich kolejnych iteracjach będzie większa niż koszt jego przeniesienia. Możliwa jest więc sytuacja, w której migracja operatora będzie opłacalna tylko we wczesnych iteracjach działania aplikacji.

## 4.7 Analiza porównawcza

Zaproponowany iteracyjny heurystyczny algorytm PWO został przetestowany i porównany z algorytmami GRO oraz URO na losowo generowanych grafach o różnym rozmiarze i o różnej liczbie iteracji. Przebadano grafy o rozmiarach od $h = 5$ do $h = 50$ operatorów, a liczba iteracji $l$ wynosiła od 10 do 250. Testy zostały przeprowadzone dla dwóch wariantów zmienności kosztów w kolejnych iteracjach, wyrażonych jako odchylenie standardowe $\sigma_k$ funkcji kosztów wykonania $k_m$ i $k_c$, kosztów transferu danych $k_v$ oraz kosztów migracji $k_g$.

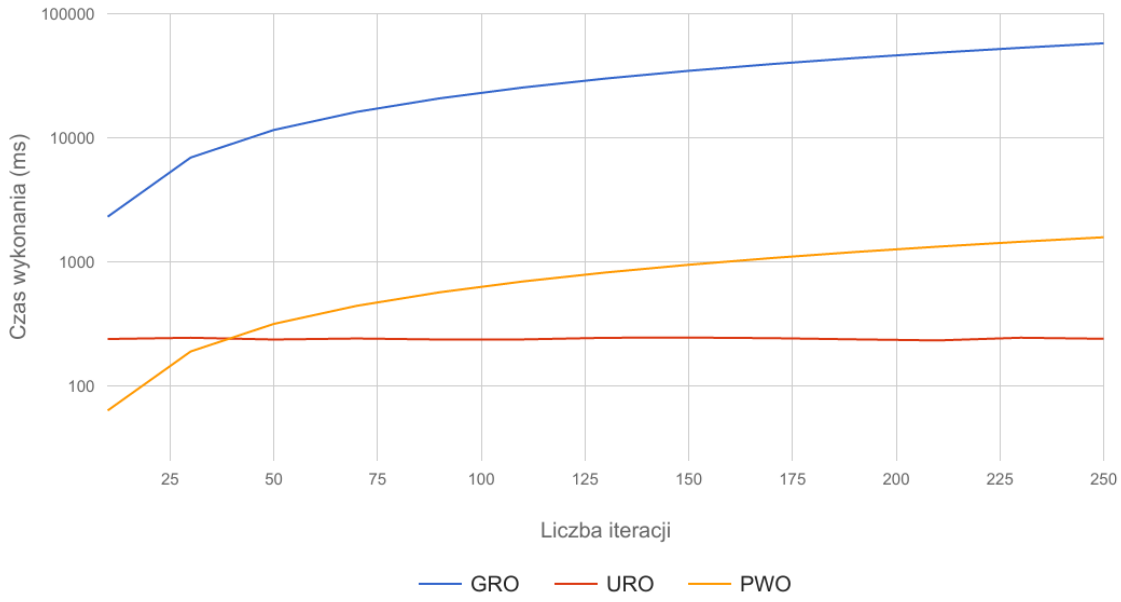
Na rysunku 4.7 przedstawiony został wykres zależności całkowitego czasu wykonania algorytmów optymalizacji dla różnych rozmiarów grafu $h$. Przyjęto stałą liczbę iteracji $l = 50$, czas przedstawiony jest w skali logarytmicznej. Jak widać, czas wykonania algorytmu GRO jest wysoki już przy niewielkich grafach – przy $h = 10$ operatorach wynosi łącznie ponad 10s, więc średnio około 200ms na iterację. Czas wykonania algorytmu URO rośnie w podobny sposób, ale jest kilkadziesiąt razy niższy ponieważ rozdział obliczany jest jednorazowo. Iteracyjny algorytm PWO ma dużo łagodniejszy wzrost i nawet dla dużych grafów o liczbie operatorów $h = 50$ łączny czas wykonania to jedynie około 1,5s, a więc około 30ms na jedną iterację.




Rysunek 4.7: Czas optymalizacji dla różnych rozmiarów grafów $h$

Zależność czasu wykonania od liczby iteracji została przedstawiona na rysunku 4.8.

Przyjęto stałą liczbę operatorów $h = 10$. Również w tym wypadku widać, że algorytm GRO nie nadaje się do zastosowania w modelu iteracyjnym – przy 100 iteracjach łączny czas wykonania to ponad 20s. Algorytm URO nie zależy od liczby iteracji, więc łączny czas obliczenia rozdziału jest stały. W przypadku algorytmu PWO widać wzrost wraz z liczbą iteracji, jednak jest on dużo niższy i czas wykonania dla $l = 250$ wynosi niewiele ponad 1,5s.

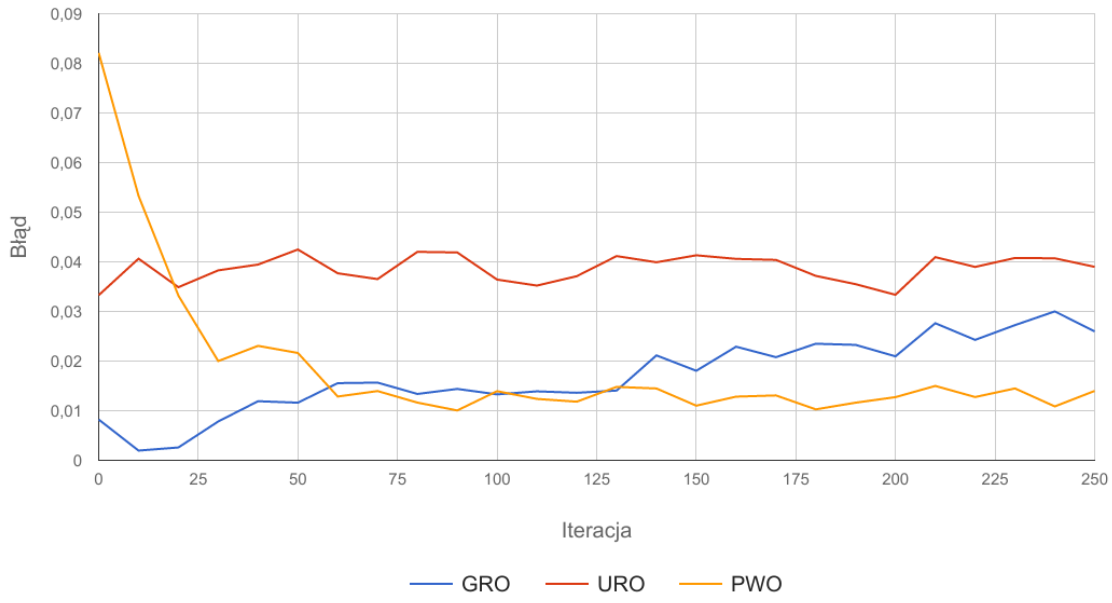



Rysunek 4.8: Czas optymalizacji dla różnej liczby iteracji $l$

Na rysunkach 4.9 i 4.10 przedstawiono wykres wielkości błędu optymalizacji w kolejnych iteracjach. Błąd optymalizacji wyrażony jest jako różnica pomiędzy całkowitym kosztem iteracji dla rozdziału wykonanego za pomocą danego algorytmu i kosztem iteracji dla rozdziału optymalnego, podzielona przez koszt optymalny. Rozdział optymalny został obliczony za pomocą algorytmu DRO. Przyjęto rozmiar grafu $h = 10$ operatorów i $l = 250$ iteracji.

Wykres 4.9 przedstawia porównanie dla niskiej zmienności kosztów w kolejnych iteracjach – odchylenie standardowe $\sigma_k = 0.1$. Rozdział obliczony za pomocą algorytmu URO jest taki sam w każdej iteracji, więc błąd jest relatywnie wysoki i dość stały przez cały czas działania aplikacji. W przypadku algorytmu GRO błąd jest niższy, ale rośnie w kolejnych iteracjach. Wzrost ten związany jest z kosztem migracji operatorów – w późniejszych iteracjach migracja operatora zazwyczaj nie jest już opłacalna, a algorytm GRO nie bierze tego pod uwagę. Algorytm PWO ma

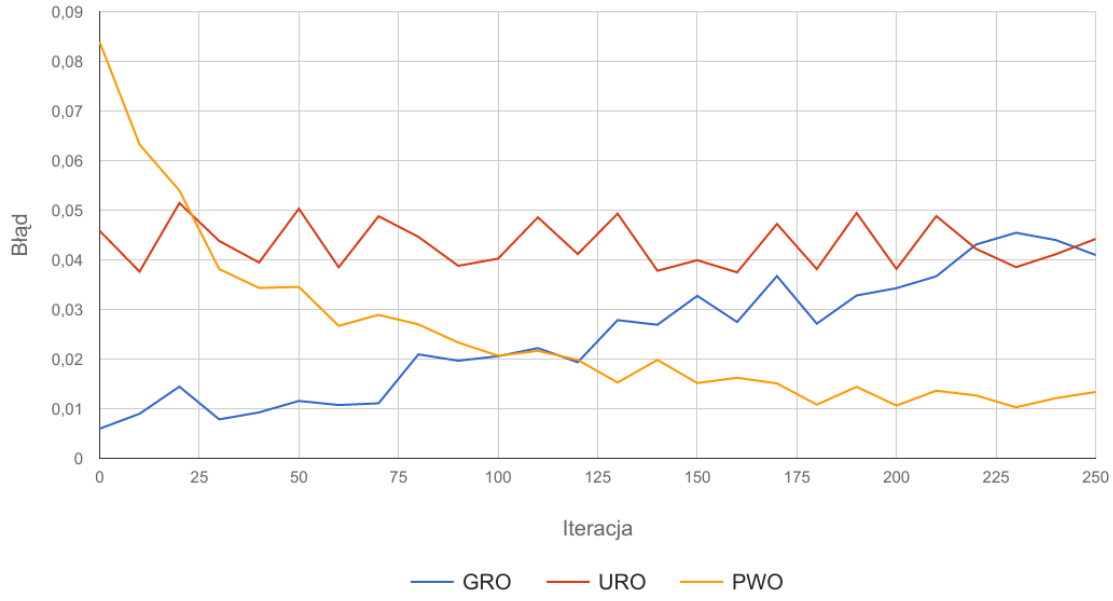
bardzo wysoki błąd optymalizacji w początkowych iteracjach, ponieważ z definicji może przenosić jedynie po jednym operatorze na każdą migrację. Jednak po około 50 iteracjach osiąga stabilny poziom błędu, porównywalny z algorytmem GRO. W końcowych iteracjach wynik jest nawet lepszy niż dla algorytmu genetycznego, ponieważ algorytm PWO uwzględnia opłacalność migracji.




Rysunek 4.9: Błąd optymalizacji w kolejnych iteracjach $i$, zmienność $\sigma_k = 0.1$

Na wykresie 4.10 przedstawione zostały wyniki porównania dla wysokiej zmienności kosztów, odchylenie standardowe $\sigma_k = 0.5$. W tym przypadku wszystkie błędy optymalizacji są generalnie nieco wyższe. Można zaobserwować większe wahania błędu dla algorytmu URO, co wydaje się dość naturalne – rozdział operatorów jest taki sam dla każdej iteracji, a optymalne koszty iteracji różnią się znacznie. Przebieg algorytmu GRO jest podobny jak w przypadku mniejszej zmienności, jednak błąd optymalizacji rośnie tutaj dużo wyraźniej i przy 250 iteracjach jest porównywalny z algorytmem URO. Algorytm PWO potrzebuje więcej iteracji do osiągnięcia rozdziału bliskiego optymalnemu, ale po około 150 iteracjach błąd optymalizacji spada poniżej 1.5%.

Analizując otrzymane wyniki można stwierdzić, że algorytm PWO ma zastosowanie w modelu dynamicznych aplikacji interaktywnych. Czas obliczenia rozdziału operatorów jest akceptowalny zarówno w przypadku dużych grafów, jak również przy dużej liczbie iteracji. Bez względu na zmienność kosztów operatorów w kolejnych




Rysunek 4.10: Błąd optymalizacji w kolejnych iteracjach $i$, zmienność $\sigma_k = 0.5$

iteracjach, algorytm PWO jest w stanie osiągnąć zadowalające wyniki optymalizacji. Jedyną negatywną cechą algorytmu jest relatywnie wysoki błąd w początkowych iteracjach działania aplikacji, jednak w aplikacjach o dużej liczbie iteracji nie ma to tak dużego znaczenia. Problem ten można zminimalizować np. stosując algorytm URO do wstępnego rozdziału operatorów, a następnie używając algorytmu PWO we wszystkich kolejnych iteracjach.

# Rozdział 5

# Badanie eksperymentalne

W celu przetestowania opracowanych algorytmów zbudowano środowisko testowe, w skład którego wchodzi reprezentatywne urządzenie końcowe oraz chmura obliczeniowa. Do zarządzania rozdziałem aplikacji opracowano autorski framework, który umożliwia dynamiczny rozdział operatorów w każdej iteracji aplikacji. Przedstawiono wyniki dla aplikacji gry w szachy oraz monitorowania parkingu.

W ramach badań eksperymentalnych porównane zostały cztery warianty alokacji operatorów aplikacji:

- Aplikacja w całości uruchomiona na urządzeniu mobilnym (*UM*),
- Aplikacja uruchomiona w całości w chmurze, oprócz interfejsu użytkownika, tzw. cienki klient (*ChO*),
- Aplikacja częściowo uruchomiona w chmurze i częściowo na urządzeniu, rozdział operatorów wykonany jednorazowo algorytmem Uśrednionego Rozdziału Operatorów (*UM/ChO-URO*, w skrócie *URO*),
- Aplikacja częściowo uruchomiona w chmurze i częściowo na urządzeniu, rozdział operatorów wykonywany w każdej iteracji za pomocą algorytmu Pojedynczej Wymiany Operatorów (*UM/ChO-PWO*, w skrócie *PWO*).

## 5.1 Scenariusze badań empirycznych

Jednym z podstawowych założeń przyjętych na początku badań było zaproponowanie rozwiązania, które będzie możliwie proste do wdrożenia do istniejących aplikacji mobilnych. Optymalizacja aplikacji oraz rozdział na chmurę i urządzenie powinien

odbywać się w sposób przezroczysty dla użytkownika, a także dla programisty. Jednocześnie, ze względu na mocno ograniczone możliwości ingerencji w system operacyjny urządzeń mobilnych w realnych zastosowaniach, optymalizacja i rozdział musi odbywać się na poziomie procesu aplikacji, a nie na poziomie systemowym. Naturalnym rozwiązaniem jest implementacja mechanizmu optymalizacji rozdziału w ramach biblioteki dołączanej do aplikacji, która dostarcza tzw. *framework* pozwalający na opakowanie głównej funkcji aplikacji.

