

ANALISIS KINERJA RENDERING APLIKASI ANDROID REACT NATIVE DENGAN OPTIMASI MEMOIZATION

Faris Adhi Laksana Yusuf, Zuly Budiarmo

Teknik Informatika, Universitas Stikubank Semarang

Jl. Tri Lomba Juang, Mugassari, Kec. Semarang Sel., Kota Semarang, Jawa Tengah, Indonesia

farisadhi6004@mhs.unisbank.ac.id

ABSTRAK

Framework cross-platform seperti React Native banyak digunakan dalam pengembangan aplikasi *mobile* karena mendukung efisiensi pengembangan pada berbagai *platform*. Namun, mekanisme *rendering* pada React Native berpotensi menyebabkan *re-render* yang tidak diperlukan sehingga dapat menurunkan performa aplikasi. Penelitian ini bertujuan menganalisis pengaruh penerapan *memoization* menggunakan *React.memo* dan *useMemo* terhadap kinerja *rendering* aplikasi Android berbasis React Native. Metode penelitian yang digunakan adalah eksperimen semu dengan pendekatan kuantitatif melalui perbandingan dua kondisi pengujian, yaitu *baseline* dan *memoized*, pada tiga skenario pengujian. Pengukuran dilakukan menggunakan React Native DevTools dan Flipper LeakCanary berdasarkan metrik jumlah *re-render*, waktu *render*, dan indikasi *memory leak*. Hasil penelitian menunjukkan bahwa *memoization* mampu menurunkan rata-rata jumlah *re-render* sebesar 85,71% dan waktu *render* sebesar 93,62% pada skenario perubahan status keranjang. Pada skenario pencarian data, jumlah *re-render* menurun sebesar 50% dan waktu *render* sebesar 77,11%. Selain itu, tidak ditemukan indikasi *memory leak* pada skenario pengujian *mount-unmount* katalog. Dengan demikian, *memoization* terbukti efektif meningkatkan performa *rendering* aplikasi React Native Android.

Kata kunci : React Native, memoization, React.memo, useMemo, rendering, performa aplikasi

1. PENDAHULUAN

Perkembangan teknologi informasi yang pesat telah mendorong transformasi digital di berbagai sektor, salah satunya dalam pengembangan aplikasi *mobile*. Seiring dengan meningkatnya kebutuhan akan aplikasi yang dapat berjalan di berbagai *platform* dengan biaya pengembangan yang efisien, *framework cross-platform* seperti React Native menjadi semakin populer di kalangan pengembang [1] [2]. React Native merupakan *framework* JavaScript yang dikembangkan oleh Facebook yang memungkinkan pengembang untuk membangun aplikasi *mobile* dengan basis kode JavaScript. Menurut penelitian [3] [4], pendekatan *cross-platform* menawarkan efisiensi pengembangan yang lebih tinggi dibandingkan pengembangan *native*, namun tetap menghadirkan tantangan performa yang perlu dioptimalkan.

Salah satu permasalahan utama dalam pengembangan aplikasi React Native adalah manajemen proses *rendering* komponen. Mekanisme Virtual DOM yang digunakan React secara otomatis melakukan *re-render* pada setiap perubahan *state* atau *props*, yang dapat menyebabkan penurunan performa jika tidak dikelola dengan baik. *Re-render* yang tidak perlu menjadi sumber masalah performa yang umum terjadi, terutama ketika aplikasi memiliki hierarki komponen yang kompleks. Penelitian empiris pada [5] [6] mengkonfirmasi bahwa inefisiensi *rendering* dan *overhead* penggunaan sumber daya merupakan penyebab utama penurunan performa pada aplikasi *cross-platform*, sehingga mengoptimalkan proses *rendering* menjadi hal yang krusial untuk menjaga kelancaran aplikasi.

Berbagai penelitian telah dilakukan untuk mengevaluasi performa *framework cross-platform*. Beberapa studi membandingkan React Native dengan *framework* lain seperti Flutter, Xamarin, dan Ionic, menganalisis metrik-metrik seperti waktu eksekusi, konsumsi memori, dan konsumsi energi. Penelitian [7] menunjukkan bahwa performa *cross-platform* sangat bervariasi tergantung pada skenario penggunaan, dan pendekatan sistematis diperlukan untuk identifikasi *bottleneck*. Pentingnya optimasi performa secara fundamental juga telah ditegaskan oleh studi komprehensif di penelitian [8]. Dalam survei mereka yang menganalisis lebih dari 150 publikasi ilmiah, mereka menyatakan bahwa untuk memastikan kepuasan dan kesuksesan pengguna, aplikasi harus memiliki performa yang tinggi. Hal ini sangat krusial mengingat perangkat *mobile* adalah sistem dengan sumber daya yang terbatas (*resource-constrained systems*). Penelitian mereka juga menyoroti bahwa karakteristik non-fungsional seperti konsumsi energi dan memori memberikan peran penting bagi kepuasan pengguna.

Salah satu teknik optimasi performa yang banyak diterapkan untuk mengurangi *re-render* yang tidak diperlukan adalah *memoization*. Penelitian [9] menjelaskan bahwa *memoization* dapat menyimpan hasil komputasi yang mahal ke dalam *cache* sehingga tidak perlu dihitung ulang pada setiap *render*. Dalam konteks React Native, *memoization* dapat diimplementasikan menggunakan hook *useMemo* untuk menyimpan hasil komputasi dan *React.memo* untuk mencegah *re-render* komponen ketika *props* tidak berubah. Sementara pada penelitian [10] menjelaskan bahwa penerapan *memoization* yang

efektif dapat memberikan peningkatan performa yang signifikan, dengan frekuensi *rendering* yang berkurang serta penggunaan memori yang lebih stabil.

