Perbandingan Arsitektur Runtime WebAssembly dan Native Code pada Pengembangan Website

Ahmd Mufahras Li Alfazh Assardew [1], I Made Suartana[2]

[1,2] Program Studi Teknik Informatika/Teknik Indormatika, Universitas Negeri Surabaya
[1]ahmdmufahras.22020@mhs.unesa.ac.id
[2]madesuartana@unesa.ac.id

***Abstrak*— Perkembangan web modern menuntut performa yang tinggi, terutama untuk aplikasi editor yang membutuhkan pengolahan gambar yang cepat. Untuk mencapai efisiensi ini terdapat beberapa teknologi arsitektur yang dapat digunakan diantaranya WebAssembly (wasm) dengan basis client side dan native code (framework Actix web) dengan basis server side. Penelitian ini bertujuan untuk menganalisis dan membandingkan kinerja dari wasm dan native, dengan fokus pengujian pada waktu pemrosesan, pengaruh internet, ukuran file dan penggunaan CPU. Hasil pengujian menunjukkan implementasi wasm unggul dalam waktu pemrosesan karena tidak melakukan request dan menunggu response dari server seperti implementasi native. Kinerja wasm juga lebih stabil pada perubahan kecepatan internet dengan perbedaan waktu hanya 0.02 s pada tiap perubahan kecepatan internet, sedangkan pada implementasi native mengalami penurunan yang seharusnya bisa kurang dari 2s pada kecepatan internet 10 Mbps menjadi lebih dari 1 menit pada kecepatan internet 0.7 Mbps. Pada pengujian penggunaan CPU client implementasi native memiliki keunggulan karena proses dilakukan pada server sedangkan pada wasm memerlukan setidaknya 1 core CPU dan menyebabkan blocking UI threads. Pengujian menggunakan ukuran FHD dan 4K menunjukkan bahwa implementasi Native cukup efektif pada pengujian non-sequence namun pada pengujian sequence implementasi WASM lebih baik. Kesimpulan dari penelitian ini memberikan acuan bagi pengembang dalam menentukan arsitektur perangkat lunak, wasm direkomendasikan untuk beban kerja yang membutuhkan kecepatan dan stabilitas performa terlepas dari kondisi jaringan, sementara native lebih cocok untuk prioritas pada load halaman cepat dengan pemrosesan pada server.**

***Kata Kunci*— Moder web, WebAssembly, Actix web, Native, Performa, Arsitektur.**

I. PENDAHULUAN

Pada awal diperkenalkan platform web difungsikan sebagai tempat pertukaran document, namun dengan perkembangan teknologi web browser seperti HTML5, CSS3, dan JavaScript platform web juga berkembang menjadi aplikasi pemrosesan gambar, audio, video, modelling 3D, dan pemrosesan yang memerlukan performa yang cukup tinggi [1]. Namun sebagus apapun perkembangan JavaScript masih terdapat kelemahan pada performa runtime [2], permasalahan ini menjadi tantangan paling besar dalam pengembangan aplikasi berbasis web dengan kompleksitas yang tinggi.

Solusi dari lemahnya proses runtime seperti pada aplikasi pengolahan gambar yang membutuhkan performa tinggi hadirlah pendekatan menggunakan native code dalam membangun aplikasi berbasis web mulai berkembang, pendekatan ini menawarkan solusi diantaranya waktu eksekusi yang cukup efisien, efisien dalam penggunaan sumber daya, dan type save. Hal ini yang diterapkan pada Actix, sebuah framework website yang dibangun oleh komunitas Rust menggunakan bahasa pemrograman Rust, dari pengujian yang dilakukan oleh Tech Power ditemukan bahwa framework Actix web memiliki performa sebesar 83.1% yang mana lebih besar daripada nodejs dengan performa sebesar 13% [2]. Actix web dapat dijalankan dengan HTTP server yang disediakan pada file eksekusi, dengan kata lain tanpa adanya server HTTP lain Actix web memungkinkan untuk menjalankan layanan HTTP/1, HTTP/2, maupun LTS (HTTPS) . Namun Actix web memungkinkan cukup rumit untuk dipelajari terutama dari perspektif programer yang belum pernah menggunakan Rust sebelumnya.