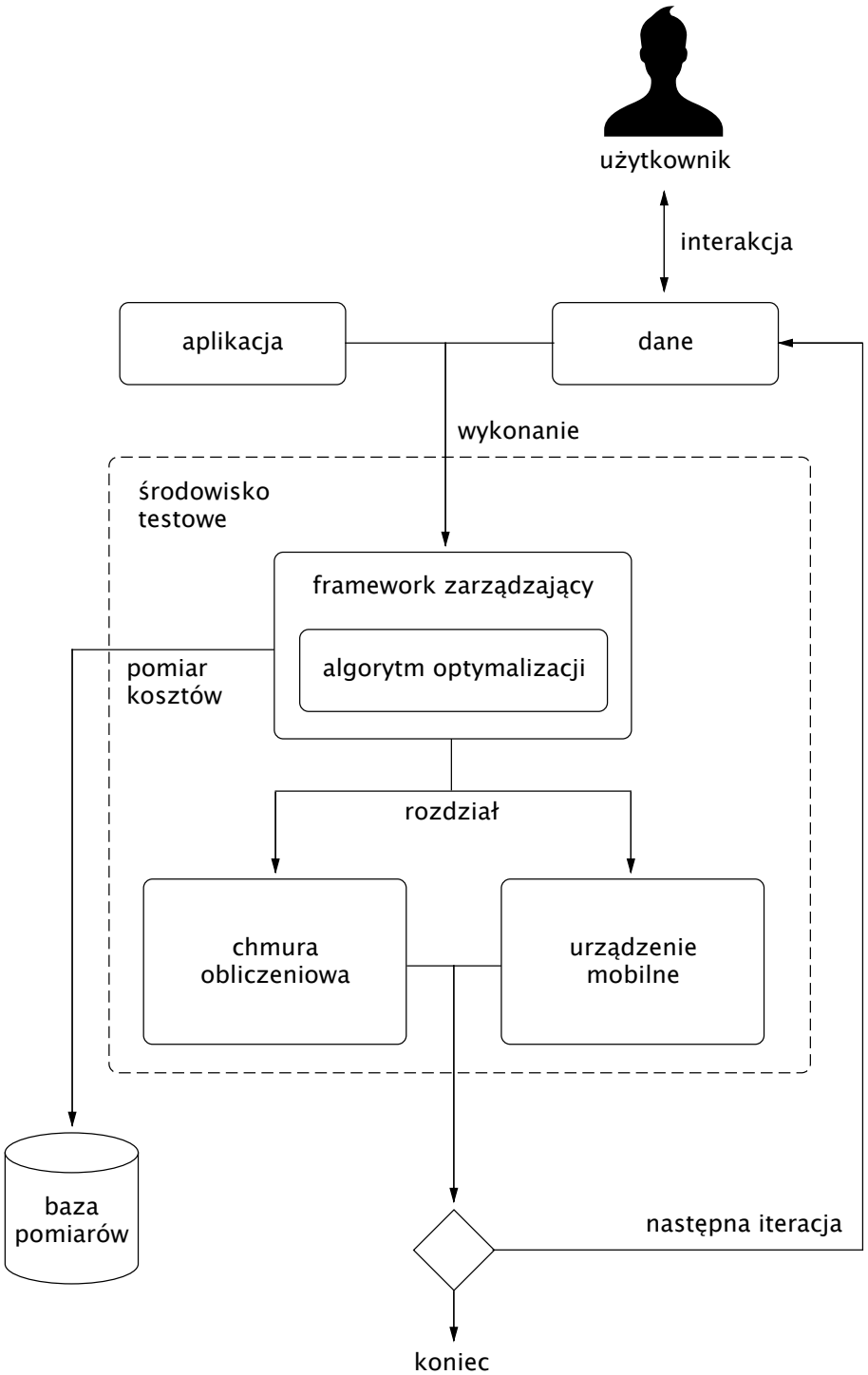
Do testów empirycznych wybrano dwie aplikacje: grę w szachy oraz aplikację do monitorowania parkingu. Gra w szachy jest typem aplikacji interaktywnej, w której liczba iteracji nie jest łatwa do przewidzenia, ponieważ zależy od niedeterministycznego zachowania użytkownika. Zmienność kosztów wykonania operatorów nie jest duża i jest dość przewidywalna – jedyny operator o wysokiej zmienności oblicza ruch komputera i zależy wprost od liczby możliwych ruchów.

Aplikacja do monitorowania parkingu to przykład aplikacji mającej zastosowanie na urządzeniach IoT. Liczba iteracji może być nieograniczona, ponieważ parking może być monitorowany w sposób ciągły przez wiele dni. Zmienność kosztów wykonania jest natomiast duża – w zależności od natężenia światła (dzień i noc) i szumu w analizowanym strumieniu wideo algorytm przetwarzania obrazu może zmienia swoją charakterystykę. Algorytm wykrywania obiektów również ma zmienną złożoność w zależności od liczby potencjalnych obiektów.

Obie aplikacje testowane były według podobnego scenariusza przedstawionego na rysunku 5.1. Użytkownik przeprowadza interakcje z aplikacją odbierając i wprowadzając dane. W przypadku gry w szachy dane to ruchu gracza i widok szachownicy. Dla aplikacji monitoringu danymi będą strumienie wideo z kamer i wyniki analizy obrazu. Aplikacja wykonywana jest za pośrednictwem frameworka zarządzające, który wykorzystując jeden z opisanych algorytmów rozdziela operatora pomiędzy chmurę obliczeniową i urządzenie. Framework nadzoruje wykonanie każdego operatora i zapisuje szacowane koszty w bazie pomiarów. Po zakończeniu iteracji następuję zakończenie działania aplikacji lub przejście do następnej iteracji.

### 5.1.1 Framework (szkielet) aplikacji i moduł zarządzający

W celu przetestowania obu modeli oraz algorytmów przygotowany został framework MOFF (*ang. Mobile Offloading Framework*) umożliwiający przeniesienie wykonania kodu aplikacji pomiędzy środowiskami chmury obliczeniowej i urządzenia mobilnego. Framework opakowuje główną funkcję aplikacji i przechwytuje wywoła-




Rysunek 5.1: Scenariusz testowania

nia wszystkich pozostałych wewnętrznych funkcji. Dzięki temu możliwe jest monitorowanie poszczególnych operatorów, tzn. obliczenie czasu wywołania oraz rozmiaru

danych wejściowych i wyjściowych. Ponadto framework może przechwycić wywołanie lokalnej funkcji i w jej miejsce uruchomić funkcję w chmurze, realizując tym samym *offloading* funkcji do chmury obliczeniowej.

Naturalnie, aby umożliwić działanie frameworka, należy poczynić pewne założenia odnośnie struktury aplikacji i sposobu jej implementacji:

1. Funkcje muszą otrzymywać wszystkie dane w postaci argumentów.
2. Funkcje muszą zwracać wynik w postaci instrukcji wyjścia (`return`).
3. Funkcje powinny być bezstanowe lub korzystać z mechanizmu pamiętania staniu dostarczanego przez framework.

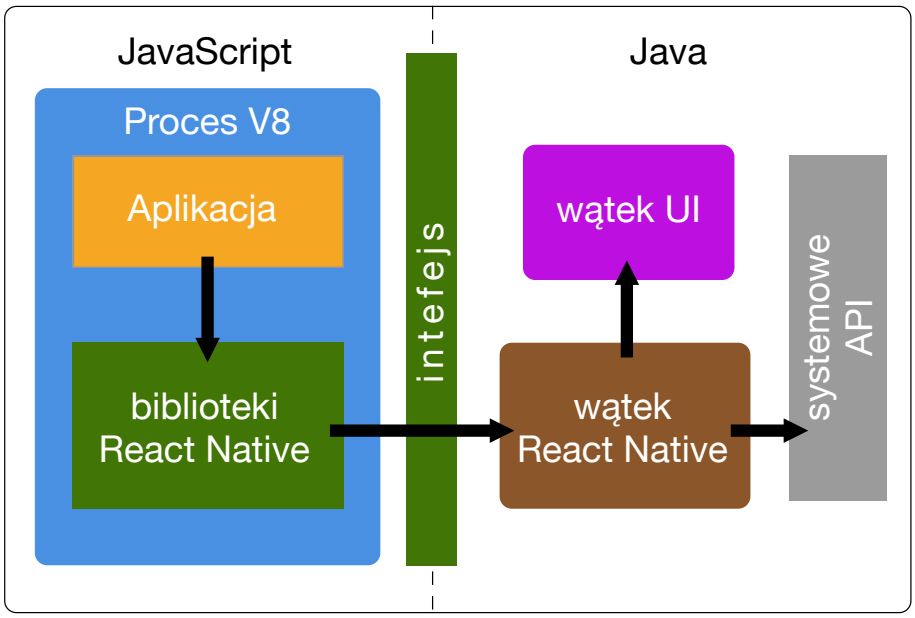
Wymagania te są spełnione kiedy aplikacja jest zaimplementowana zgodnie z paradygmatem programowania funkcyjno-reaktywnego (FRP). Po uwzględnieniu powyższych kryteriów, a także uwzględniając ograniczenia środowiska urządzeń mobilnych, podjęto decyzję o implementacji frameworka oraz aplikacji testowych w języku JavaScript.

JavaScript (JS) jest wieloparadygmatowym skryptowym językiem programowania, który umożliwia tworzenie aplikacji m.in. w paradygmacie funkcyjnym. Jest to aktualnie najpopularniejszy języka programowania przeznaczony głównie dla aplikacji webowych. Praktycznie wszystkie szeroko dostępne urządzenia mobilne, takie jak smartfony czy tablety posiadają przeglądarki internetowe wyposażone w interpreter języka JavaScript.

Ze względu na swoją popularność w aplikacjach webowych, JavaScript szybko znalazł zastosowanie w innych kategoriach aplikacji, m.in. w aplikacjach mobilnych. Istnieje wiele narzędzi ułatwiających implementację aplikacji dla systemów Android oraz iOS, takich jak Apache Cordova [29] czy Adobe PhoneGap [39], które opierają się na wykorzystaniu interpretera JavaScript oraz silnika renderowania przeglądarki systemowej. Z punktu widzenia programisty jest to najprostsze rozwiązanie, ponieważ nie różni się właściwie od sposobu implementacji aplikacji webowych. Minusem tych narzędzi jest często dużo niższa wydajność wynikowych aplikacji w stosunku do aplikacji pisanych natywnie, tzn. w języku Java [9] lub Objective C [48].

Innym podejściem są rozwiązania typu React Native [76] lub Nativescript [2], które wykorzystują bardzo wydajny interpreter V8 [34] oraz natywne implementacje renderowania kontrolek interfejsu użytkownika. Pozwala to na tworzenie aplikacji,

których logika przetwarzania zaimplementowana jest w języku JavaScript, a wydajnościowo porównywalne są z aplikacjami natywnymi. Schemat architektury aplikacji z wykorzystaniem React Native został przedstawiony na rys. 5.2. Framework został zaprojektowany w taki sposób aby mógł być wykorzystany w obu przypadkach.



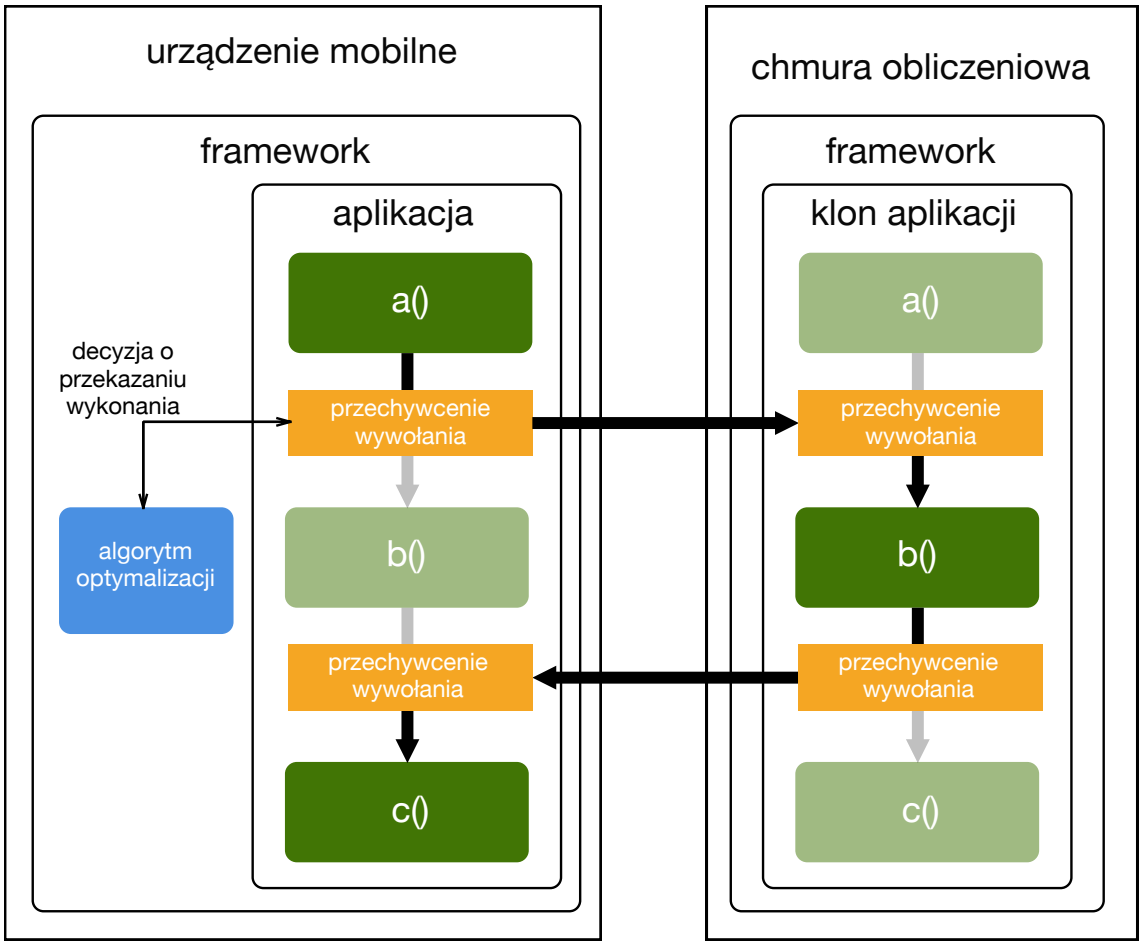

Rysunek 5.2: Architektura React Native w systemie Android [82]

W celu ułatwienia implementacji aplikacji z graficznym interfejsem użytkownika w paradygmacie funkcyjno-reaktywnym wykorzystana została biblioteka CycleJS [93]. Umożliwia ona implementację całej logiki aplikacji w postaci bezstanowych funkcji i obsługuje zarządzanie stanem, a także renderowanie interfejsu np. przy użyciu biblioteki React Native.

Do uruchomienia aplikacji w środowisku chmury obliczeniowej wykorzystana została platforma NodeJS [21]. Po stronie chmury działa moduł frameworka odpowiedzialny za nadzorowanie uruchomienia poszczególnych funkcji aplikacji. Moduł ten komunikuje się z frameworkiem na urządzeniu mobilnym za pomocą protokołu WebSocket [28]. W środowisku chmury uruchamiany jest dokładnie ten sam kod źródłowy co na urządzeniu mobilnym, co przede wszystkim jest ułatwieniem dla programisty, a także pozwala na wyeliminowanie błędów związanych z innym zachowaniem aplikacji przy przekazaniu wykonania do chmury.