Berdasarkan uraian tersebut, dapat diketahui bahwa berbagai penelitian sebelumnya lebih banyak berfokus pada perbandingan performa React Native dengan *framework* lain atau pada evaluasi karakteristik performa secara umum. Penelitian yang secara khusus mengevaluasi efektivitas teknik *memoization* terhadap kinerja *rendering* aplikasi React Native Android masih relatif terbatas. Oleh karena itu, penelitian ini bertujuan menganalisis pengaruh penerapan *memoization* menggunakan React.memo dan useMemo terhadap kinerja *rendering* aplikasi Android berbasis React Native.

Untuk mencapai tujuan tersebut, penelitian dilakukan menggunakan metode eksperimen semu dengan membandingkan kondisi *baseline* dan *memoized* berdasarkan metrik jumlah re-render, waktu render, serta indikasi kebocoran memori menggunakan React Native DevTools dan Flipper dengan *plugin* LeakCanary. Hasil penelitian diharapkan dapat memberikan bukti empiris mengenai efektivitas *memoization* sebagai strategi optimasi *rendering* pada aplikasi React Native Android.

2. TINJAUAN PUSTAKA

2.1. React Native

React Native adalah *framework* JavaScript sumber terbuka yang dikembangkan oleh Meta untuk membangun aplikasi *mobile* lintas platform dengan basis kode yang sama untuk Android dan iOS [1]. Berbeda dengan pendekatan *hybrid* tradisional yang menggunakan *web view*, React Native mengompilasi komponen antarmuka ke dalam komponen *native platform*, sehingga performanya lebih mendekati aplikasi *native* murni [1].

2.2. Memoization pada React Native

Memoization adalah teknik optimasi yang menyimpan hasil komputasi yang mahal ke dalam *cache* sehingga tidak perlu adanya perhitungan ulang pada setiap *render* [9]. Dalam konteks React Native, *memoization* dapat diimplementasikan menggunakan React.memo untuk komponen fungsional dan useMemo untuk komputasi nilai. React.memo adalah *higher-order component* yang digunakan untuk mengoptimalkan komponen fungsional dengan cara menyimpan hasil render terakhir. Ketika komponen induk mengalami *re-render*, React.memo akan membandingkan *props* yang lama dengan *props* yang baru menggunakan perbandingan *shallow* (Object.is). Apabila tidak ada perubahan pada *props*, React akan melewati proses *re-render* komponen tersebut dan menggunakan hasil *render* yang tersimpan [9]. Sedangkan useMemo adalah hook bawaan React yang digunakan untuk *memoize* hasil dari suatu komputasi yang mahal. useMemo menerima dua argumen, yaitu sebuah fungsi yang mengembalikan nilai yang ingin di-*cache*, dan sebuah *array* dependensi. Nilai yang di-

cache hanya akan dihitung ulang ketika salah satu dependensi berubah [9].

2.3. Alat Ukur Performa

Pengukuran performa *rendering* yang akurat dan komprehensif sangatlah diperlukan untuk hasil penilaian yang maksimal. Sejalan dengan pendekatan evaluasi berbasis metrik yang ditekankan pada penelitian [10] untuk menilai fitur *debugging* dan performa aplikasi, penggunaan kombinasi alat memungkinkan pengukuran performa *rendering* yang lebih valid dibandingkan hanya mengandalkan satu alat ukur. Maka dari itu penelitian ini memanfaatkan dua alat *debugging* dan *profiling* resmi dari ekosistem React Native, yaitu React Native DevTools dan Flipper. React Native DevTools adalah seperangkat alat *debugging* dan *profiling* resmi yang disediakan oleh ekosistem React Native [11]. Fitur utama untuk analisis performa *rendering* adalah Panel Profiler yang mencatat setiap aktivitas *rendering* komponen, termasuk komponen mana yang mengalami *re-render* dan durasi setiap *render* dalam milidetik. Sedangkan Flipper adalah platform *debugging* untuk aplikasi *mobile* yang mendukung React Native [12]. Dalam konteks optimasi performa, Flipper menyediakan *plugin* Memory dan LeakCanary yang berfungsi untuk memantau konsumsi memori aplikasi secara *real-time* dan mendeteksi kebocoran memori [10] [13].

2.4. Penelitian Terdahulu

Penelitian mengenai kinerja React Native sering menyoroti *overhead* sumber daya dibandingkan aplikasi *native*. Pada penelitian [14] misalnya, penelitian tersebut membandingkan aplikasi *hybrid* React Native dengan aplikasi Android *native* pada fitur-fitur umum. Hasilnya menunjukkan bahwa aplikasi React Native mengonsumsi sekitar 6–8% lebih banyak energi dan sedikit lebih banyak CPU serta memori daripada aplikasi *native* yang setara. Hal ini mengindikasikan ada beban tambahan pada React Native, konsisten dengan temuan [15] yang menguji satu aplikasi *native* dan dua aplikasi *mobile cross-platform development* (termasuk React Native) dengan pengukuran penggunaan CPU, memori, dan *frame* saat interaksi UI. Studi yang dilakukan ini menemukan *framework lintas-platform* (termasuk React Native) membebani perangkat lebih tinggi dalam hal CPU, memori utama, dan memori GPU dibandingkan aplikasi *native*. Kedua studi tersebut menggarisbawahi bahwa aplikasi React Native Android cenderung memiliki efisiensi rendah pada skenario UI kompleks, sehingga memerlukan optimasi tambahan.