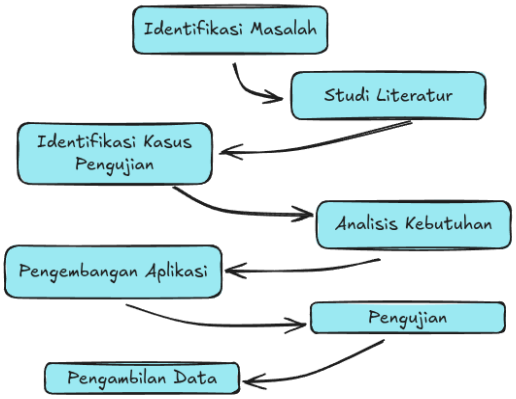
Solusi lain yang dapat ditawarkan yaitu pendekatan menggunakan WebAssembly (WASM). WebAssembly pertama kali diperkenalkan pada tahun 2015, secara konsep teknologi WebAssembly menggabungkan ActiveX, PNaCl, dan asm.js sebagai arsitektur dasar [3]. Dengan menggunakan WebAssembly programer hanya perlu menuliskan fungsi yang mereka butuhkan menggunakan bahasa pemrograman low-level seperti C, C++ atau Rust. WebAssembly menggunakan bahasa low-level tersebut untuk membangun fungsi yang kemudian fungsi tersebut akan dikompilasi menuju format file biner, dan hasil dari kompilasi tersebut dipanggil menggunakan JavaScript, JavaScript masih dibutuhkan karena WebAssembly tidak bisa berjalan sendiri dan mempermudah untuk komunikasi antara FrontEnd dan WebAssembly[4]. Dengan adanya hal itu WebAssembly dapat mempermudah programer dalam membangun aplikasi website dengan performa yang cukup tinggi terutama yang belum pernah menggunakan bahasa low-level [5].

Penelitian ini akan membandingkan waktu eksekusi, waktu eksekusi dengan pengaruh kecepatan internet, ukuran file, dan besar penggunaan resource CPU pada implementasi WebAssembly dengan native code (Actix web) dengan pengujian pada beberapa algoritma image processing pada Aplikasi pengolahan gambar pada browser. Harapan dari penelitian ini dapat memberikan penulis wawasan terhadap performa yang didapat ketika menggunakan WebAssembly dan native code, dan penelitian ini dapat menjadi acuan peneliti atau programer lain saat membangun website yang membutuhkan performa tinggi.

II. METODEOLOGI PENELITIAN

A. Metode Penelitian

Penelitian ini menggunakan pendekatan kuantitatif menggunakan metode eksperimen, dimana melakukan pengujjian keoada WebAssembly dan Natice code untuk melakukan eksekusi menggunakan beberapa algoritma *image processing* yang diterapkan pada *WebApp image editor*.



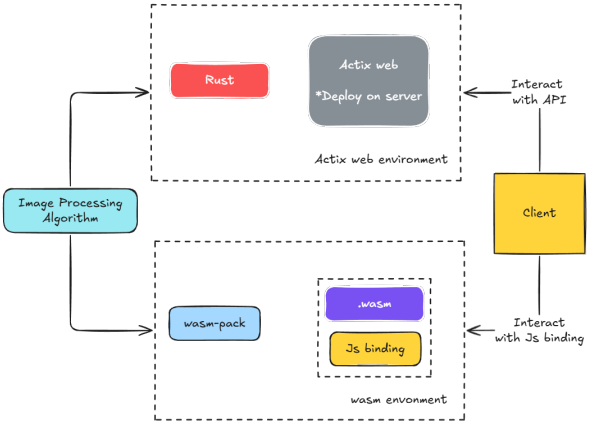
Gbr. 1 Alur Metode Penelitian

Dari Grb 1 dapat diketahui bahwa alur dari penelitian dimulai dari mengidentifikasi permaslaahan apa yang akan dihadapi, kemudian mempersiapkan kasus pengujian yang akan dilakukan, selanjutnya melakukan pengembangan aplikasi sesuai dengan identifikasi masalah dan kasus pengujian yang dibuat sebelumnya, dan yang terakhi melakukan pengujian dan pengambilan data dari pengeditan yang dilakukan sebelumnya.