W celu zapewnienia jak najniższych opóźnień podczas przełączania pomiędzy funkcją w chmurze i na urządzeniu, podczas uruchomienia aplikacji na urządzeniu

równocześnie uruchamiania jest kopia aplikacji w środowisku chmury obliczeniowej. Kopia aplikacji nie wykonuje żadnych obliczeń dopóki nie otrzyma danych wejściowych od aplikacji uruchomionej na urządzeniu mobilnym. Na rysunku 5.3 został przedstawiony schemat działania frameworka.




Rysunek 5.3: Schemat działania frameworka przełączania operatorów

## 5.2 Budowa środowiska testowego

W ramach środowiska testowego można wyróżnić dwa systemy: chmurę obliczeniową i urządzenie mobilne. Na potrzeby testów chmura obliczeniowa została uproszczona do jednej maszyny, na której uruchomiony był moduł frameworka odpowiedzialny za uruchamianie funkcji aplikacji.

### 5.2.1 Chmura obliczeniowa

Jako przykładową chmurę obliczeniową wykorzystano chmurę opartą o środowisko OpenStack [88], wdrożoną w Centrum Informatycznym Trójmiejskiej Akademickiej Sieci Komputerowej (CI TASK) [96] na Politechnice Gdańskiej. Chmura działa na klastrze superkomputerowym złożonym ze 124 węzłów obliczeniowych, wyposażonych w dwa procesory Intel Xeon E5 v3 @ 2,3 GHz każdy, połączonych siecią InfiniBand o przepustowości 56 Gb/s.

### 5.2.2 Emulator

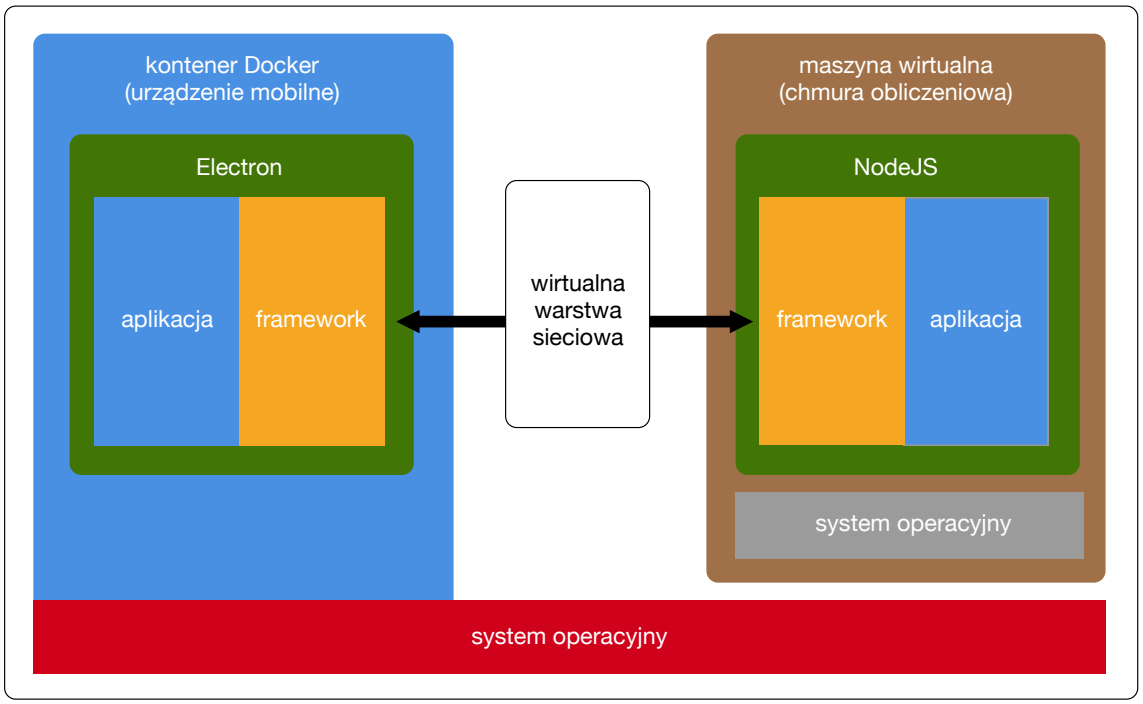
W związku z tym, że testowane aplikacje oraz framework zaimplementowane są w języku JavaScript do emulowania środowiska urządzenia mobilnego wykorzystana została platforma Electron [32]. Pozwala ona na uruchamianie aplikacji JavaScript jako aplikacji desktopowych, odpowiadając funkcjonalnością platformom Apache Cordova i Adobe PhoneGap na urządzeniach mobilnych.

Aplikacja uruchamiana jest w kontenerze Docker [69], w celu separacji środowiska emulatora od środowiska komputera hosta. Kontenery pozwalają na wirtualizację na poziomie systemu operacyjnego, tzn. izolację przestrzeni procesów, przestrzeni użytkownika oraz warstwy sieciowej, przy współdzieleniu jądra systemu z systemem hosta. Dzięki uruchamianiu emulatora w kontenerze możliwe jest sterowanie liczbą dostępnych cykli CPU, dzięki czemu można przetestować zachowanie się frameworka dla różnych stosunków wydajności emulowanego procesora mobilnego do procesora w chmurze obliczeniowej, a także sterowanie opóźnieniem komunikacji sieci. Architektura emulatora została przedstawiona na rysunku 5.4.

## 5.3 Realizowane eksperymenty

Badania zostały przeprowadzone dla różnych konfiguracji środowiska testowego. Przez konfigurację rozumiany jest zestaw następujących parametrów:

1. Wydajność urządzenia $W$ wyrażona jako stosunek wydajności przetwarzania symulowanego urządzenia do wydajności przetwarzania chmury obliczeniowej, np. $W = 0.5$ oznacza dwa razy wolniejsze przetwarzanie na urządzeniu niż w chmurze.




Rysunek 5.4: Architektura emulatora

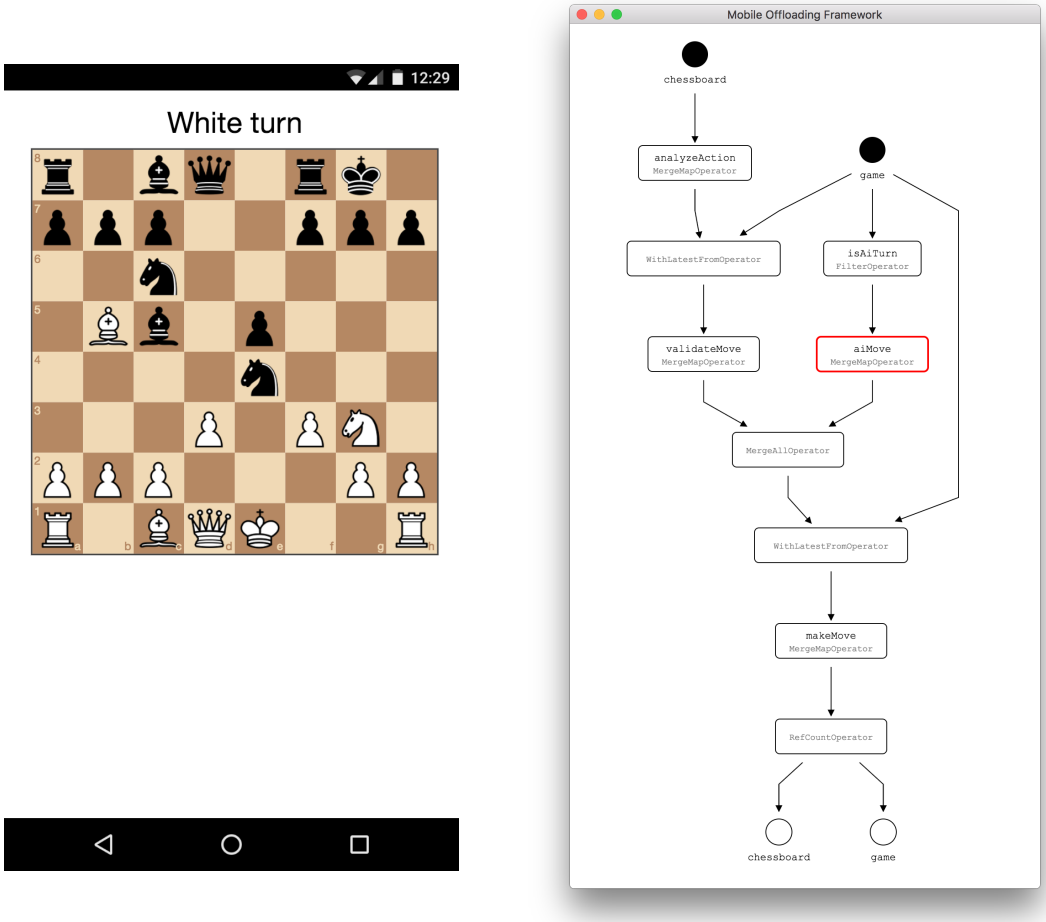
2. Szybkość sieci, na którą składa się przepustowość oraz opóźnienie. Badania wykonano dla dwóch symulowanych szybkości sieci: 3G (opóźnienie 200ms, przepustowość 5Mbps) oraz WiFi (opóźnienie 100ms, przepustowość 20Mbps).

Dla uproszczenia analizy przyjęto, że wartości współczynników $P_m$, $P_c$, $P_e$ są równie 1. Oznacza to, że zmierzone koszty wykonania iteracji można interpretować jako czas wykorzystania CPU i czas transferu sieciowego wyrażony w milisekundach.

Przed wykonaniem testów obie badane aplikacje zostały uruchomione z pełnym zestawem danych we wszystkich konfiguracjach środowiska w celu zmierzenia parametrów wykonania: czasów wykonania operatorów i czasów transmisji danych. Następnie obliczone zostały średnie koszty wykonania operatorów i średnie koszty komunikacji, które wykorzystywane były podczas testów do optymalizacji rozdziału przez algorytm URO. Algorytm PWO wykorzystywał do optymalizacji dane zbierane dynamicznie podczas właściwego wykonania aplikacji (testu) i przyjęto, że początkowo wszystkie operatory uruchomione są na urządzeniu mobilnym.

Wszystkie eksperymenty zostały przeprowadzone z wykorzystaniem zaproponowanego frameworka zarządzającego wykonaniem aplikacji. Rysunek 5.5 przedstawia zrzut ekranu z aplikacji do gry w szachy uruchomionej na emulatorze urządzenia mo-

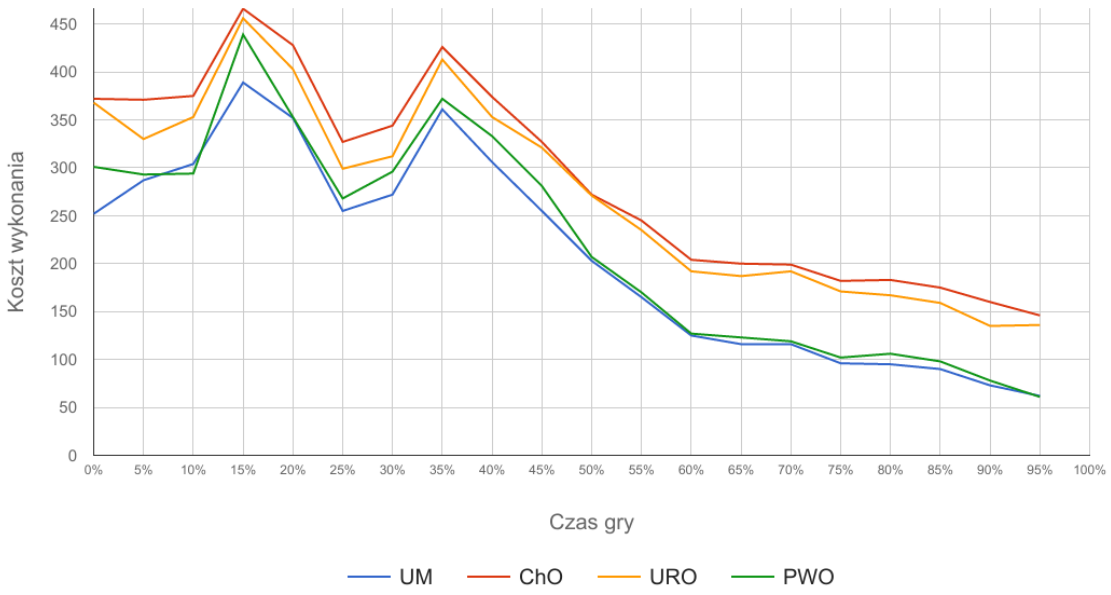
bilnego oraz podgląd modułu monitorującego wykonanie. Na podglądzie widoczny jest model aplikacji, w którym wyszczególnione są wszystkie operatory obsługujące logikę wykonania. Operator aktualnie wykonywany w chmurze obliczeniowej oznaczony jest czerwoną ramką.




Rysunek 5.5: Zrzut ekranu z aplikacji do gry w szachy oraz podgląd wykonania

### 5.3.1 Gra w szachy

Do zbadania aplikacji gry w szachy przygotowanych zostało 20 zestawów danych wejściowych, tzn. strumieni interakcji użytkownika. Wyniki testów zostały uśrednione ze wszystkich gier. Ponieważ czas trwania (liczba iteracji) każdej gry może być inny, wyniki zostały znormalizowane i na wykresach przedstawiono koszty iteracji względem stopnia zaawansowania gry – od pierwszej (0%) do ostatniej iteracji



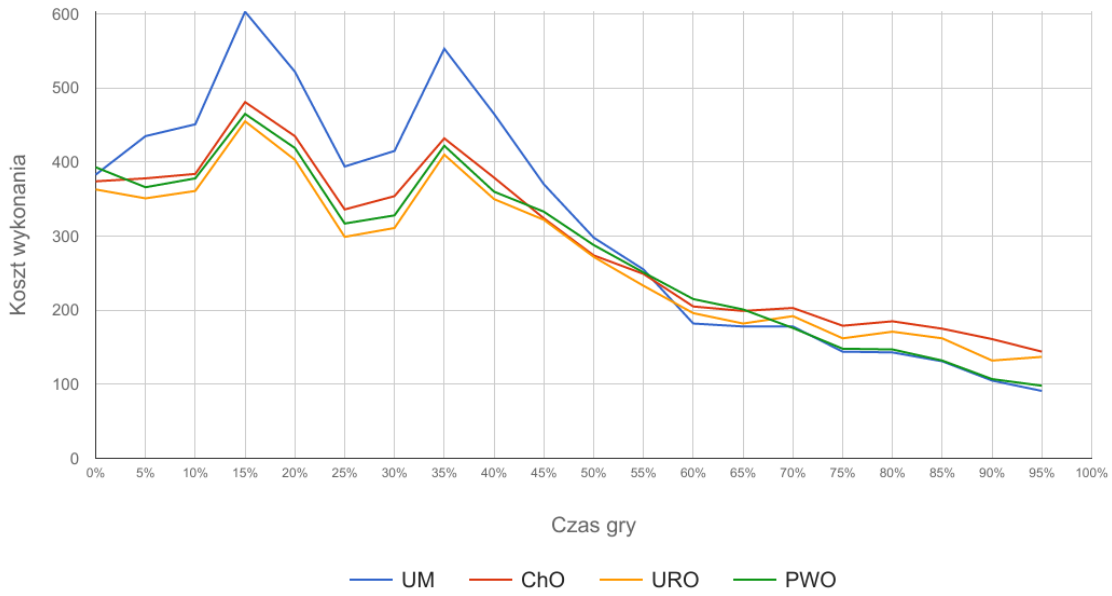

Rysunek 5.6: Wyniki testów dla $W = 1$ i sieci WiFi

(100%).