Sejumlah studi secara khusus membandingkan React Native dengan *framework* lain, terutama Flutter. Studi [16] melakukan pengujian praktis aplikasi *social media* berbasis React Native dan Flutter pada *emulator* Android. Mereka mengukur beberapa metrik performa (CPU, memori, *frame rate*) dalam skenario *infinite scroll* dan akses kamera. Hasilnya menunjukkan Flutter unggul dalam kecepatan *rendering* UI dan

efisiensi CPU, sementara aplikasi React Native mengalami penurunan *frame rate* yang signifikan saat *scrolling* berkelanjutan. Selain perbandingan dengan *framework* Flutter, ada juga penelitian [17] yang melakukan pengembangan dan evaluasi aplikasi *chatbot* AI yang diimplementasikan dengan Kotlin Multiplatform dan React Native dengan fitur dan desain UI yang identik. Analisis performa yang dilakukan pada perangkat Android dan iOS menunjukkan bahwa Kotlin Multiplatform mencapai performa yang lebih baik dalam efisiensi CPU dan manajemen memori, sementara React Native menunjukkan kecepatan pengembangan yang lebih cepat dan responsivitas UI yang sedikit lebih baik.

Selain perbandingan lintas-platform, beberapa studi membahas optimasi dan kerangka kerja React Native. Studi [10] mengulas kinerja dan fitur *debugging* pada aplikasi *hybrid* React Native. Mereka mencatat bahwa pengembang cenderung lebih fokus pada antarmuka UI daripada performa, sehingga penting adanya evaluasi metrik performa React Native secara sistematis. Artikel ini menekankan kebutuhan pengukuran performa aplikasi React Native yang didasarkan data objektif. Sementara pada studi [18] menyajikan analisis komparatif komprehensif tentang performa berbagai mesin JavaScript dalam konteks aplikasi *hybrid* React Native, dengan fokus khusus pada perbedaan antara arsitektur JavaScript *Interface* dan *bridge* tradisional. Pada studi ini dilakukan serangkaian uji performa pada mesin JavaScript paling populer, termasuk Hermes, JavaScriptCore, dan V8, menggunakan tolok ukur dan kasus penggunaan tertentu. Hasil eksperimen menunjukkan perbedaan performa antar mesin tergantung pada skenario serta perbedaan antara penggunaannya dalam arsitektur *bridge* dan *JSI*. Seperti yang juga ditegaskan pada studi [1] yang menyatakan bahwa *overhead* JavaScript meningkatkan latensi rendering, menegaskan bahwa performa React Native secara konsisten lebih rendah dalam skenario grafis intensif.

Secara keseluruhan, kajian-kajian terdahulu mengidentifikasi bahwa aplikasi React Native Android cenderung memiliki *overhead* CPU/memori lebih tinggi dan *frame rendering* lebih lambat dibanding aplikasi *native* atau *framework* lain (Flutter misalnya). Belum banyak studi yang secara khusus menguji teknik optimasi *rendering* (seperti *memoisasi*) pada React Native Android. Oleh karena itu, terdapat celah penelitian dalam mengevaluasi dampak *memoization* pada performa *rendering* React Native Android, yang menjadi fokus penelitian ini.

3. METODE PENELITIAN

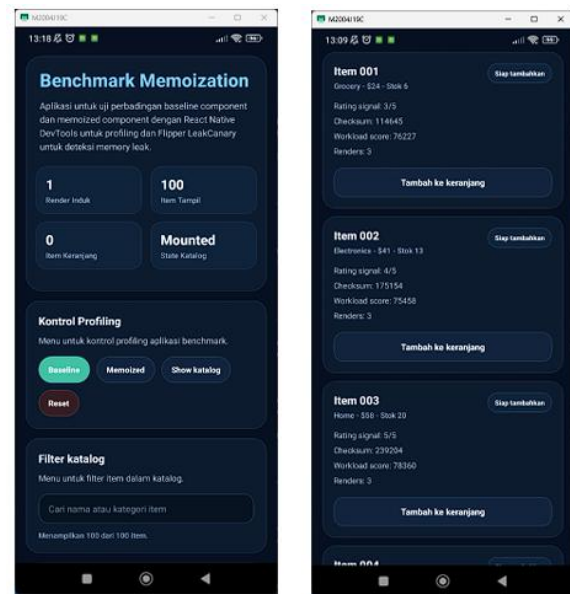
3.1. Desain Penelitian

Penelitian ini menggunakan pendekatan kuantitatif dengan metode eksperimen semu (*quasi-experiment*) menggunakan rancangan *repeated measures*. Metode ini digunakan karena pengukuran dilakukan pada objek yang sama dalam dua kondisi yang berbeda, yaitu kondisi *baseline* dan kondisi

memoized. Dengan rancangan penelitian tersebut, perbedaan performa aplikasi dapat diamati secara lebih objektif karena faktor perangkat, aplikasi, dan lingkungan pengujian yang tidak berbeda. Pendekatan ini selaras dengan penelitian-penelitian sebelumnya yang menilai *overhead* performa aplikasi *cross-platform* melalui eksperimen terkontrol dan pengukuran metrik objektif [5] [6].

3.2. Objek Penelitian

Objek dalam penelitian ini adalah aplikasi *benchmark* berbasis React Native 0.76 yang dirancang untuk menguji pengaruh *memoization* terhadap kinerja *rendering* aplikasi Android.



Gambar 1. Tampilan aplikasi *benchmark*

Dari tampilan aplikasi pada Gambar 1 terlihat aplikasi tersebut memuat katalog berisi 100 item, fitur pencarian, fitur keranjang, *render counter* pada setiap item katalog, serta kontrol untuk menampilkan dan menyembunyikan katalog. Aplikasi ini digunakan untuk mensimulasikan pola *rendering* pada aplikasi *mobile* yang kompleks. Dengan demikian, hasil pengujian dapat menggambarkan efek *memoization* secara lebih nyata pada komponen yang sering mengalami perubahan *state*.