Penelitian ini memiliki fokus dalam melakukan perbandingan performa WebAssembly dengan Native Code dengan 4 pengujian utama *time execution*, *time execution* dengan pengaruh internet, ukuran *resource* file pada client, dan besar penggunaan *resource* CPU.

B. Pengembangan Aplikasi

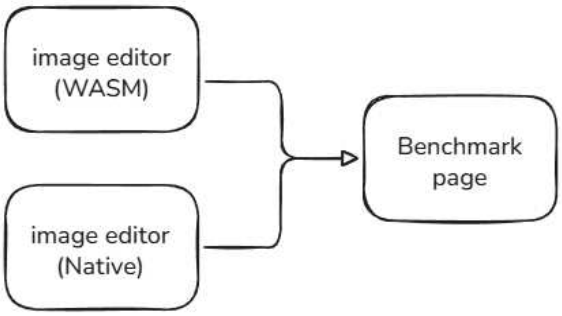
Aplikasi WebApp Image Editor akan mengimplementasikan arsitektur WebAssembly dan Native code, kedua implementasi tersebut akan menggunakan algoritma *image processing* yang sama, namun beda dalam penerapannya sebagai gambaran bisa dilihat pada Gbr 2.



Gbr. 2 Arsitektur WebAssembly dan Acix web.

Pada Gbr 2 dapat dilihat aplikasi akan menggunakan algoritma yang sama dengan dua arsitektur yang berbeda, dimana pada implementasi WebAssembly kode *image processing* akan di *compile* menggunakan wasm-pack kemudian menghasilkan satu package dengan Kumpulan file diantaranya JavaScrip dan WebAssembly [6], kedua file tersebut yang menjalankan fitur utama dari aplikasi dimana proses *image processing* akan dijalankan pada WebAssembly dan file JavaScript akan menjadi *bridging* antara FrontEnd dan WebAssembly. Sedangkan pada implementasi Native akan memanfaatkan framework Actix web, jadi kode *image processing* yang telah dibangun sebelumnya akan dijalankan pada framework yang kemudian dideploy pada server, interaksi yang dilakukan oleh client hanya melalui API endpoint yang dihasilkan actix web [7]. Alasan mengapa menggunakan algoritma yang sama yaitu untuk mempertahankan konsistensi kode, dan menjaga fokus dari penelitian yang membandingkan kedua arsitektur bukan dari algoritma mana yang lebih cepat.

Aplikasi WebApp Image Editor yang dikembangkan akan mencakup implementasi dari WebAssembly dan Native code (Axtic Web), WebApp ini dibangun menggunakan *framework* NextJs, dengan memanfaatkan teknologi react hook demi mempermudah *lifecycle* pada pembangunan WebApp, pada aplikasi ini akan mencakup beberapa halaman diantaranya *image editor* dengan implementasi WebAssembly, *image editor* dengan implementasi Native Code, dan yang terakhir halaman untuk hasil *benchmark* yang dilakukan. Alur aplikasi akan terlihat seperti Gbr 3.



Gbr. 3 Alur aplikasi.

Pada halaman *image editor* yang dibangun akan menerapkan beberapa algoritma *image processing*, algoritma yang akan digunakan dibagi menjadi dua kelompok berdasarkan cara penggunaan pada aplikasi diantaranya *sequence* dan *non-sequence*. Kelompok *sequence* mencakup manipulasi ketajaman gambar, manipulasi saturasi, manipulasi *temperature*, manipulasi *tint*, manipulasi *exposure*, dan manipulasi *contrast*. Kelompok *non-sequence* hanya terdapat algoritma *transfer color*. Perbedaan dari kedua jenis kelompok algoritma tersebut yaitu pada penggunaan pada WebApp, kelompok *srquence* menggunakan slider atau input *range* yang mana input yang diberikan menjadi beruntun, sedangkan kelompok *non-sequence* yaitu algoritma *transfer color* ia membutuhkan inputan berupa referensi gambar sehingga tugas

yang dilakukan tidak bisa bertumpuk seperti kelompok *srquence*.