Na rysunku 5.6 przedstawiono koszty iteracji dla urządzenia o wydajności równej wydajności chmury obliczeniowej i szybkiej sieci WiFi. Pierwszą obserwacją jest charakterystyczny przebieg wykresu, który powtarza się we wszystkich konfiguracjach środowiska testowego. W ogólności późniejsze iteracje wykonują się szybciej niż początkowe, co jest skutkiem mniejszej liczby figur na szachownicy w końcowej fazie gry – przeszukanie drzewa możliwych ruchów jest więc dużo szybsze. Zauważalne są też dwa lokalne ekstrema, około 15% oraz 35% zaawansowania gry, które związane są z charakterystyką wykorzystywanej implementacji silnika do gry w szachy [98].

W tym wypadku przenoszenie wykonania jakiegokolwiek operatora do chmury nie jest opłacalne, ponieważ wydajność urządzenia i chmury jest taka sama, a przeniesienie wiązałoby się z dodatkowym kosztem komunikacji. Algorytm URO operuje na danych uśrednionych z wszystkich konfiguracji, a zatem mimo wszystko przenosi część operatorów do chmury. Algorytm PWO w tej sytuacji zachowuje się dużo lepiej i pozostawia wszystkie operatory na urządzeniu.

W sytuacji gdy urządzenie mobilne jest o jedną trzecią mniej wydajne niż chmura, a sieć jest relatywnie szybka, wykonywanie operatorów w chmurze ma sens w pierwszej połowie gry, jak widać na wykresie 5.7. Różnica w kosztach iteracji jest jednak dość niewielka, dlatego zarówno algorytm URO jak i PWO zachowują się podob-



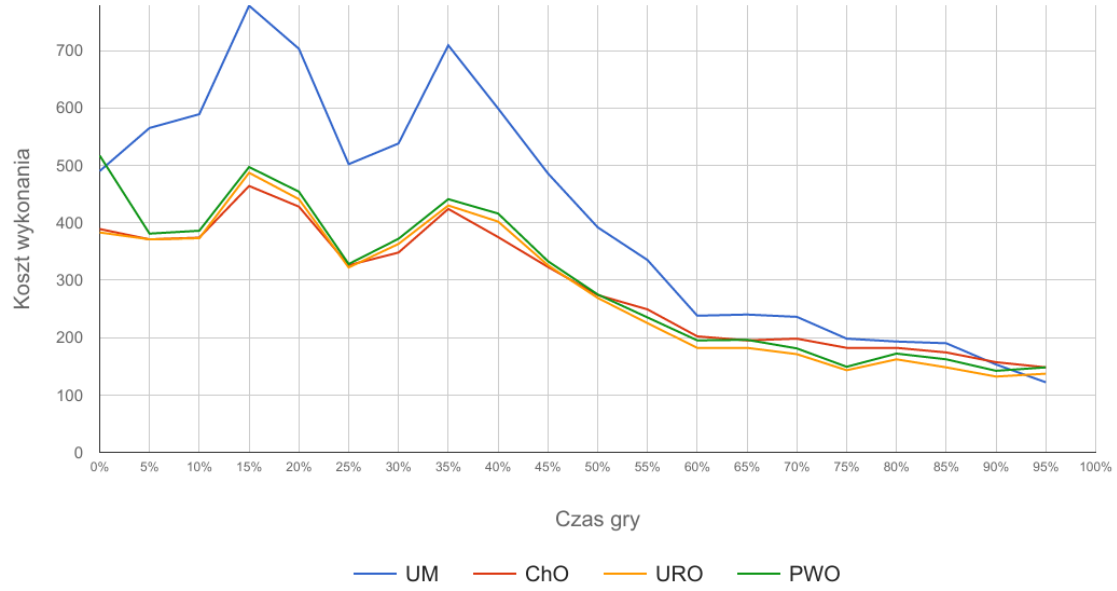

Rysunek 5.7: Wyniki testów dla $W = 0.67$ i sieci WiFi

nie. Można zaobserwować jednak nieznaczną przewagę algorytmu PWO w końcowej fazie działania aplikacji, kiedy opłacalna jest migracja operatorów z powrotem na urządzenie mobilne.

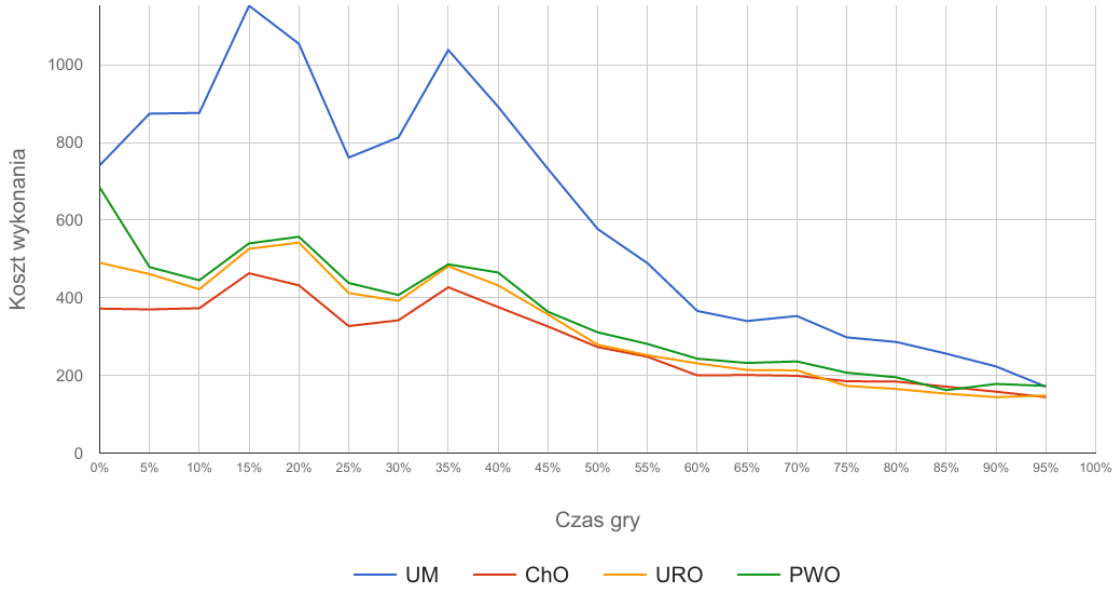
Na rysunkach 5.8 i 5.9 przedstawione zostały wykresy dla konfiguracji środowiska o niskiej wydajności urządzenia mobilnego, odpowiednio $W = 0.5$ i $W = 0.33$. Widać, że w przypadku gdy urządzenie jest dwa lub więcej razy wolniejsze niż chmura i mamy do czynienia z szybką siecią, optymalnym rozwiązaniem jest uruchomienie całej aplikacji w chmurze obliczeniowej. Algorytmy URO i PWO dobrze sprawdzają się w takiej sytuacji i koszty wykonania kolejnych iteracji są zgodne z kosztami uruchomienia aplikacji w chmurze.

Wyniki testów dla wolniejszej sieci 3G i równej wydajności chmury oraz urządzenia są bardzo podobne jak dla wcześniej przedstawionej konfiguracji z siecią WiFi. Na wykresie 5.10 widać wyraźnie, że przenoszenie wykonania do chmury nie jest opłacalne, ponieważ koszty komunikacji są zbyt wysokie.

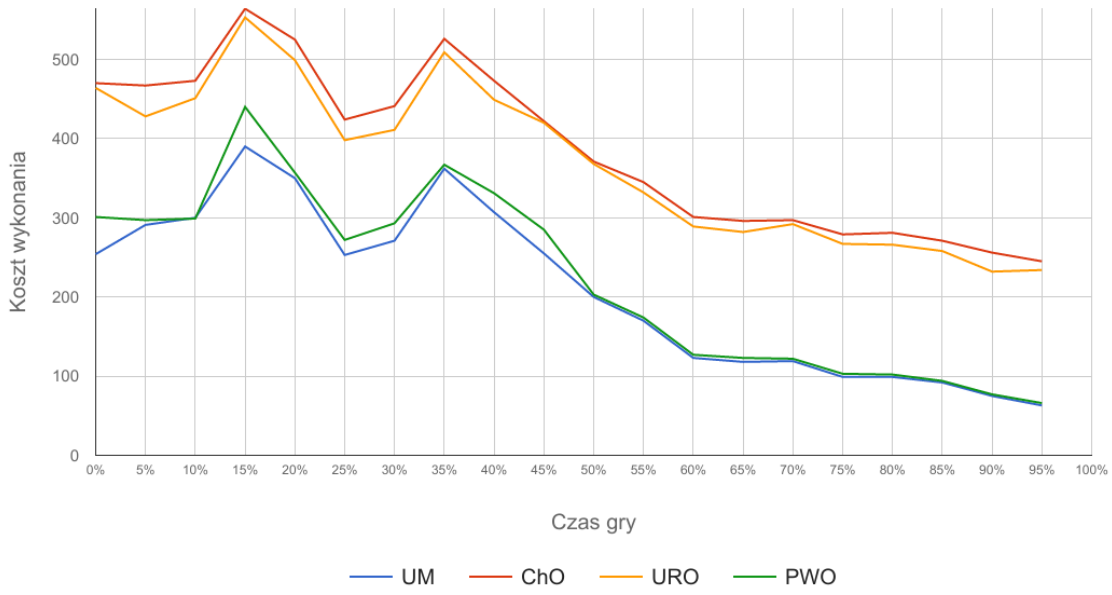
Także przy $W = 0.67$ koszt wykonania iteracji na urządzeniu mobilnym jest generalnie niższy. Wykres dla tej konfiguracji został przedstawiony na rysunku 5.11. Jedynie w początkowej fazie gry, gdy czas obliczenia następnego ruchu jest wysoki, niższe koszty wykonania operatora w chmurze obliczeniowej rekompensują wysokie koszty komunikacji. Algorytm PWO lepiej rozdziela operatory pod koniec gry, kiedy




Rysunek 5.8: Wyniki testów dla $W = 0.5$ i sieci WiFi

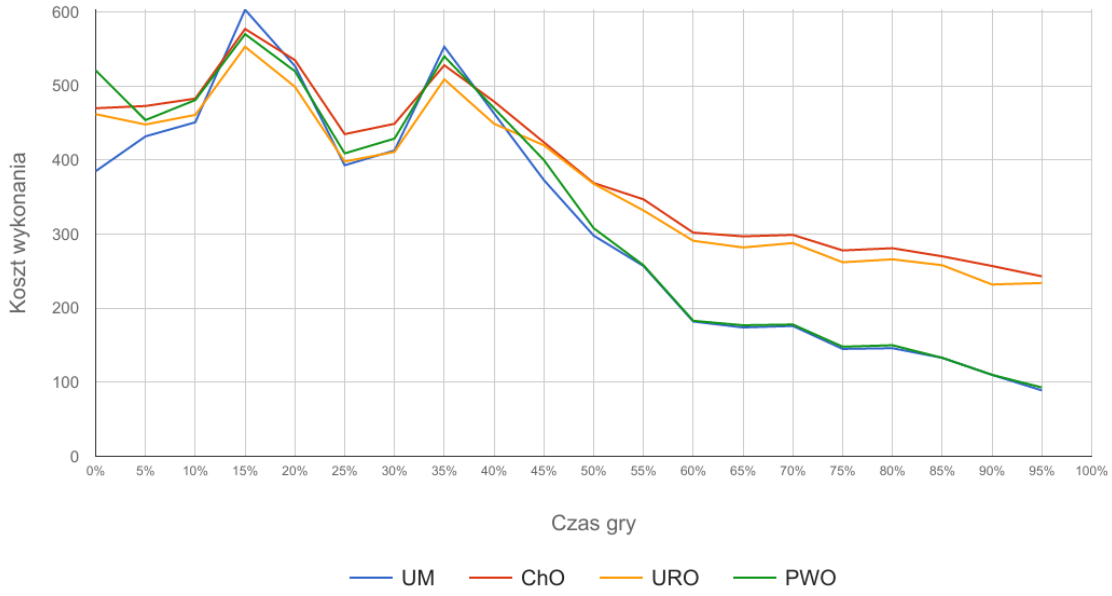



Rysunek 5.9: Wyniki testów dla $W = 0.33$ i sieci WiFi




Rysunek 5.10: Wyniki testów dla $W = 1$ i sieci 3G

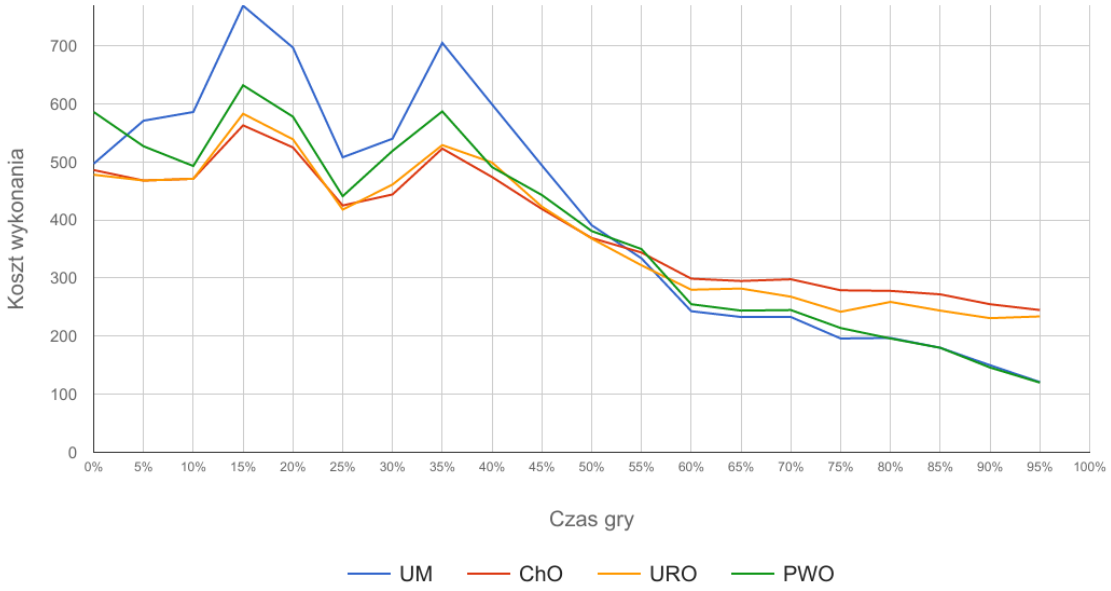
koszty wykonania na urządzeniu są relatywnie niskie.