3.3. Variabel dan Metrik Penelitian

Variabel dalam penelitian ini terdiri atas variabel bebas, variabel terikat, dan variabel kontrol.

Tabel 1. Variabel penelitian

Jenis Variabel	Variabel	Keterangan
Bebas	Penerapan <i>memoization</i>	Mode <i>baseline</i> dan mode <i>memoized</i>
Terikat	Jumlah <i>re-render</i>	Frekuensi <i>render</i> ulang komponen
Terikat	Waktu <i>render</i>	Durasi proses <i>render</i> dalam milidetik

Jenis Variabel	Variabel	Keterangan
Terikat	Kebocoran memori	<i>Retained object</i> atau tidak
Kontrol	Perangkat Android	Perangkat uji dibuat sama
Kontrol	Struktur aplikasi	Jumlah item dan komponen tetap
Kontrol	Skenario pengujian	Interaksi diuji seragam
Kontrol	Versi React Native	React Native 0.76

Dari Tabel 1 dapat dilihat semua variabel yang digunakan dalam penelitian ini secara mendetail mulai dari jenis variabel, nama variabel, serta keterangan dari variabel tersebut.

3.4. Perangkat dan Alat Penelitian

Penelitian ini menggunakan perangkat dan alat dengan performa yang mumpuni untuk mendukung proses penelitian secara keseluruhan.

Tabel 2. Spesifikasi perangkat dan alat

Kategori	Komponen	Spesifikasi / Versi
Perangkat keras	Laptop	Ryzen 5 5600H, RAM 16GB, SSD 512GB, Windows 11
Perangkat keras	Smartphone	Helio G80, RAM 4GB, Android 12
Perangkat lunak	Editor	Visual Studio Code 1.119.0
Perangkat lunak	Runtime	Node.js 24.12.0
Perangkat lunak	Framework	React Native 0.76.0
Perangkat lunak	Profiling	React Native DevTools 0.76.0
Perangkat lunak	Debugging	Flipper 0.273.0 + LeakCanary
Perangkat lunak	Browser	Microsoft Edge 147.0.3912.98

Dari Tabel 2 dapat dilihat semua informasi perangkat dan alat yang digunakan dalam segala proses yang dilakukan untuk melakukan penelitian ini.

3.5. Metrik dan Alat Ukur Penelitian

Pengukuran performa pada penelitian ini dilakukan dengan dua alat utama, yaitu React Native DevTools dan Flipper dengan plugin LeakCanary.

Tabel 3. Metrik dan alat ukur

Metrik	Alat Ukur	Satuan	Cara Pengukuran
Jumlah re-render	<i>Render counter internal</i>	Kali	Dibaca dari tampilan Renders: X
Waktu render	React Native DevTools Profiler	Milisecond	Dicatat dari durasi commit
Kebocoran memori	Flipper dengan plugin LeakCanary	Ada/Tidak	Diamati dari notifikasi <i>retained object</i>

Dari Tabel 3 dapat dilihat metrik beserta alat ukurnya yang digunakan dalam penelitian ini mulai dari *Render counter internal* yang terletak pada tampilan aplikasi untuk mengukur jumlah *re-render*, lalu React Native DevTools yang digunakan untuk waktu *render* melalui panel *Profiler*, sampai Flipper dengan plugin LeakCanary yang digunakan untuk mendeteksi kebocoran memori pada saat katalog *di-mount* dan *di-unmount*. Pemilihan alat ukur ini sejalan dengan kajian yang menekankan pentingnya observabilitas performa melalui *tooling* resmi dan analisis profil eksekusi aplikasi [9] [11] [19].

3.6. Skenario Pengujian

Pengujian dilakukan berdasarkan tiga skenario utama. Skenario 1, yaitu perubahan status keranjang yang dilakukan dengan menambahkan dan menghapus item dari keranjang sebanyak 10 kali untuk mengamati pengaruh perubahan *state*. Sedangkan Skenario 2, yaitu pencarian *filter* data yang dilakukan dengan memasukkan kata kunci pada kotak pencarian untuk menguji pengaruh perubahan *filter* terhadap komponen item yang ditampilkan. Skenario 1 dan 2 memiliki kesamaan fungsi dalam pelaksanaannya yaitu untuk mengetahui jumlah *re-render* dan waktu *render* komponen yang mengalami perubahan *state*. Sementara pada Skenario 3, yaitu *mount* dan *unmount* katalog yang dilakukan dengan menampilkan dan menyembunyikan katalog sebanyak 5 kali untuk mendeteksi kemungkinan kebocoran memori menggunakan Flipper dengan plugin LeakCanary.

Setiap skenario pada Skenario 1 dan 2 diulang sebanyak 10 kali untuk memperoleh estimasi rata-rata metrik performa yang stabil. Hal ini sesuai dengan pengujian yang telah dilakukan oleh tim Chromium yang mengulangi *benchmark* power.mobile sebanyak 10 kali pada perangkat Android untuk mereduksi *noise* pengukuran [20]. Sedangkan Skenario 3 diulang sebanyak 3 kali karena bersifat detektif (ada/tidak) dan lebih menekankan pada konsistensi kemunculan *retained object*.