Pada halaman image editor baik implementasi WebAssembly maupun Native code akan mencakup beberapa fitur diantaranya komponen perubahan algoritma *sequence* menggunakan komponen *slider* atau *range*, dan kopmponen perubahan *non-sequence* yang mencakup algoritma *transfer color*, yang mana pada komponen ini terdapat input sebagai gambar referensi dan tombol sebagai *trigger* algoritma.

Selain fitur untuk *image processing* halaman editor juga dilengkapi dengan menu *benchmark*, menu *benchmark* ini memiliki tujuan untuk memulai dan memonitor apakah percobaan pada aplikasi sudah mencukupi untuk mendapatkan hasil benchmark, data dari percobaan algoritma yang telah dilakukan akan disimpan pada localstorage yang kemudian akan ditampilkan pada halaman benchmark, detail data yang disimpan diantaranya ukuran pixel gambar, waktu pemrosesan, jenis algoritma yang dicoba, dan kecepatan internet jika ada. Dalam menunjang pengujian penelitian ini akan menggunakan *Google Dev Tools* sebagai alat untuk monitoring pengujian fitur yang akan digunakan diantaranya *log activity*, *performance tab*, *network activity*, dan *local storage* [8].

C. *Skenario Pengujian*

Bagian scenario pengujian akan dibagi menjadi 4 skenario utama diantaranya *time execution*, *time execution* dengan pengaruh internet, ukuran file dan *load page*, dan beban CPU.

Pengujian akan dilakukan langsung pada webapp yang telah dibangun. Pengujian *time execution* dan *time execution* dengan pengaruh kecepatan internet akan dibagi menjadi 2 pengujian algoritma *sequence* dan algoritma *non-sequence*, pada pengujian *sequence* akan mengambil 5 hasil *time execution* dan pada pengujian *non-sequence* akan mengambil 10 hasil *time execution* dari hjasil tersebut akan diambil rata-rata dan perubahan yang terjadi pada tiap algoritma. Sebelum melakukan benchmark, pengguna harus menekan tombol mulai benchmark supaya data tercatat saat melakukan pengeditan, sebagai catatan untuk pengujian *time execution* dengan pengaruh kecepatan internet sebelum menekan tombol mulai benchmark pengguna harus mengaktifkan *toggle* untuk mencatat kecepatan internet saat ini pula.

Pengujian selanjutnya mengukur load time dan ukuran file pada client, pengujian ini akan memanfaatkan *chrome dev tools* pada menu *network*, skenario pengujiannya yaitu buka *chrome dev tools* terlebih dahulu pada halaman editor lakukan *reload page* sehingga mendapatkan data ukuran resource yang digunakan pada client dan berapa waktu yang dibutuhkan untuk melakukan *load page*.

Pengujian untuk memeriksa penggunaan CPU dari kedua implementasi akan menggunakan *tools* dari chrome dengan nama *browser task manager*, skenarionya buka *browser task manager* kemudian lakukan pengeditan gambar dari awal sampai akhir, catat seluruh perubahan untuk mendapatkan penggunaan CPU pada bagian user dari kedua implementasi. Untuk implementasi Native code terdapat pengujian tambahan untuk memeriksa penggunaan CPU pada sisi server, tools yang digunakan cukup menggunakan terminal dengan perintah *top*, lakukan pencatatan saat terdapat perubahan seperti yang sebelumnya.

III. Hasil Dan Pembahasan

Proses pengujian akan mengikuti skenario perbandingan antara implementasi WebAssembly dan Native code yang telah disusun seblumnya. Dimana setiap scenario akan melakukan pengeditan sebanyak 5 kali percobaan untuk algoritma *transfer color* dan 10 kali untuk algoritma yang lain. Berikut hasil dari pengujian yang telah dilakukan pada penelitian.

A. *Pengujian Time Execution*

Pengujian ini akan menggunakan gambar dengan ukuran 685px × 1024px. Pada pengujian algoritma *transfer color* menggunakan beberapa ukuran gambar dari 768px × 869px sampai gambar dengan ukuran 1042px × 1461px. Dari pengujian tersebut didapatkan hasil implementasi WebAssembly memakan waktu 1,96s sampai 2,77s. Sedangkan pada implementasi Native memerlukan waktu sebanyak 1,14s sampai 1,47s, hal ini dapat terjadi dikarenakan pemrosesan yang cukup besar pada client akan berpengaruh juga dengan resource yang dimiliki client, selain hal tersebut WebAssembly menjalankan prosesnya pada *sandbox* browser sehingga menimbulkan overhead pada saat validasi antara kedua lingkungan tersebut, Pada penelitian [9] disebutkan pula permasalahan dari WebAssembly yaitu masih belum mendukung multithreading sehingga hanya memanfaatkan performa dari CPU sehingga ketika terdapat tugas berlebih maka akan terjadi *overhead* pada sisi client. Hasil dari pengujian dapat dilihat pada Gbr 4 dibawah ini.