Rysunek 5.11: Wyniki testów dla $W = 0.67$ i sieci 3G

Wykres 5.12 przedstawia wyniki testów dla konfiguracji środowiska, w której chmura jest dwa razy bardziej wydajna niż urządzenie mobilne. Bardzo dobrze

widoczne są tutaj zalety algorytmu PWO, który mniej więcej do połowy gry rozdziela operatory do chmury obliczeniowej, a w drugiej połowie gry migruje je z powrotem na urządzenie. Tym samym koszty iteracji są na każdym etapie gry bliskie optymalnym.

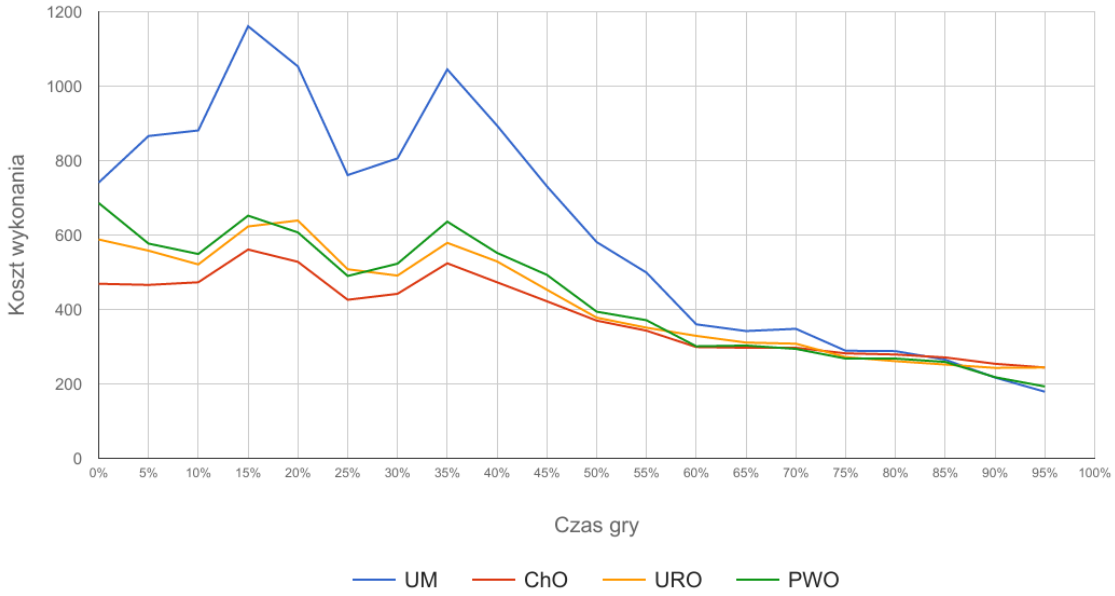



Rysunek 5.12: Wyniki testów dla $W = 0.5$ i sieci 3G

W przypadku $W = 0.33$ przedstawionym na rysunku 5.13 również widoczny jest moment, w którym migracja wykonania operatorów na urządzenie mobilne staje się opłacalna. Następuje to jednak już w końcowej fazie działania aplikacji, stąd różnice w kosztach nie są tutaj już tak duże, a skuteczność algorytmów URO i PWO jest bardzo podobna.

Na rysunku 5.14 i w tabeli 5.1 przedstawiono porównanie całkowitych kosztów dla wszystkich konfiguracji środowiska testowego. Można zauważyć, że z wyjątkiem konfiguracji z urządzeniem o bardzo niskiej wydajności w stosunku do chmury, algorytm PWO jest bardzo blisko rozwiązania optymalnego, bez względu na to czy jest to uruchomienie aplikacji w chmurze obliczeniowej, na urządzeniu czy rozdział operatorów pomiędzy chmurę i urządzenie. W konfiguracji z $W = 0.5$ i siecią 3G algorytm PWO osiąga lepszy rezultat niż którykolwiek inny sposób rozdziału aplikacji. Algorytm URO natomiast ma bardzo nierówne wyniki – w pewnych sytuacjach osiąga najniższy koszt całkowity (szczególnie w przypadku szybkiej sieci), a w innych generuje zupełnie nieoptymalne rozdziały (wysoka wydajność urządzenia).

Ciekawe wnioski wyciągnąć można z uśrednionych kosztów po wszystkich konfi-



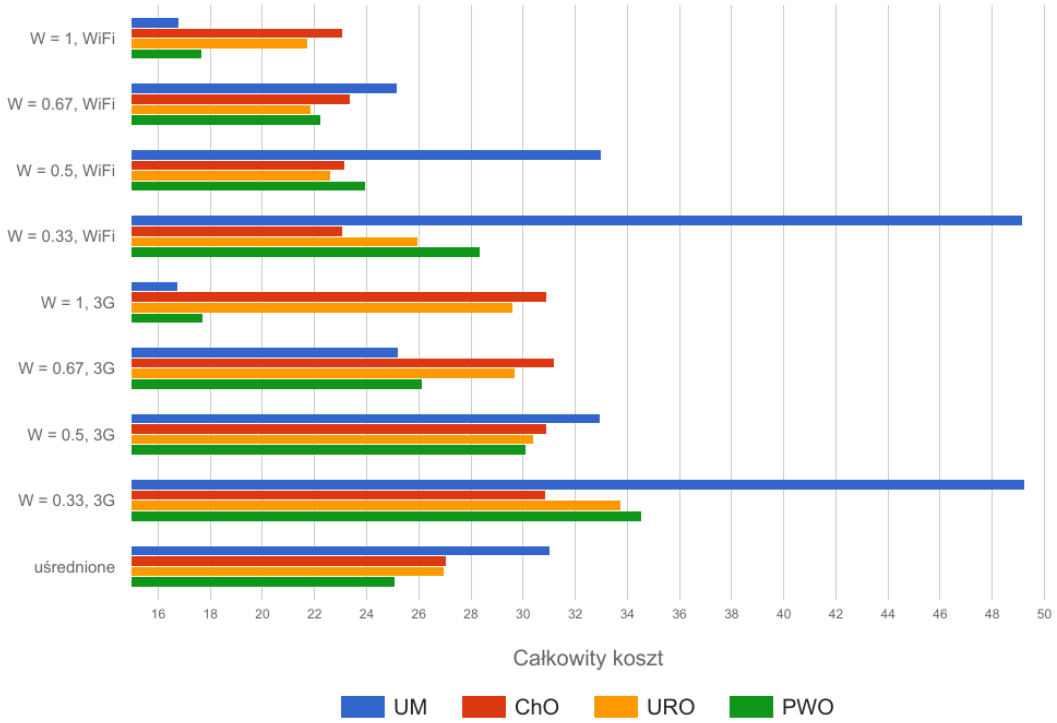

Rysunek 5.13: Wyniki testów dla $W = 0.33$ i sieci 3G

guracjach środowiska. W tym wypadku algorytm PWO jest zdecydowanie najlepszym rozwiązaniem. Średni koszt jest o ponad 7% niższy niż w przypadku algorytmu URO i aż o 19% niższy niż w przypadku uruchomienia aplikacji wyłącznie na urządzeniu mobilnym. Na rynku dostępne są urządzenia o bardzo zróżnicowanej wydajności i wykorzystywane są sieci bezprzewodowe o różnych charakterystykach, a więc w praktycznym zastosowaniu w aplikacji dostępnej dla szerokiej grupy użytkowników, wykorzystanie algorytmu PWO pozwalałoby na uzyskanie najniższych kosztów wykonania.

### 5.3.2 Monitorowanie parkingu

Aplikacja do monitorowania parkingu została zaimplementowana z wykorzystaniem biblioteki Tracking.js [24]. Dla uproszczenia testy aplikacji zostały przeprowadzone na jednym strumieniu testowym złożonym z 720 klatek wideo o rozdzielczości 640x480 pikseli. Przygotowany strumień jest przyspieszonym nagraniem z pełnej doby monitoringu, wybrane zostały więc klatki w 2-minutowych odstępach. Każda klatka strumienia wideo jest przetwarzana w ramach jednej iteracji.

Na rysunku 5.15 przedstawiono koszty iteracji dla urządzenia o wydajności równej wydajności chmury obliczeniowej i szybkiej sieci WiFi. Z przebiegu wykresu



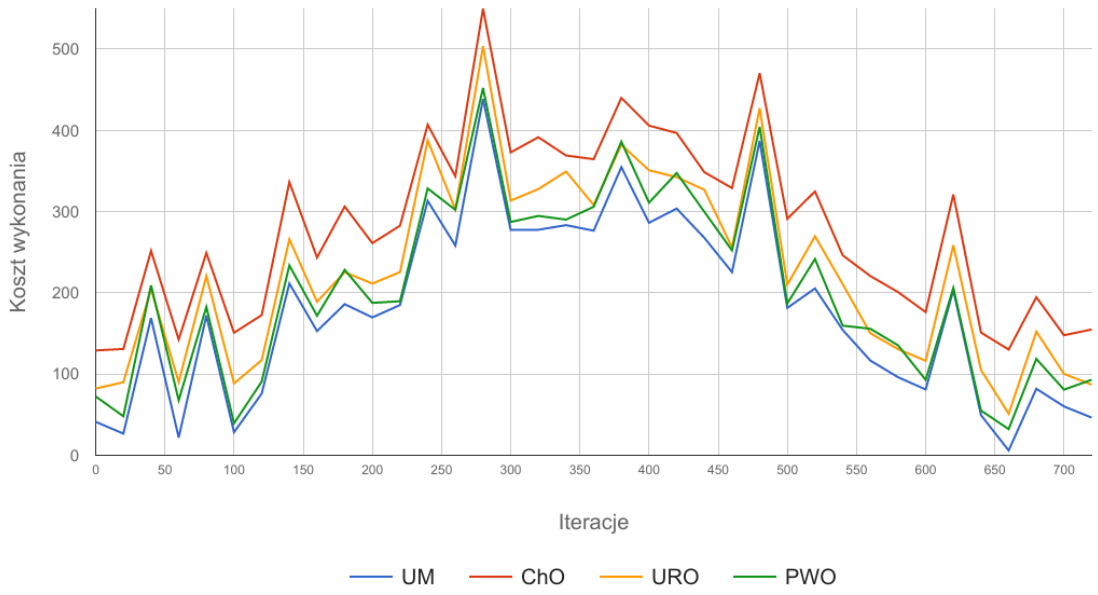

Rysunek 5.14: Całkowite koszty dla wszystkich konfiguracji środowiska testowego

Tabela 5.1: Całkowite koszty dla wszystkich konfiguracji środowiska testowego

| | | całkowity koszt | | | |
|---|---|---|---|---|---|
| **W** | **sieć** | **UM** | **ChO** | **URO** | **PWO** |
| 1 | WiFi | 16,82 | 23,11 | 21,78 | 17,8 |
| 0,67 | WiFi | 25,2 | 23,37 | 21,86 | 22,17 |
| 0,5 | WiFi | 33,09 | 23,12 | 22,6 | 23,98 |
| 0,33 | WiFi | 49,25 | 23,13 | 25,97 | 28,16 |
| 1 | 3G | 16,76 | 30,91 | 29,61 | 17,73 |
| 0,67 | 3G | 25,21 | 31,18 | 29,69 | 26,13 |
| 0,5 | 3G | 32,98 | 30,93 | 30,4 | 30,11 |
| 0,33 | 3G | 49,23 | 30,88 | 33,75 | 34,54 |
| | średnia | 31,07 | 27,08 | 26,96 | 25,08 |

wyraźnie widać, że koszty wykonania są znacznie wyższe w iteracjach 200-500. Jest to spowodowane wyższą złożonością przetwarzania klatek nagranych w nocy, w gorszych warunkach oświetleniowych. Można też zaobserwować lokalne ekstrema związane prawdopodobnie z większą liczbą obiektów znajdujących się na obrazie.

W przypadku gdy $W = 1$ przenoszenie wykonania do chmury obliczeniowej nie



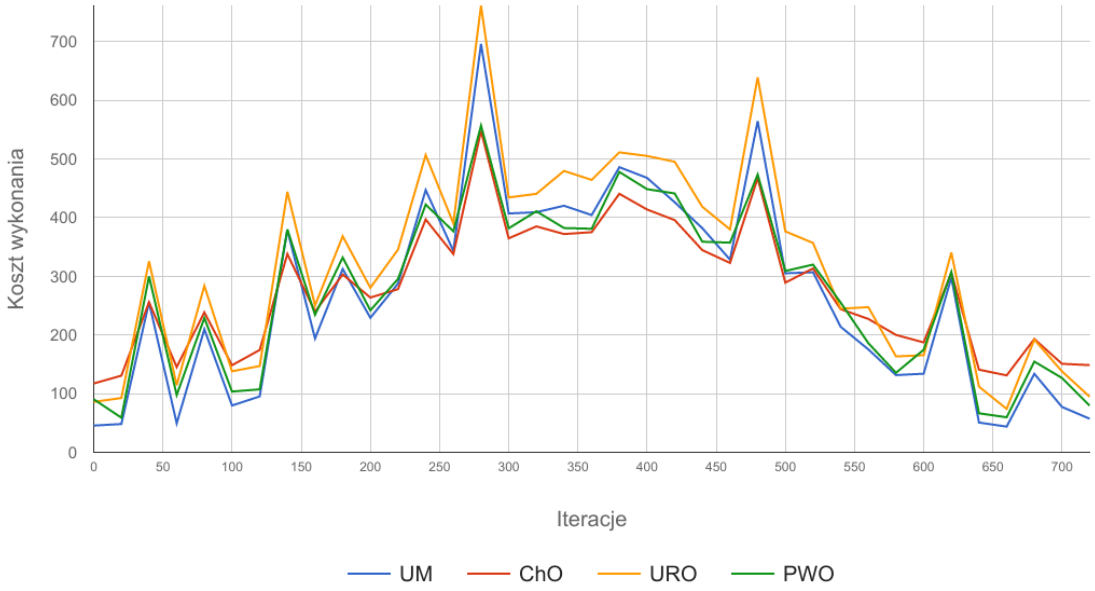

Rysunek 5.15: Wyniki testów dla $W = 1$ i sieci WiFi

jest opłacalne – koszty przesłania strumienia są zbyt duże. Nieco inaczej wygląda sytuacja dla $W = 0.67$ przedstawiona na wykresie 5.16. Koszty wykonania na urządzeniu i w chmurze są bardzo podobne ze względu na nieco niższą wydajność urządzenia. W obu przypadkach algorytmy rozdziału operatorów URO i PWO zachowują się podobnie, algorytm PWO daje jednak wyniki nieco bliższe optymalnym.