3.7. Teknik Analisis Data

Teknik analisis data yang digunakan adalah analisis deskriptif dan inferensial. Pada penelitian ini analisa deskriptif yang menghasilkan statistik digunakan untuk menggambarkan kecenderungan data hasil pengujian dengan parameter yang digunakan meliputi nilai rata-rata, nilai minimum, nilai maksimum, dan standar deviasi. Hasil statistik inilah yang akan digunakan untuk menentukan presentase penurunan kebutuhan *rendering* yang diperlukan komponen pada aplikasi.

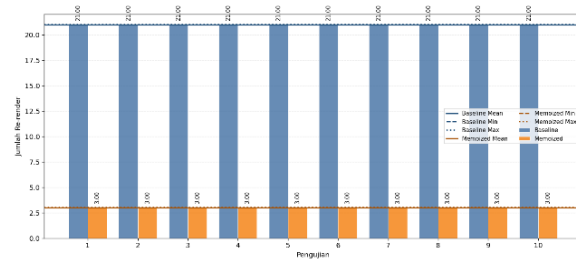
Selain analisa deskriptif, uji normalitas juga dilakukan pada penelitian ini menggunakan Shapiro-Wilk untuk mengetahui apakah data berdistribusi normal. Uji ini digunakan karena jumlah sampel relatif kecil dan umum dipakai dalam penelitian eksperimen semu [21]. Hasil dari uji normalitas inilah yang akan digunakan untuk menentukan *test* untuk uji hipotesis.

Setelah dilakukan uji normalitas, penelitian dilanjutkan dengan uji hipotesis. Apabila selisih data tidak bersifat konstan dan berdistribusi normal, maka pengujian hipotesis dilakukan menggunakan Paired T-Test. Namun apabila selisih data bersifat konstan atau tidak memenuhi asumsi normalitas, maka digunakan Wilcoxon Signed-Rank Test [22]. Hipotesis penelitian dirumuskan dengan H_0 , menyatakan tidak terdapat perbedaan yang signifikan antara mode *baseline* dan mode *memoized*, sedangkan H_1 menyatakan terdapat perbedaan yang signifikan. Keputusan hipotesis ditentukan pada taraf signifikansi 0,05. Dengan demikian H_0 ditolak apabila $p\text{-value} < 0,05$.

4. HASIL DAN PEMBAHASAN

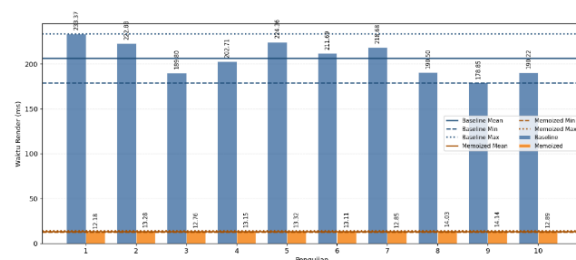
4.1. Hasil Pengujian

Pengujian dilakukan pada aplikasi *benchmark memoization* dalam dua kondisi, yaitu *baseline* dan *memoized*. Pengujian ini dilakukan berdasarkan 3 skenario yang telah dirancang. Pada skenario 1, pengujian dilakukan dengan menambahkan dan menghapus item dari keranjang pada 10 item yang berbeda untuk mengamati pengaruh perubahan *state* terhadap performa *rendering* aplikasi.



Gambar 2. Hasil jumlah *re-render* skenario 1

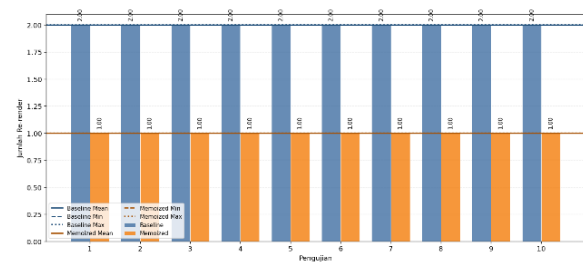
Dari grafik pada Gambar 2, terlihat bahwa data jumlah *re-render* pada mode *baseline* memiliki jumlah *re-render* yang jauh lebih tinggi dibandingkan dengan mode *memoized* untuk skenario 1.



Gambar 3. Hasil waktu *render* skenario 1

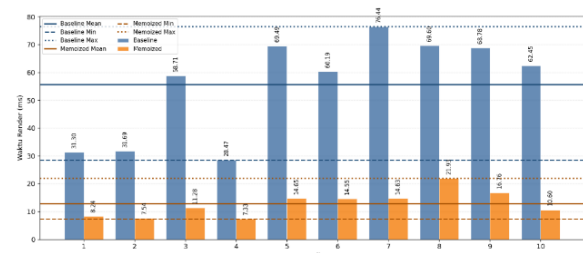
Dari grafik pada Gambar 3, terlihat bahwa data mode *memoized* memiliki waktu *render* yang jauh lebih rendah dibandingkan mode *baseline* untuk skenario 1.

Pada skenario 2, pengujian dilakukan dengan memasukkan kata kunci pencarian untuk memfilter item katalog dan mengamati perubahan performa *rendering* aplikasi.



Gambar 4. Hasil jumlah *re-render* skenario 2

Dari grafik pada Gambar 4, terlihat bahwa data jumlah *re-render* pada mode *baseline* memiliki jumlah *re-render* yang lebih tinggi dibandingkan dengan mode *memoized* untuk skenario 2.



Gambar 5. Hasil waktu *render* skenario 2

Dari grafik pada Gambar 5, terlihat bahwa data mode *memoized* memiliki waktu *render* yang lebih rendah dibandingkan mode *baseline* untuk skenario 2.

Pada skenario 3, pengujian dilakukan untuk mengetahui apakah terdapat indikasi kebocoran memori setelah penerapan *memoization* pada proses *show/hide* katalog secara berulang sebanyak 5 kali.