Gbr. 4 Hasil *Time Execution* Transfer color.

Untuk pengujian kelompok *sequence* implementasi WebAssembly membutuhkan waktu pemrosesan sebesar 0,14s sampai 1,12s. Sedangkan untuk implementasi Native membutuhkan waktu 1,13s sampai 1,79s menggunakan gambar dengan ukuran yang sama. Hasil selengkapnya dari

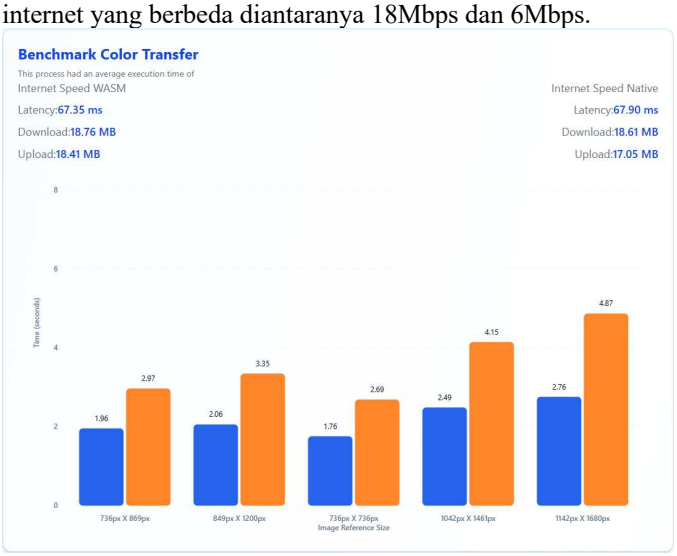
pengujian *time execution* selain algoritma *transfer color* dapat dilihat pada Tabel I.

TABEL I
PENGUJIAN TIME EXECUTION

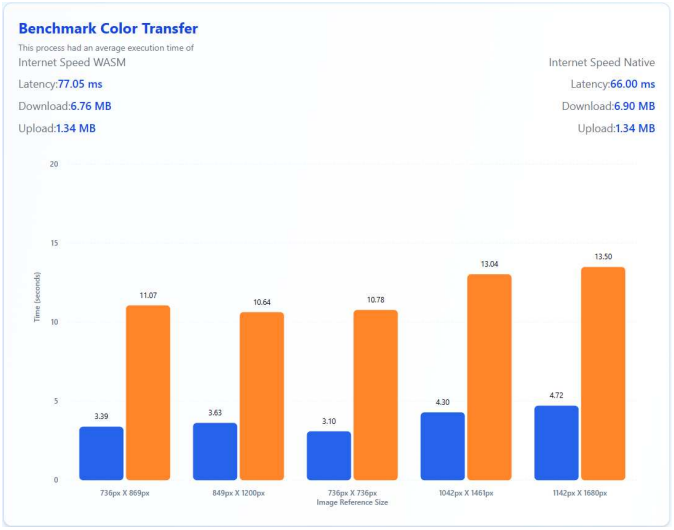
Algorithm	Rata-rata Time Exevution	
	WASM(*s*)	Native Code(*s*)
Sharpness	0.3	1.05
Saturation	0.214	0.802
Temperature	0,175	0.955
Tint	0.162	0.843
Contrast	0.17	0.688
Exposure	1.063	1.233

B. *Pengujian Time Execution dengan pengaruh Internet*

Sama seperti pengujian sebelumnya hanya terdapat tambahan untuk pengujian menggunakan beberapa kecepatan internet yang berbeda diantaranya 18Mbps dan 6Mbps.



Gbr. 5 Hasil *Time Execution* Transfer color dengan 18Mbps kecepatan internet.