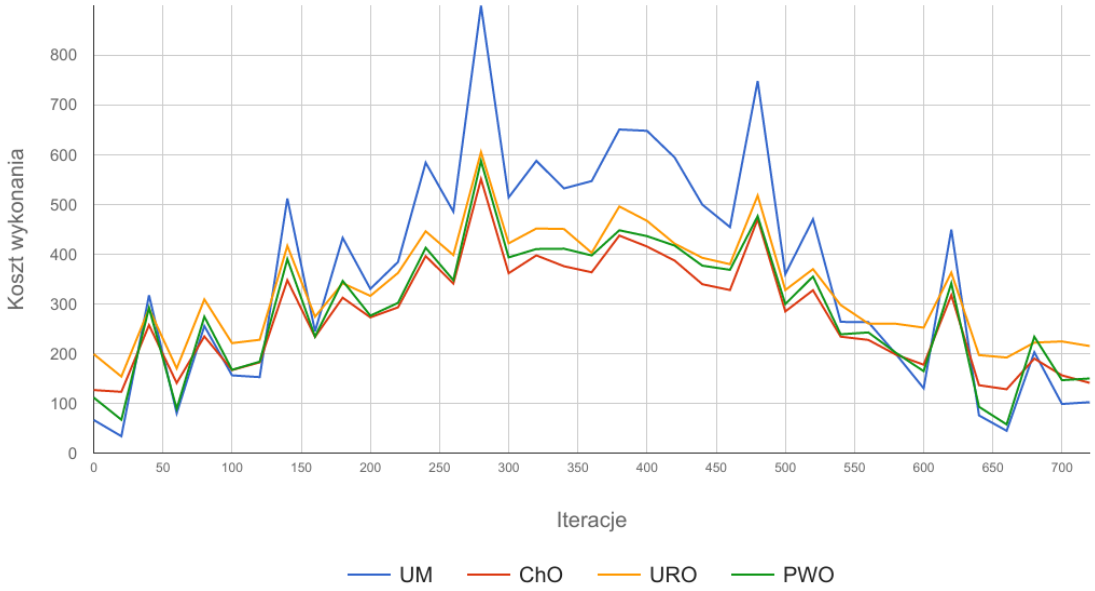
Na rysunku 5.17 przedstawiono wyniki dla urządzenia dwa razy mniej wydajnego od wykorzystywanej chmury obliczeniowej. Dzięki szybkiej sieci WiFi, koszty wykonania w środkowej fazie działania aplikacji są dużo niższe po przeniesieniu wykonania operatorów do chmury. Algorytm PWO bardzo dobrze reaguje w tej sytuacji i przenosi operatory do chmury około 130 iteracji, a następnie z powrotem na urządzenie po iteracji 600.

Wyniki testów dla konfiguracji środowiska testowego z urządzeniem o stosunku wydajności $W = 0.33$ zostały przedstawione na rysunku 5.18. W tym przypadku koszty uruchomienia w chmurze obliczeniowej są niższe właściwie przez cały czas działania aplikacji. Oba algorytmy rozdziału zachowują się bardzo podobnie – przebiegi funkcji kosztów są bliskie kosztom wykonania na chmurze.

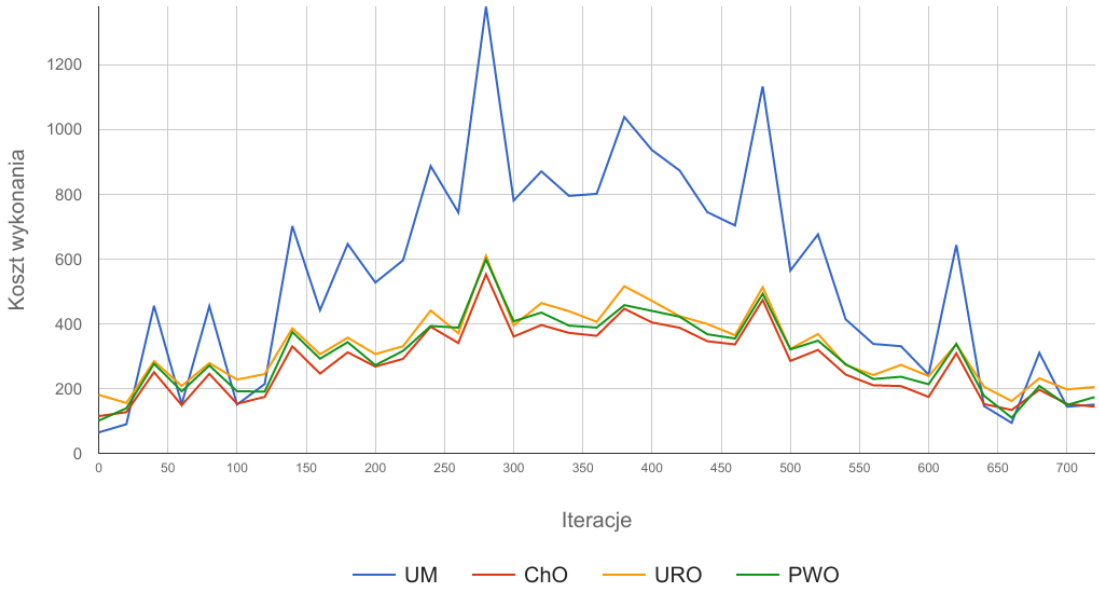
Na rysunku 5.19 przedstawione zostały wyniki dla konfiguracji środowiska z wolniejszą siecią 3G i urządzeniem o wydajności równej wydajności chmury. W takiej sytuacji niższe koszty wykonania aplikacji uzyskujemy przy uruchomieniu wszystkich




Rysunek 5.16: Wyniki testów dla $W = 0.67$ i sieci WiFi




Rysunek 5.17: Wyniki testów dla $W = 0.5$ i sieci WiFi




Rysunek 5.18: Wyniki testów dla $W = 0.33$ i sieci WiFi

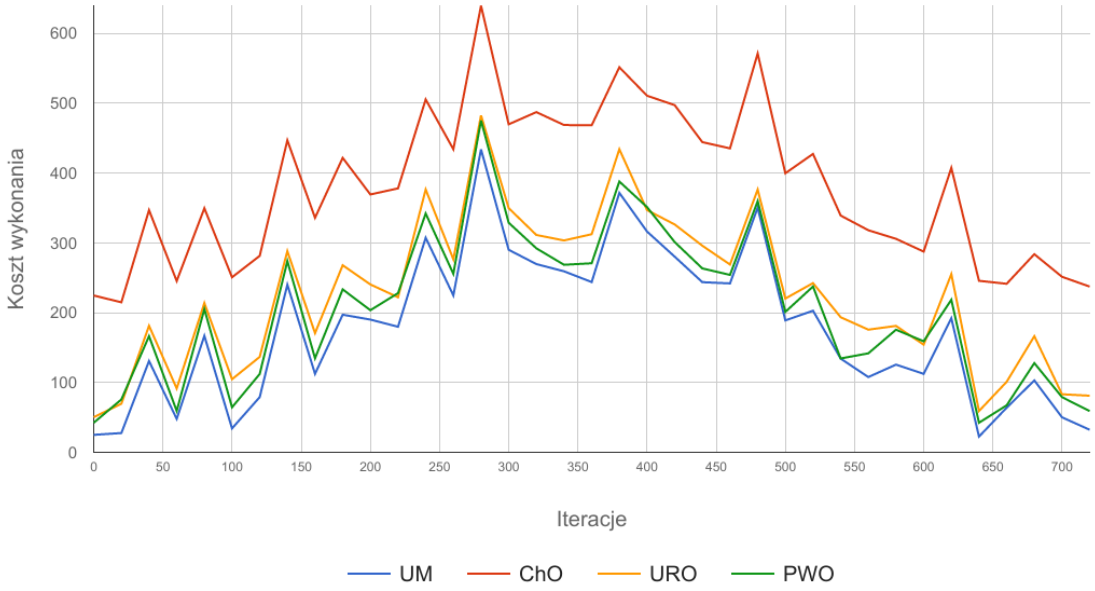
operatorów na urządzeniu, algorytmy rozdziału zachowują się zgodnie z oczekiwaniem i nie przenoszą żadnych operatorów do chmury.

Niewiele zmienia się w konfiguracji z nieco wolniejszym urządzeniem, której wyniki przedstawiono na rysunku 5.20. Narzut kosztów związanych z transferem strumienia jest nadal zbyt wysoki i wykorzystanie chmury obliczeniowej nie jest opłacalne.

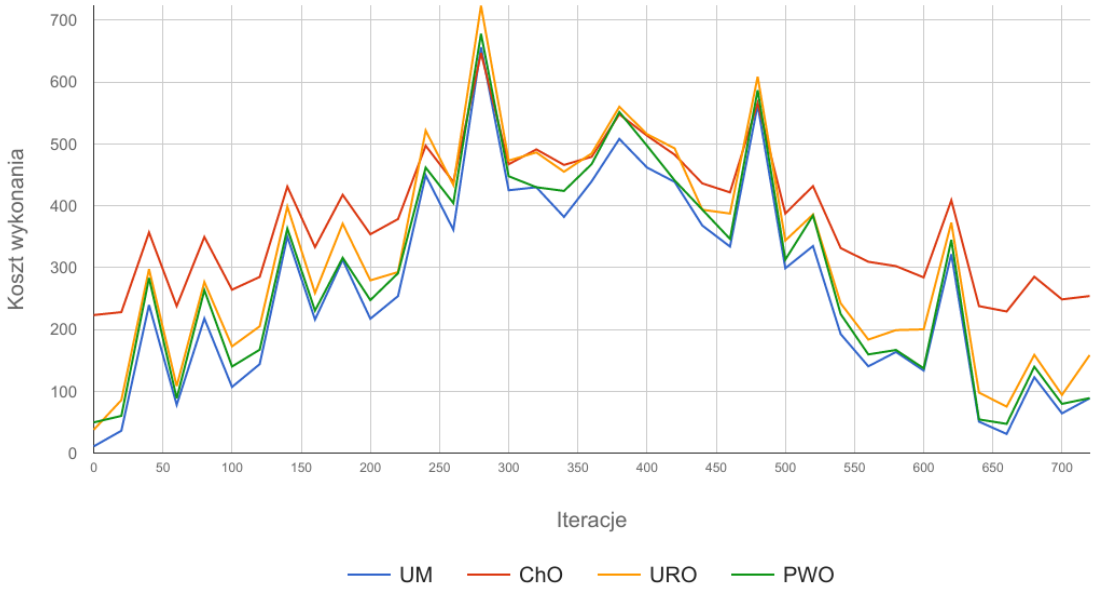
Sytuacja zmienia się dopiero dla $W = 0.5$. Wyniki testów dla tej konfiguracji środowiska testowego zostały przedstawione na wykresie 5.21. W środkowych iteracjach koszt wykonania w chmurze jest już znacznie niższy niż na urządzeniu. Jednorazowy rozdział operatorów wykonany przez algorytm URO nie sprawdza się w takim przypadku. Koszty wykonania uzyskane za pomocą algorytmu PWO, który oblicza rozdział w każdej iteracji, są bliskie optymalnym.

Rysunek 5.22 przedstawia wyniki testów dla konfiguracji środowiska z $W = 0.33$. Jeszcze bardziej widoczna jest tutaj duża różnica w kosztach wykonania w środkowej fazie działania aplikacji. Koszty dla początkowych i końcowych iteracji są jednak podobne w obu środowiskach, stąd niewielkie różnice pomiędzy algorytmami URO i PWO.

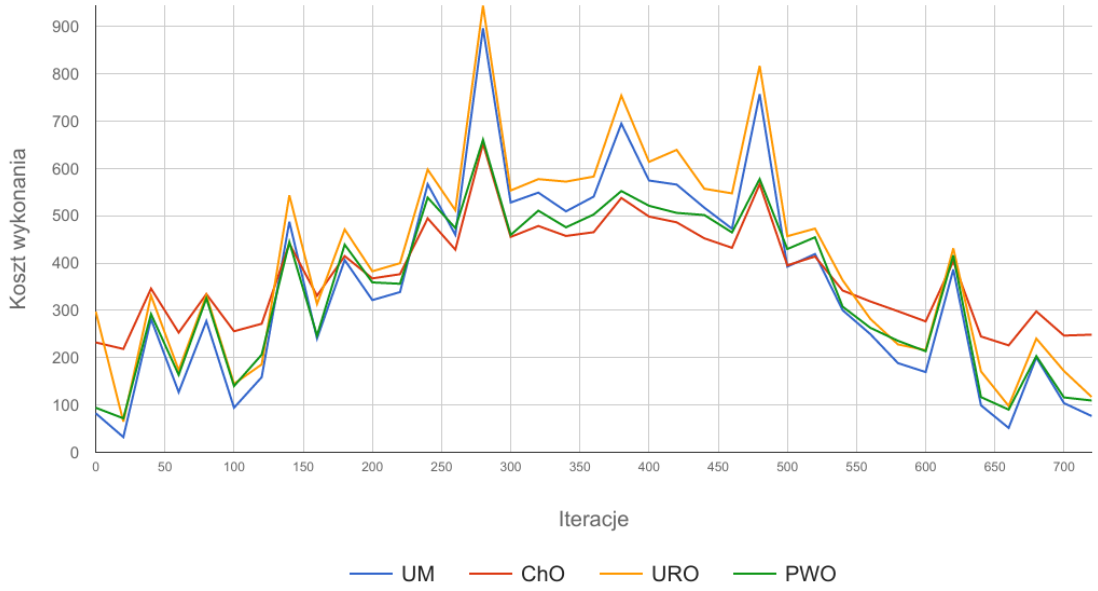
Podsumowanie całkowitych kosztów wykonania (w sekundach) zostało przedstawione na rysunku 5.23 oraz w tabeli 5.2. Aplikacja monitorująca parking ma dużo