Tabel 4. Hasil pengujian kebocoran memori

Pengujian	Baseline	Memoized
1	Tidak ditemukan leak	Tidak ditemukan leak
2	Tidak ditemukan leak	Tidak ditemukan leak
3	Tidak ditemukan leak	Tidak ditemukan leak

Dari Tabel 4 dapat dilihat bahwa hasil pengujian tidak menemukan adanya leak pada seluruh pengujian, baik pada *baseline* maupun *memoized*. Hasil ini menunjukkan bahwa optimasi *memoization* tidak menimbulkan indikasi kebocoran memori.

4.2. Analisis Statistik Deskriptif

Analisis statistik deskriptif dilakukan untuk mengetahui kecenderungan data hasil pengujian.

Tabel 5. Statistik deskriptif jumlah *re-render* skenario 1

Parameter	Baseline	Memoized
Rata-rata	21,0000	3,0000
Minimum	21,0000	3,0000
Maksimum	21,0000	3,0000
Standar Deviasi	0,0000	0,0000

Dari Tabel 5 dapat dilihat bahwa hasil statistik deskriptif jumlah *re-render* skenario 1 menunjukkan bahwa data pada mode *baseline* bersifat konstan dengan nilai rata-rata 21 dan standar deviasi $< 0,001$. Sedangkan pada mode *memoized*, nilai rata-rata turun menjadi 3 dengan standar deviasi yang sama juga dengan mode *baseline* yaitu $< 0,001$. Hal ini menunjukkan bahwa *memoization* secara konsisten mampu menekan jumlah *re-render* sebesar 85,71% pada skenario 1.

Tabel 6. Statistik deskriptif waktu *render* skenario 1

Parameter	Baseline	Memoized
Rata-rata	206,3069	13,1712
Minimum	186,0000	12,1786
Maksimum	233,3727	14,1442
Standar Deviasi	18,4481	0,5838

Dari Tabel 6 dapat dilihat bahwa hasil statistik deskriptif waktu *render* skenario 1 menunjukkan bahwa mode *memoized* memiliki waktu *render* yang lebih rendah dibandingkan mode *baseline* dengan rata-rata waktu *render* pada mode *baseline* adalah 206,30 ms, sedangkan pada mode *memoized* turun menjadi 13,17 ms. Hal ini membuktikan bahwa penerapan *memoization* dapat menurunkan waktu *render* sebesar 93,62% pada skenario 1. Sedangkan standar deviasi pada mode *memoized* juga lebih kecil, yang menunjukkan bahwa hasil waktu *render* pada mode *memoized* lebih stabil.

Tabel 7. Statistik deskriptif jumlah *re-render* skenario 2

Parameter	Baseline	Memoized
Rata-rata	2,0000	1,0000
Minimum	2,0000	1,0000
Maksimum	2,0000	1,0000
Standar Deviasi	0,0000	0,0000

Dari Tabel 7 dapat dilihat bahwa hasil statistik deskriptif jumlah *re-render* skenario 2 menunjukkan bahwa data pada mode *baseline* bersifat konstan dengan nilai rata-rata 2 dan standar deviasi $< 0,001$. Sedangkan pada mode *memoized*, nilai rata-rata turun menjadi 1 dengan standar deviasi yang sama juga dengan mode *baseline* yaitu $< 0,001$. Hal ini menunjukkan bahwa *memoization* secara konsisten mampu menekan jumlah *re-render* sebesar 50% pada skenario 2.

Tabel 8. Statistik deskriptif waktu *render* skenario 2

Parameter	Baseline	Memoized
Rata-rata	55,7121	12,7515
Minimum	28,4710	7,3334
Maksimum	76,4396	21,9303
Standar Deviasi	18,1744	4,6474

Dari Tabel 8 dapat dilihat bahwa hasil statistik deskriptif waktu *render* skenario 2 menunjukkan bahwa mode *memoized* memiliki waktu *render* yang lebih rendah dibandingkan mode *baseline* dengan rata-

rata waktu *render* pada mode *baseline* adalah 55,71 ms, sedangkan pada mode *memoized* turun menjadi 12,75 ms. Hal ini membuktikan bahwa penerapan *memoization* dapat menurunkan waktu *render* sebesar 77,11% pada skenario 2. Sedangkan standar deviasi pada mode *memoized* juga lebih kecil, yang menunjukkan bahwa hasil waktu *render* pada mode *memoized* lebih stabil.

4.3. Uji Hipotesis

Pengujian hipotesis dilakukan untuk mengetahui apakah terdapat perbedaan performa yang signifikan antara mode *baseline* dan mode *memoized*. Sebelum pengujian hipotesis dilakukan, terlebih dahulu dilakukan uji normalitas menggunakan Shapiro-Wilk terhadap data selisih berpasangan. Hasil uji normalitas digunakan sebagai dasar untuk menentukan jenis uji statistik yang sesuai. Apabila selisih berpasangan berdistribusi normal, maka digunakan uji Paired T-Test. Namun, apabila selisih data bersifat konstan atau tidak memenuhi asumsi normalitas, maka digunakan uji Wilcoxon Signed-Rank sebagai alternatif nonparametrik.