Gbr. 6 Hasil *Time Execution* Transfer color dengan 6Mbps kecepatan internet.

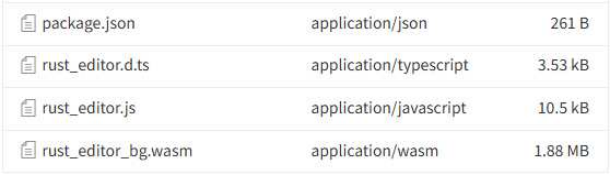
TABEL II
PENGUJIAN TIME EXECUTION DENGAN KECEPATAN INTERNET

Algorithm	Rata-rata Time Execution			
	18Mbps		6Mbps	
	WASM(*s*)	Native(*s*)	WASM(*s*)	Native(*s*)
Sharpness	0.3	6.5	0.586	50.26
Saturation	0.16	1.19	0.22	44.6
Temperature	0.16	1.1	0.21	49
Tint	0.17	1.1	0.23	46.26
Contrast	0.163	1.1	0.33	43.67
Exposure	1.1	1.6	1.75	47.2

Dari hasil pengujian *time execution* dengan pengaruh kecepatan internet pada Gbr 4, Gbr 5, dan Tabel II didapatkan hasil pada kecepatan internet 18Mbps implementasi Native mengalami lonjakan waktu dengan kisaran 0.2s sampai 5s, kemudian pada kecepatan internet 6Mbps mengalami lonjakan waktu pemrosesan sangat tinggi dibanding sebelumnya yang membutuhkan waktu pemrosesan sebanyak 43s sampai 50s lamanya. Sedangkan pada implementasi WebAssembly memiliki waktu pemrosesan yang cukup stabil dibanding implementasi Native Code, hal ini dapat terjadi dikarenakan pada implementasi Native berjalan pada server sedangkan implementasi WebAssembly berjalan pada bagian client yang membutat kececpatan internet *upload* dan *download* sangat berpengaruh pada impelmentasi Native.

C. *Pengujian ukuran file dan load time*

Perbandingan ini akan mengambil ukuran file yang digunakan untuk melakukan pemrosesan baik server maupun client, dan berapa total waktu yang dibutuhkan untuk melakukan *load page* pada tiap implementasi.



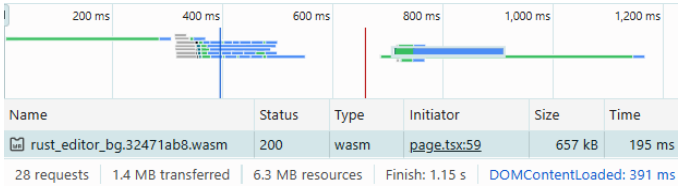
Gbr. 7 List file package WebAssembly.

```
total 13992
-rwxr-xr-x  2 alfazh alfazh 14245680 Nov 17 09:45 actix-img-editor
-rw-r--r--  1 alfazh alfazh      380 Nov 17 09:45 actix-img-editor.d
drwxr-xr-x 55 alfazh alfazh     4096 Oct 13 10:45 build
drwxr-xr-x  2 alfazh alfazh    61440 Nov 17 09:45 deps
drwxr-xr-x  2 alfazh alfazh     4096 Oct 13 10:45 examples
drwxr-xr-x  2 alfazh alfazh     4096 Oct 13 10:45 incremental
```

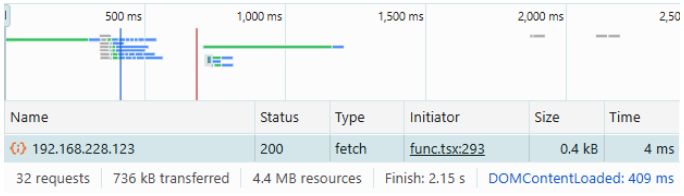
Gbr. 8 List file build actix-web pada server.