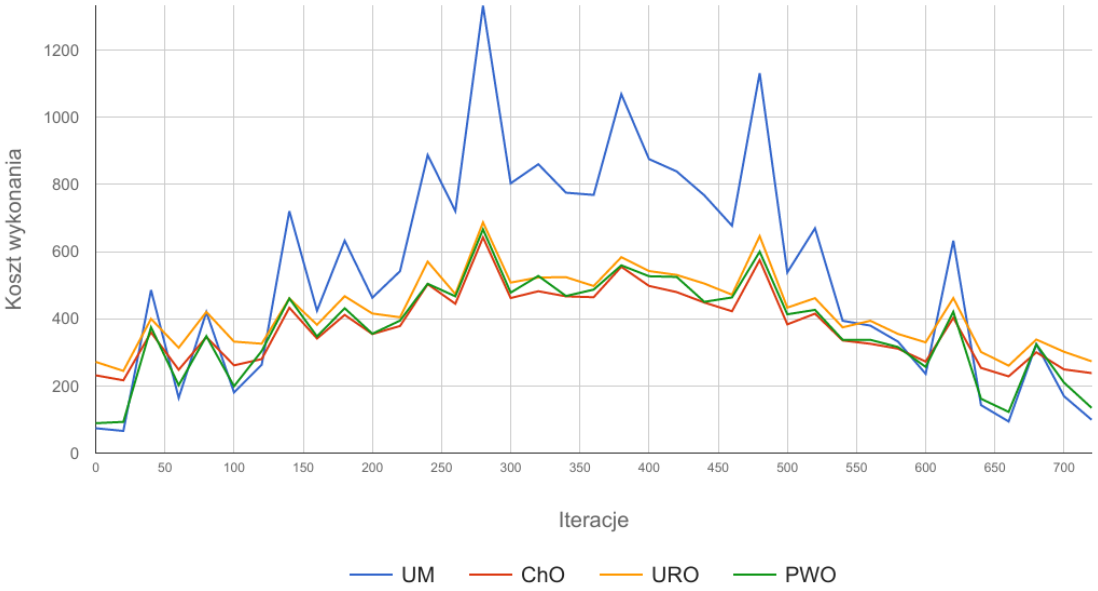

Rysunek 5.19: Wyniki testów dla $W = 1$ i sieci 3G



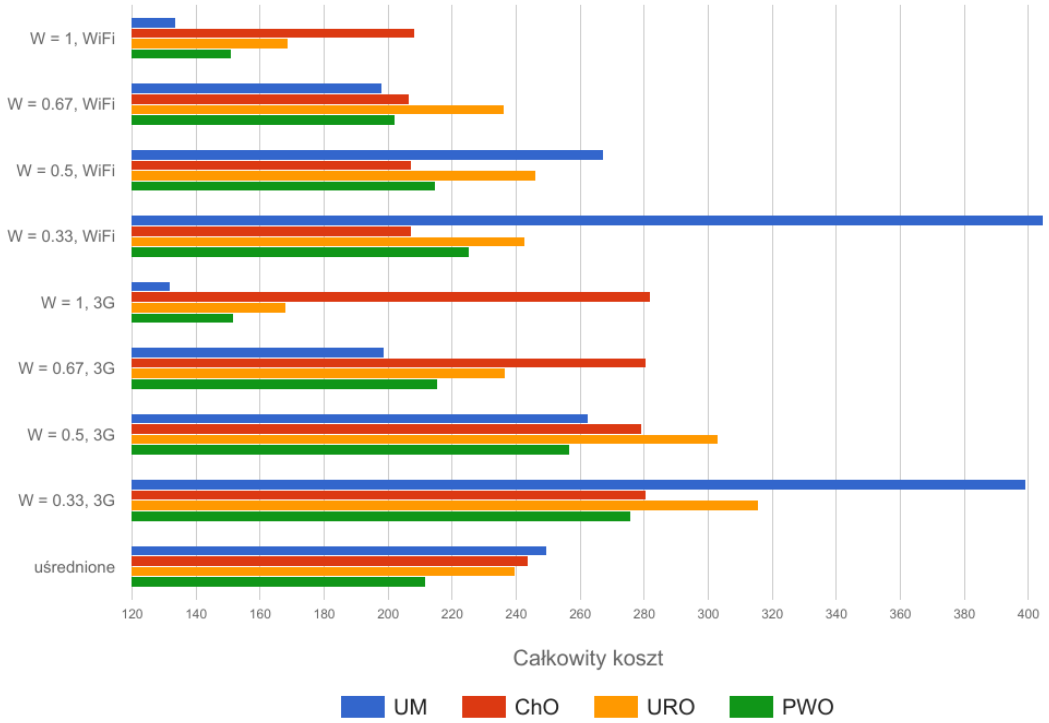

Rysunek 5.20: Wyniki testów dla $W = 0.67$ i sieci 3G




Rysunek 5.21: Wyniki testów dla $W = 0.5$ i sieci 3G




Rysunek 5.22: Wyniki testów dla $W = 0.33$ i sieci 3G




Rysunek 5.23: Całkowite koszty dla wszystkich konfiguracji środowiska testowego

większą zmienność kosztów w kolejnych iteracjach niż aplikacja do gry w szachy, dlatego można zaobserwować jeszcze większe różnice pomiędzy algorytmem URO i PWO, na korzyść algorytmu PWO. Dynamiczna zmiana rozdziału operatorów w każdej iteracji może w tym przypadku obniżyć całkowity koszt wykonania aż o 13%, uśredniając po wszystkich testowanych konfiguracjach.

Tabela 5.2: Całkowite koszty dla wszystkich konfiguracji środowiska testowego

| | | całkowity koszt | | | |
|---|---|---|---|---|---|
| **W** | **sieć** | **UM** | **ChO** | **URO** | **PWO** |
| 1 | WiFi | 130,9 | 207,58 | 170,67 | 149,16 |
| 0,67 | WiFi | 196,64 | 206,79 | 234,03 | 203,45 |
| 0,5 | WiFi | 265,31 | 206,7 | 243,71 | 215,3 |
| 0,33 | WiFi | 401,03 | 206,94 | 245,42 | 220,94 |
| 1 | 3G | 131,89 | 280,67 | 169,09 | 149,47 |
| 0,67 | 3G | 199,22 | 280,34 | 235,19 | 214,34 |
| 0,5 | 3G | 267,4 | 281,13 | 308,25 | 256,45 |
| 0,33 | 3G | 398,83 | 281,1 | 318,1 | 275,98 |
| | średnia | 248,9 | 243,91 | 240,56 | 210,64 |

# Rozdział 6

# Wnioski końcowe

Do najistotniejszych osiągnięć rozprawy należy zaliczyć:

1. Zaproponowanie modelu interaktywnych aplikacji mobilnych oraz ich podstawowych parametrów jakościowych, związanych z kosztem wykonania w dzielonym środowisku chmury obliczeniowej i urządzenia mobilnego.

2. Opracowanie dwóch reprezentatywnych algorytmów alokacji komponentów aplikacji interaktywnych dla zapewnienia efektywnej współpracy chmury obliczeniowej z inteligentnym urządzeniem końcowym oraz oszacowanie ich złożoności obliczeniowej.

3. Budowa programu zarządzającego współpracą chmury obliczeniowej oraz urządzenia użytkownika w celu określenia zakresu użyteczności zaproponowanych algorytmów.

4. Przygotowanie środowiska testowego w oparciu o otwarte oprogramowanie do przeprowadzanie analizy zachowania się tych algorytmów w konkretnym środowisku.

Pierwsza tezy rozprawy została wykazana za pomocą dowodu przedstawionego w rozdziale 3.5.2. Druga teza udowodniona została przez zaproponowane algorytmy rozdziału, odpowiednio algorytm GRK o złożoności wielomianowej w rozdziale 3.5.4 i algorytm PWO o liniowej złożoności w rozdziale 4.6. Wyniki eksperymentalne potwierdziły, że algorytm PWO ma zastosowanie w optymalizacji rozdziału aplikacji o zmiennej charakterystyce wykonania.

Oprócz realizacji badań teoretycznych związanych z opracowaniem modelu aplikacji i opracowaniem algorytmów ich rozdziału oraz oszacowaniem złożoności obli-

czeniowej (teoria grafów) opracowano środowisko praktyczne, framework zarządzający oraz zestaw testów umożliwiających testowanie różnego typu aplikacji interaktywnych różniących się zarówno architekturą, jak też charakterystykami wydajnościowymi. Określono warunki użyteczności opracowanych algorytmów oraz oprogramowania zarządzającego dla trzech różnych konfiguracji: wszystkie obliczenia na urządzeniu użytkownika, wszystkie obliczenia na chmurze (cienki klient) lub podział obliczeń na chmurę oraz urządzenie (framework). Tego typu rozwiązania mogą być więc szeroko wykorzystane w przypadku dowolnych urządzeń współpracujących z chmurą obliczeniową. Odpowiada to współczesnym trendom rozwojowym internetu rzeczy (IoT).

W rozprawie analizowano przypadek jednej aplikacji współpracującej z jednym urządzeniem i jedną chmurą obliczeniową oraz jednym użytkownikiem. Przedstawione wyniki stwarzają warunki wyjściowe do współpracy wielu urządzeń ze sobą przy realizacji różnego typu aplikacji interaktywnych. Wówczas przedstawione rozwiązania dotyczące modelu klient-serwer można przenieść do modelu peer-to-peer. Problemem otwartym pozostaje budowa bardziej złożonego programu zarządzającego. Zadanie to jeszcze bardziej się komplikuje w sytuacji jednoczesnego wykorzystania zasobów chmury obliczeniowej do obliczeń realizowanych jednocześnie w czasie rzeczywistym. Wzrastają wówczas wymagania czasowe dotyczące działającego frameworka.

Wydaje się, że w wyżej wymienionych przypadkach, kiedy liczba różnego typu zdarzeń jest bardzo duża, należy wyróżnić reprezentatywne scenariusze działań, które dzięki poznanym wynikom przedstawionym w rozprawie, mogą być znacząco uproszczone, tak że tak złożona siła urządzeń i chmury obliczeniowej, może efektywnie funkcjonować w zadanych przestrzeniach IoT.

Zaprezentowane algorytmy dla statycznego i dynamicznego modelu współpracy mogą być również dalej rozwijane zarówno pod względem zmniejszania złożoności obliczeniowej, jak i uwzględnienia wielu istotnych parametrów jakościowych. W tym celu mogą być wykorzystane różne heurystyki optymalizacyjne związane z algorytmami genetycznymi czy mrówkowymi. Jednak z praktycznego punktu widzenia powinny być to algorytmy proste, same w sobie nie angażujące mocno zasobów obliczeniowych urządzeń i chmury obliczeniowej.

# Bibliografia


[1] Saeid Abolfazli, Zohreh Sanaei, Erfan Ahmed, Abdullah Gani i Rajkumar Buyya. 'Cloud-based augmentation for mobile devices: motivation, taxonomies, and open challenges'. W: *Communications Surveys & Tutorials, IEEE* 16.1 (2014), s. 337–368.

[2] Telerik AD. *Cross-Platform Native Development with JavaScript*. 2016. URL: `https://www.nativescript.org/` (term. wiz. 27. 09. 2016).

[3] Faisal Alam, Preeti Ranjan Panda, Nikhil Tripathi, Neelam Sharma i Sanjiv Narayan. 'Energy optimization in Android applications through wakelock placement'. W: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE. 2014, s. 1–4.

[4] Amazon. *AWS IoT - Amazon Web Services*. 2016. URL: `https://aws.amazon.com/iot/` (term. wiz. 27. 07. 2016).

[5] Arash Asadi i Vincenzo Mancuso. 'WiFi Direct and LTE D2D in action'. W: *Wireless Days (WD), 2013 IFIP*. IEEE. 2013, s. 1–8.

[6] Luigi Atzori, Tiziana Dessi i Vlad Popescu. 'Indoor navigation system using image and sensor data processing on a smartphone'. W: *Optimization of Electrical and Electronic Equipment (OPTIM), 2012 13th International Conference on*. IEEE. 2012, s. 1158–1163.

[7] Wael Badawy i Graham A Julien. *System-on-chip for Real-time Applications*. T. 711. Springer Science & Business Media, 2012.

[8] Harald Bauer, Yetloong Goh, Sebastian Schlink i Christopher Thomas. 'The supercomputer in your pocket'. W: *MkKinsey on Semiconductors (Autumn* (2012), s. 14–27.

[9] Joshua Bloch. *Effective Java*. Prentice Hall PTR, 2008.

[10] Flavio Bonomi, Rodolfo Milito, Jiang Zhu i Sateesh Addepalli. 'Fog computing and its role in the internet of things'. W: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, s. 13–16.

[11] AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu i Colin Dixon. 'Home automation in the wild: challenges and opportunities'. W: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2011, s. 2115–2124.

[12] Aaron Carroll, Gernot Heiser i in. 'An Analysis of Power Consumption in a Smartphone.' W: *USENIX annual technical conference*. T. 14. Boston, MA. 2010, s. 21–21.

[13] Grazia Cecere, Nicoletta Corrocher i Riccardo David Battaglia. 'Innovation and competition in the smartphone industry: Is there a dominant design?' W: *Telecommunications Policy* 39.3 (2015), s. 162–175.

[14] Kai-Di Chang, Jiann-Liang Chen, Wei-Ming Chen, Han-Chieh Chao i Chin-Feng Lai. 'Cloud multimedia system with dynamic adjustable streaming service'. W: *World Automation Congress (WAC), 2012*. IEEE. 2012, s. 1–6.

[15] Whai-En Chen, Chun-Chieh Chiu i Yuan-Bo Chang. 'A performance study on context-aware real-time multimedia transmission between smartphone and cloud'. W: *Intelligent Signal Processing and Communications Systems (ISPACS), 2012 International Symposium on*. IEEE. 2012, s. 211–215.

[16] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik i Ashwin Patti. 'Clonecloud: elastic execution between mobile device and cloud'. W: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, s. 301–314.

[17] Cisco. *Cisco Jasper | The ON Switch for The Internet of Things*. 2016. URL: `http://www.jasper.com/` (term. wiz. 27. 07. 2016).

[18] Cisco Visual Networking Index Cisco. 'Global mobile data traffic forecast update, 2013–2018'. W: *white paper* (2014).

[19] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra i Paramvir Bahl. 'MAUI: making smartphones last longer with code offload'. W: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM. 2010, s. 49–62.

[20] Evan Czaplicki. ‘Elm: Concurrent FRP for Functional GUIs’. W: *Senior thesis, Harvard University* (2012).

[21] Ryan Dahl. ‘Node. js: evented I/O for v8 javascript’. W: *URL: https://www. nodejs. org* (2012).

[22] Ivanoe De Falco, Eryk Laskowski, Richard Olejnik, Umberto Scafuri, Ernesto Tarantino i Marek Tudruj. ‘Load balancing in distributed applications based on extremal optimization’. W: *European Conference on the Applications of Evolutionary Computation*. Springer. 2013, s. 52–61.

[23] Hoang T Dinh, Chonho Lee, Dusit Niyato i Ping Wang. ‘A survey of mobile cloud computing: architecture, applications, and approaches’. W: *Wireless communications and mobile computing* 13.18 (2013), s. 1587–1611.

[24] Zeno Rocha Pablo Carvalho Eduardo Lundgren Thiago Rocha i Maira Bello. *tracking.js – A modern approach for Computer Vision on the web*. 2017. URL: `https://trackingjs.com/` (term. wiz. 06. 06. 2017).

[25] Opher Etzion i Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.

[26] Muhamad Felemban, Saleh Basalamah i Abdul Ghafoor. ‘A distributed cloud architecture for mobile multimedia services’. W: *Network, IEEE* 27.5 (2013), s. 20–27.

[27] Niroshinie Fernando, Seng W Loke i Wenny Rahayu. ‘Mobile cloud computing: A survey’. W: *Future Generation Computer Systems* 29.1 (2013), s. 84–106.

[28] Ian Fette. ‘The websocket protocol’. W: (2011).

[29] The Apache Software Foundation. *Apache Cordova*. 2016. URL: `https://cordova.apache.org/` (term. wiz. 27. 09. 2016).