Tabel 9. Hasil Uji Normalitas dan Uji Hipotesis

Metrik	Uji Normalitas	Uji Hipotesis	p-value
Jumlah <i>Re-render</i> Skenario 1	Data konstan	Wilcoxon Signed-Rank	0,00195
Waktu <i>Render</i> Skenario 1	$p = 0,54197$ ($>0,05$)	Paired T-Test	0,00000
Jumlah <i>Re-render</i> Skenario 2	Data konstan	Wilcoxon Signed-Rank	0,00195
Waktu <i>Render</i> Skenario 2	$p = 0,05822$ ($>0,05$)	Paired T-Test	0,00001

Dari Tabel 9 dapat dilihat bahwa seluruh pengujian hipotesis menghasilkan p-value di bawah 0,05 sehingga H_0 ditolak. Artinya, perbedaan performa antara *baseline* dan *memoized* bersifat signifikan secara statistik. Dengan demikian, optimasi *memoization* bukan hanya memberi peningkatan yang terlihat secara visual, tetapi juga terbukti melalui analisis inferensial.

4.4. Pembahasan

Hasil penelitian menunjukkan bahwa penerapan *memoization* menggunakan React.memo dan useMemo memberikan pengaruh yang sangat besar terhadap peningkatan kinerja *rendering* aplikasi React Native Android. Pada pengujian Skenario 1, mode *memoized* berhasil menurunkan rata-rata *re-render* sebesar 85,71%. Selain itu, rata-rata waktu *render* juga mengalami penurunan sebesar 93,62%. Penurunan tersebut menunjukkan bahwa React.memo efektif dalam mencegah *render* ulang yang tidak diperlukan pada komponen katalog ketika *state* keranjang

berubah. Sedangkan pada pengujian Skenario 2, mode *memoized* berhasil menurunkan rata-rata *re-render* sebesar 50%. Selain itu, rata-rata waktu *render* juga mengalami penurunan sebesar 77,11%. Hasil ini memperlihatkan bahwa penerapan *memoization* tetap efektif pada proses penyaringan data yang memicu pembaruan tampilan daftar.

Meskipun pengujian Skenario 1 dan Skenario 2 menggunakan pengukuran dengan metrik yang sama yaitu jumlah *re-render* dan waktu *render*, hasil pengujian dua skenario tersebut memberikan nilai presentase penurunan yang cukup berbeda. Hal ini dikarenakan pada Skenario 1 pengujian dilakukan dengan mengubah *state* item keranjang yang berbeda sebanyak 10 kali disetiap pengujiannya yang mengakibatkan banyak item pada katalog mengalami perubahan, sedangkan pada Skenario 2 dilakukan pengujian dengan melakukan pencarian item dengan proses penyaringan data yang mengakibatkan perubahan tampilan pada daftar item hanya sekali saja. Sehingga dapat dikatakan bahwa semakin banyak perubahan yang terjadi pada suatu daftar item maka semakin besar pula keuntungan yang dirasakan dari penggunaan *memoization* untuk optimasi performa aplikasi.

Pada pengujian *memory leak*, Flipper dengan *plugin* LeakCanary tidak menemukan adanya objek yang tertahan di memori pada kedua mode. Hasil ini menunjukkan bahwa penerapan *memoization* tidak menimbulkan kebocoran memori dan tetap aman digunakan dalam skenario penelitian ini.

Secara keseluruhan, hasil penelitian membuktikan bahwa penerapan *memoization* pada aplikasi React Native Android yang memiliki komponen daftar dan komputasi berat dapat memberikan peningkatan performa secara signifikan dan *memoization* tidak berdampak negatif terhadap stabilitas memori aplikasi karena tidak menimbulkan indikasi kebocoran memori (*memory leak*).

5. KESIMPULAN DAN SARAN

Berdasarkan hasil pengujian dan analisis yang telah dilakukan, penerapan teknik *memoization* menggunakan React.memo dan useMemo terbukti mampu meningkatkan kinerja *rendering* aplikasi Android berbasis React Native secara signifikan. Pada pengujian Skenario 1, rata-rata jumlah *re-render* menurun sebesar 85,71% dan rata-rata waktu *render* menurun sebesar 93,62%. Pada pengujian Skenario 2, rata-rata jumlah *re-render* menurun sebesar 50% dan rata-rata waktu *render* menurun sebesar 77,11%. Sementara hasil uji hipotesis dengan Wilcoxon Signed-Rank dan Paired T-Test menghasilkan nilai $p\text{-value} < 0,05$ pada seluruh pengujian, sehingga H_0 ditolak dan dapat dinyatakan bahwa terdapat perbedaan yang signifikan antara mode *baseline* dan mode *memoized*. Sedangkan pada pengujian *memory leak* menggunakan Flipper dengan *plugin* LeakCanary, tidak ditemukan adanya kebocoran memori pada kedua mode, hal ini menunjukkan bahwa

penerapan *memoization* tidak berdampak negatif terhadap stabilitas memori. Sehingga dapat ditarik kesimpulan bahwa penerapan *memoization* paling efektif digunakan pada komponen yang sering mengalami *re-render* akibat perubahan *state* pada komponen induk, memiliki *props* yang relatif stabil, serta melibatkan proses komputasi yang cukup berat. Pada skenario dengan perubahan *state* yang lebih sering dan kompleks, manfaat *memoization* terhadap peningkatan performa *rendering* terlihat lebih signifikan dibandingkan skenario dengan perubahan data yang lebih sederhana. Dengan demikian, penggunaan *memoization* sebaiknya diterapkan secara selektif berdasarkan hasil *profiling* performa dan karakteristik komponen yang digunakan.