[30] Alexander Gaeta, Sokol Kosta, Julinda Stefa i Alessandro Mei. ‘StreamSmart: P2P video streaming for smartphones through the cloud’. W: *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*. IEEE. 2013, s. 233–235.

[31] Inc. Gartner. *Gartner Smart Phone Marketshare 2015 Q4*. 2016. URL: `http://www.gartner.com/newsroom/id/3215217` (term. wiz. 04. 04. 2016).

[32] GitHub. *Electron - Build cross platform desktop apps with JavaScript, HTML, and CSS*. 2016. URL: `http://electron.atom.io/` (term. wiz. 27. 09. 2016).

[33] Carles Gomez, Joaquim Oller i Josep Paradells. 'Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology'. W: *Sensors* 12.9 (2012), s. 11734–11753.

[34] Google. *Chrome V8*. 2016. URL: `https://developers.google.com/v8/` (term. wiz. 27. 09. 2016).

[35] JX Hao i James B Orlin. 'A faster algorithm for finding the minimum cut in a directed graph'. W: *Journal of Algorithms* 17.3 (1994), s. 424–446.

[36] Qusay Hassan. 'Demystifying cloud computing'. W: *The Journal of Defense Software Engineering* (2011), s. 16–21.

[37] Christian Hohn i Colin R Reeves. 'Graph partitioning using genetic algorithms'. W: *in Proceedings of the 2nd International Conference on Massively Parallel Computing Systems*. Citeseer. 1996.

[38] International Data Corporation IDC. 'Worldwide Internet of Things Forecast, 2014–2020'. W: *white paper* (2014).

[39] Adobe Systems Inc. *PhoneGap*. 2016. URL: `http://phonegap.com/` (term. wiz. 27. 09. 2016).

[40] Google Inc. *ART and Dalvik | Android Open Source Project*. 2017. URL: `https://source.android.com/devices/tech/dalvik/` (term. wiz. 07. 03. 2017).

[41] Xin Jin i Yu-Kwong Kwok. 'Cloud assisted P2P media streaming for bandwidth constrained mobile subscribers'. W: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE. 2010, s. 800–805.

[42] Jinki Jung, Jaewon Ha, Sang-Wook Lee, Francisco A Rojas i Hyun S Yang. 'Efficient mobile AR technology using scalable recognition and tracking based on server-client model'. W: *Computers & Graphics* 36.3 (2012), s. 131–139.

[43] Holden Karau, Andy Konwinski, Patrick Wendell i Matei Zaharia. *Learning spark: lightning-fast big data analysis*. Ó'Reilly Media, Inc.", 2015.

[44] Roelof Kemp, Nicholas Palmer, Thilo Kielmann i Henri Bal. 'Cuckoo: a computation offloading framework for smartphones'. W: *Mobile Computing, Applications, and Services*. Springer, 2010, s. 59–79.

[45] Ab Rouf Khan, Marini Othman, Sajjad Ahmad Madani i Samee U Khan. 'A survey of mobile cloud computing application models'. W: *Communications Surveys & Tutorials, IEEE* 16.1 (2014), s. 393–413.

[46] Patrick Kinney i in. 'Zigbee technology: Wireless control that simply works'. W: *Communications design conference*. T. 2. 2003, s. 1–7.

[47] Dejan Koavchev, Yiwei Cao i Ralf Klamma. 'Mobile multimedia cloud computing and the web'. W: *Multimedia on the Web (MMWeb), 2011 Workshop on*. IEEE. 2011, s. 21–26.

[48] Stephen G Kochan. *Programming in objective-C*. Addison-Wesley Professional, 2011.

[49] Keiko Kohmoto, Kengo Katayama i Hiroyuki Narihisa. 'Performance of a genetic algorithm for the graph partitioning problem'. W: *Mathematical and computer modelling* 38.11-13 (2003), s. 1325–1332.

[50] Richard E Korf i David Maxwell Chickering. 'Best-first minimax search'. W: *Artificial intelligence* 84.1 (1996), s. 299–337.

[51] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier i Xinwen Zhang. 'Unleashing the power of mobile cloud computing using thinkair'. W: *arXiv preprint arXiv:1105.3232* (2011).

[52] Dejan Kovachev i Ralf Klamma. 'Context-aware mobile multimedia services in the cloud'. W: *Proceedings of the 10th International Workshop of the Multimedia Metadata Community on Semantic Multimedia Database Technologies*. Citeseer. 2009.

[53] Henryk Krawczyk i Michał Nykiel. 'Mobile devices and computing cloud resources allocation for interactive applications'. W: *Archives of Control Sciences*. T. 27. 2017.

[54] Henryk Krawczyk i Michał Nykiel. 'Optymalizacja współpracy urządzenia mobilnego i chmury obliczeniowej dla efektywnego wykonania złożonych aplikacji internetowych'. W: *Automatyzacja Procesów Dyskretnych*. Politechnika Śląska. 2016.

[55] Henryk Krawczyk, Michał Nykiel, Piotr Orzechowski i Jerzy Proficz. 'Platforma KASKADA - proces wytwarzania oprogramowania'. W: *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne*. 2013.

[56] Henryk Krawczyk, Michał Nykiel i Jerzy Proficz. 'Mobile Offloading Framework: Solution for Optimizing Mobile Applications Using Cloud Computing'. W: *Computer Networks*. Springer International Publishing, 2015, s. 293–305.

[57] Henryk Krawczyk, Michał Nykiel i Jerzy Proficz. 'Tryton supercomputer capabilities for analysis of massive data streams'. W: *Polish Maritime Research* 22.3 (2015), s. 99–104.

[58] Tom Krazit. *ARMed for the living room*. 2006. URL: `http://www.cnet.com/news/armed-for-the-living-room/` (term. wiz. 03. 04. 2016).

[59] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu i Bharat Bhargava. 'A survey of computation offloading for mobile systems'. W: *Mobile Networks and Applications* 18.1 (2013), s. 129–140.

[60] Li Li, Xiong Li, Sun Youxia i Liu Wen. 'Research on mobile multimedia broadcasting service integration based on cloud computing'. W: *Multimedia Technology (ICMT), 2010 International Conference on*. IEEE. 2010, s. 1–4.

[61] Jesse Liberty, Paul Betts i Stefan Turalski. *Programming Reactive Extensions and LINQ*. Springer, 2011.

[62] Fangming Liu, Peng Shu, Hai Jin, Linjie Ding, Jie Yu, Di Niu i Bo Li. 'Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications'. W: *Wireless Communications, IEEE* 20.3 (2013), s. 14–22.

[63] Chessgames Services LLC. *Chess Statistics*. 2016. URL: `http://www.chessgames.com/chessstats.html` (term. wiz. 04. 05. 2016).

[64] Ricky KK Ma, King Tin Lam i Cho-Li Wang. 'excloud: Transparent runtime support for scaling mobile applications in cloud'. W: *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE. 2011, s. 103–110.

[65] Muhammad Yasir Malik. 'Power consumption analysis of a modern smartphone'. W: *arXiv preprint arXiv:1212.1896* (2012).

[66] Verdi March, Yan Gu, Erwin Leonardi, George Goh, Markus Kirchberg i Bu Sung Lee. '$\mu$cloud: towards a new paradigm of rich mobile applications'. W: *Procedia Computer Science* 5 (2011), s. 618–624.

[67] John McCarthy. 'Reminiscences on the history of time sharing'. W: *Online: http://www-formal. stanford. edu/jmc/history/timesharing/timesharing. htm. Last accessed: April* 7 (1983), s. 2008.

[68] Peter Mell i Tim Grance. ‘The NIST definition of cloud computing’. W: (2011).

[69] Dirk Merkel. ‘Docker: lightweight linux containers for consistent development and deployment’. W: *Linux Journal* 2014.239 (2014), s. 2.

[70] Microsoft. *Azure IoT Suite - Connect Devices and Data*. 2016. URL: `https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite` (term. wiz. 27. 07. 2016).

[71] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[72] Michel Mouly, Marie-Bernadette Pautet i Thomas Foreword By-Haug. *The GSM system for mobile communications*. Telecom publishing, 1992.

[73] Jarek Nabrzyski, Jennifer M Schopf i Jan Weglarz. *Grid resource management: state of the art and future trends*. T. 64. Springer Science & Business Media, 2012.

[74] Henrik Nilsson, Antony Courtney i John Peterson. ‘Functional reactive programming, continued’. W: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. 2002, s. 51–64.

[75] NVIDIA. *Tegra X1 Whitepaper*. 2015. URL: `http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf` (term. wiz. 08. 11. 2016).

[76] T Occhino. ‘React Native: Bringing modern web techniques to mobile’. W: *Facebook code* 26 (2015).

[77] Stefan Poslad. *Ubiquitous computing: smart devices, environments and interactions*. Wiley-Blackwell, 2009.

[78] D Pountain. ‘BYTE UK - A Plethora Of Portables’. W: *BYTE* 9.12 (1984), s. 413.

[79] G Baba Prasad, K Seema, UH Shrikant, Gopi Krishna Garge, SVR Anand i Malati Hegde. ‘SeaMo+: A virtual real-time multimedia service framework on handhelds to enable remote real-time patient monitoring for mobile doctors’. W: *Communication Systems and Networks (COMSNETS), 2013 Fifth International Conference on*. IEEE. 2013, s. 1–6.

[80] John W Rittinghouse i James F Ransome. *Cloud computing: implementation, management, and security*. CRC press, 2009.

[81] Ana Paula Rocha, Oscar Pereira, Diogo Ribeiro, Jose Maria Fernandes, Silva Cunha i Joao Paulo. 'MonitorMe: Online video monitoring for first responders using a smartphone'. W: *e-Health Networking, Applications & Services (Healthcom), 2013 IEEE 15th International Conference on*. IEEE. 2013, s. 731–733.

[82] Rick Rogers, John Lombardo, Zigurd Mednieks i Blake Meike. *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 2009.

[83] Salesforce. *Internet of Things (IoT) Platform & Software*. 2016. URL: `http://www.salesforce.com/iot-cloud/` (term. wiz. 27. 07. 2016).

[84] Zohreh Sanaei, Saeid Abolfazli, Abdullah Gani i Rajkumar Buyya. 'Heterogeneity in mobile cloud computing: taxonomy and open challenges'. W: *Communications Surveys & Tutorials, IEEE* 16.1 (2014), s. 369–392.

[85] Mahadev Satyanarayanan. 'Mobile computing: the next decade'. W: *Proceedings of the 1st ACM workshop on mobile cloud computing & services: social networks and beyond*. ACM. 2010, s. 5.

[86] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres i Nigel Davies. 'The case for vm-based cloudlets in mobile computing'. W: *Pervasive Computing, IEEE* 8.4 (2009), s. 14–23.

[87] Gheorghe Sebestyen, Anca Hangan, Katalin Sebestyen i Roland Vachter. 'Self-tuning multimedia streaming system on cloud infrastructure'. W: *Procedia Computer Science* 18 (2013), s. 1342–1351.

[88] Omar Sefraoui, Mohammed Aissaoui i Mohsine Eleuldj. 'OpenStack: toward an open-source solution for cloud computing'. W: *International Journal of Computer Applications* 55.3 (2012).

[89] Claude E Shannon. 'Programming a computer for playing chess'. W: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), s. 256–275.

[90] Shiann-Tsong Sheu i Chi-Hao Huang. 'Mixed P2P-CDN system for media streaming in mobile environment'. W: *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*. IEEE. 2011, s. 657–660.

[91] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt i Roy Campbell. 'Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming'. W: *Proceedings of the 19th ACM international conference on Multimedia*. ACM. 2011, s. 103–112.

[92] Tolga Soyata, Rajani Muraleedharan, Colin Funai, Minseok Kwon i Wendi Heinzelman. 'Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture'. W: *Computers and Communications (ISCC), 2012 IEEE Symposium on*. IEEE. 2012, s. 000059–000066.

[93] André Staltz. *Cycle.js - a functional and reactive JavaScript framework for cleaner code*. 2016. URL: `http://cycle.js.org/` (term. wiz. 18. 04. 2016).

[94] Mechthild Stoer i Frank Wagner. 'A simple min-cut algorithm'. W: *Journal of the ACM (JACM)* 44.4 (1997), s. 585–591.

[95] Sasu Tarkoma, Matti Siekkinen, Eemil Lagerspetz i Yu Xiao. *Smartphone energy consumption: modeling and optimization*. Cambridge University Press, 2014.

[96] CI TASK. *Centrum Informatyczne Trójmiejskiej Akademickiej Sieci Komputerowej*. 2016. URL: `http://www.task.gda.pl` (term. wiz. 21. 12. 2016).

[97] Microsoft Open Technologies. *The Reactive Extensions for JavaScript*. 2016. URL: `https://github.com/Reactive-Extensions/RxJS` (term. wiz. 18. 04. 2016).

[98] Marco Costalba Tord Romstad i Joona Kiiski. *Stockfish - Strong open source chess engine*. 2017. URL: `https://stockfishchess.org/` (term. wiz. 01. 06. 2017).

[99] Geoff Walker. 'A review of technologies for sensing contact location on the surface of a display'. W: *Journal of the Society for Information Display* 20.8 (2012), s. 413–440.

[100] Shaoxuan Wang i Sujit Dey. 'Adaptive mobile cloud computing to enable rich mobile multimedia applications'. W: *Multimedia, IEEE Transactions on* 15.4 (2013), s. 870–883.

[101] Xiaofei Wang, Min Chen, Ted Taekyoung Kwon, Laurence T Yang i Victor CM Leung. 'AMES-cloud: a framework of adaptive mobile video streaming and efficient social video sharing in the clouds'. W: *Multimedia, IEEE Transactions on* 15.4 (2013), s. 811–820.

[102] Roy Want. ‘Near field communication’. W: *IEEE Pervasive Computing* 10.3 (2011), s. 4–7.

[103] Mark Weiser. ‘The computer for the 21st century’. W: *Scientific american* 265.3 (1991), s. 94–104.

[104] Tom White. *Hadoop: The definitive guide*. Ó’Reilly Media, Inc.", 2012.

[105] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han i Alvin Chan. ‘A framework for partitioning and execution of data stream applications in mobile cloud computing’. W: *ACM SIGMETRICS Performance Evaluation Review* 40.4 (2013), s. 23–32.

[106] Hongchi Zhang, Anas Al-Nuaimi, Xiaoyu Gu, Michael Fahrmair i Ryota Ishibashi. ‘Seamless and efficient stream switching of multi-perspective videos’. W: *Packet Video Workshop (PV), 2012 19th International*. IEEE. 2012, s. 31–36.

[107] Xinwen Zhang, Anugeetha Kunjithapatham, Sangoh Jeong i Simon Gibbs. ‘Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing’. W: *Mobile Networks and Applications* 16.3 (2011), s. 270–284.

[108] Honbo Zhou. *The internet of things in the cloud: A middleware perspective*. CRC press, 2012.

[109] Wenwu Zhu, Chong Luo, Jianfeng Wang i Shipeng Li. ‘Multimedia cloud computing’. W: *Signal Processing Magazine, IEEE* 28.3 (2011), s. 59–69.