DAFTAR PUSTAKA

- [1] P. Nawrocki, K. Wrona, M. Marczak, and B. Sniezynski, "A Comparison of Native and Cross-Platform Frameworks for Mobile Applications," *Computer (Long. Beach. Calif.)*, vol. 54, no. 3, pp. 18–27, Mar. 2021, doi: 10.1109/MC.2020.2983893.
- [2] G. Jošt and V. Taneski, "State-of-the-Art Cross-Platform Mobile Application Development Frameworks: A Comparative Study of Market and Developer Trends," *Informatics*, vol. 12, no. 2, p. 45, Apr. 2025, doi: 10.3390/informatics12020045.
- [3] F. Mushtaq, F. Azam, and M. W. Anwar, "Performance Comparison of Single Code Base Development Tools: Flutter, React Native, and Xamarin," in *2024 14th International Conference on Software Technology and Engineering (ICSTE)*, IEEE, Aug. 2024, pp. 17–23. doi: 10.1109/ICSTE63875.2024.00011.
- [4] A. Souha, L. Benaddi, C. Ouaddi, and A. Jakimi, "Comparative analysis of mobile application Frameworks: A developer's guide for choosing the right tool," *Procedia Comput. Sci.*, vol. 236, pp. 597–604, 2024, doi: 10.1016/j.procs.2024.05.071.
- [5] A. Bjørn-Hansen, C. Rieger, T.-M. Grønli, T. A. Majchrzak, and G. Ghinea, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," *Empir. Softw. Eng.*, vol. 25, no. 4, pp. 2997–3040, Jul. 2020, doi: 10.1007/s10664-020-09827-6.
- [6] W. Oliveira, B. Moraes, F. Castor, and J. P. Fernandes, "Analyzing the Resource Usage Overhead of Mobile App Development Frameworks," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA: ACM, Jun. 2023, pp. 152–161. doi: 10.1145/3593434.3593487.
- [7] V. Saxena, V. Danwa, and D. Mehta, "Comprehensive Insights into Cross-Platform Mobile Development Frameworks," in

- Proceedings of the 6th International Conference on Information Management & Machine Intelligence*, New York, NY, USA: ACM, Dec. 2024, pp. 1–6. doi: 10.1145/3745812.3745824.
- [8] M. Hort, M. Kechagia, F. Sarro, and M. Harman, “A Survey of Performance Optimization for Mobile Applications,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2879–2904, Aug. 2022, doi: 10.1109/TSE.2021.3071193.
- [9] E. A. Ratuszniak, “Analysis of modern optimization methods in ‘React,’” *Программные системы и вычислительные методы*, no. 3, pp. 1–9, Mar. 2025, doi: 10.7256/2454-0714.2025.3.75278.
- [10] P. Garg, B. Yadav, S. Gupta, and B. Gupta, “Performance Analysis and Optimization of Cross Platform Application Development Using React Native,” 2023, pp. 559–567. doi: 10.1007/978-981-19-9304-6_51.
- [11] “React Native DevTools · React Native.” Accessed: Apr. 23, 2026. [Online]. Available: <https://reactnative.dev/docs/0.81/react-native-devtools>
- [12] M. Markowski and J. Smółka, “A comparative analysis of the Flutter and React Native frameworks,” *Journal of Computer Sciences Institute*, vol. 29, pp. 346–351, Dec. 2023, doi: 10.35784/jcsi.3794.
- [13] “GitHub - facebook/flipper: A desktop debugging platform for mobile developers. · GitHub.” Accessed: May 07, 2026. [Online]. Available: <https://github.com/facebook/flipper>
- [14] T. Dorfer, L. Demetz, and S. Huber, “Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features,” *Procedia Comput. Sci.*, vol. 175, pp. 189–196, 2020, doi: 10.1016/j.procs.2020.07.029.
- [15] S. Huber, L. Demetz, and M. Felderer, “Analysing the Performance of Mobile Cross-platform Development Approaches Using UI Interaction Scenarios,” 2020, pp. 40–57. doi: 10.1007/978-3-030-52991-8_3.
- [16] M. Mahendra and B. Anggorojati, “Evaluating the performance of Android based Cross-Platform App Development Frameworks,” in *Proceedings of the 6th International Conference on Communication and Information Processing*, New York, NY, USA: ACM, Nov. 2020, pp. 32–37. doi: 10.1145/3442555.3442561.
- [17] A. S. Pramayoga, S. Kurniawan, and Y. F. Ashari, “Performance Analysis of Cross-Platform Mobile Frameworks: A Case Study on an AI Chatbot for Academic Regulation Services,” in *2025 8th International Conference on Informatics and Computational Sciences (ICICoS)*, IEEE, Oct. 2025, pp. 1–4. doi: 10.1109/ICICoS68590.2025.11330137.
- [18] Ł. Kurant and J. Bylina, “Impact of Selected JavaScript Engines on the Performance of Mobile Hybrid Applications,” *Journal of Software: Evolution and Process*, vol. 38, no. 2, Feb. 2026, doi: 10.1002/smr.70086.
- [19] “LeakCanary.” Accessed: May 07, 2026. [Online]. Available: https://square.github.io/leakcanary/?utm_source=chatgpt.com
- [20] 3573f412a8a780cb7994639f176b07e65d869283 - chromium/src/testing - Git at Google.” Accessed: May 12, 2026. [Online]. Available: <https://chromium.googlesource.com/chromium/src/testing/+3573f412a8a780cb7994639f176b07e65d869283>
- [21] R. R. de Souza, M. Toebe, A. C. Mello, and K. C. Bittencourt, “Sample size and Shapiro-Wilk test: An analysis for soybean grain yield,” *European Journal of Agronomy*, vol. 142, p. 126666, Jan. 2023, doi: 10.1016/j.eja.2022.126666.
- [22] T. Rietveld and R. van Hout, “The paired t test and beyond: Recommendations for testing the central tendencies of two paired samples in research on speech, language and hearing pathology,” *J. Commun. Disord.*, vol. 69, pp. 44–57, Sep. 2017, doi: 10.1016/j.jcomdis.2017.07.002.