Bagian ini akan memaparkan total ukuran file yang akan digunakan pada client untuk implementasi WebAssembly dan server untuk implementasi Native. Pada Gbr 7 merupakan list file yang akan di gunakan pada bagian client, yang mana pada list tersebut terdapat beberapa file package.json untuk informasi package, file TypeScript dan JavaScript sebagai bridging antara FrontEnd dan WebAssembly, yang terakhir file WebAssembly yang mana merupakan tempat seluruh algoritma *image processing* berada. Total yang akan digunakan oleh user sebanyak 1.9MB. Selanjutnya untuk

implementasi Native pada Gbr 8 merupakan list file yang berada pada bagian server dari keseluruhan file yang akan digunakan yaitu file *actix-img-editor* yang merupakan hasil build dari framework dengan ukuran file 14MB, namun file tersebut hanya berada pada bagian server.



Gbr. 9 Load time pada implementasi WebAssembly.

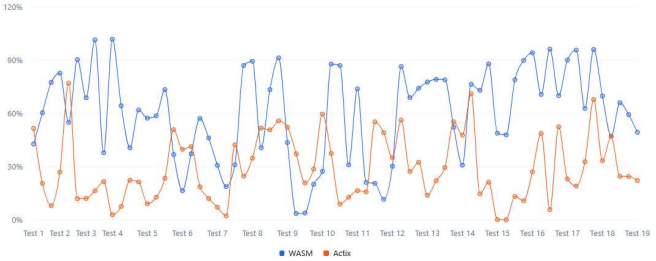


Gbr. 10 Load time pada implementasi Native.

Selanjutnya pada Gbr 9 dan Grb 10 merupakan hasil yang didapatkan dari Google dev tools dengan menu Network, dari gambar tersebut dapat diketahui untuk implementasi WebAssembly mengkonsumsi resource lebih banyak disbanding implementasi Native, yang mana pada implementasi WebAssembly menggunakan resource sebanyak 6.3MB sedangkan untuk implementasi Native membutuhkan resource sebanyak 4.4MB. Selanjutnya waktu yang dibutuhkan untuk *load page* pada implementasi WebAssembly memiliki waktu yang cukup singkat dikisaran 1.15s sedangkan untuk implementasi Native membutuhkan waktu setidaknya 2.15s. Dari kedua hal tersebut terjadi karena yang pertama resource pada implementasi WebAssembly lebih banyak dikarenakan pada browser diahruskan untuk menginstall package yang terlah dibangun sebelumnya, sedangkan pada implemnetaasi Native code tidak perlu karena kode yang ditulis berada pada server. Untuk total waktu *load page* menjadi lebih banyak pada implementasi Native terjadi karena membutuhkan saat inisiasi atau pengecekan apakah sudah tersambung dengan server membutuhkan waktu tambahan untuk melakukan *request* dan menunggu *response* dari server.

D. *Pengujian Penggunaan CPU*

Pengujian ini akan mengambil total resource yang digunakan oleh browser dan server (untuk implementasi Native) saat proses editing berjalan. Untuk pengecekan resource CPU yang terpakai pada browser akan memanfaatkan *browser task manager* dan pada server hanya perlu menggunakan *command* '*top*' pada terminal.



Gbr. 11 Hasil *benchmark resource* CPU client WebAssembly dan Native.

Dari Gbr 11 dapat dilihat resource yang digunakan pada bagian client untuk kedua implementasi mendapatkan hasil berupa impelmentasi WebAssembly membutuhkan resource yang lebih banyak dari pada implementasi Native code. Dari gambar tersebut didapatkan nilai untuk WebAssembly 50-101% *resource* CPU, proses paling tinggi berada pada saat melakukan pemrosesan *sequence* sedangkan saat melakukan proses *non-sequence* hanya menggunakan 42%-60% resource CPU. Berbeda dari implementasi WebAssembly, implementasi Native code menggunakan resource cukup kecil dengan kisaran 22%-77%, bahkan dalam beberapa proses hanya menggunakan *resource client* sebesar 0.1% .

Gbr. 12 *Resource* CPU server implementasi Native code.

Dari Gbr 12 menunjukkan bahwa penggunaan resource CPU server untuk implementasi Native cukup besar terutama saat melakukan proses algoritma *sharp*, Dimana pada proses tersebut membutuhkan setidaknya 500% *resource* CPU, namun pada proses yang lain hanya membutuhkan resource CPU server sebanyak 66%-115%, dengan nilai paling rendah saat aplikasi tidak melakukan proses apapun (*idle*) berada pada nilai 2.3%.

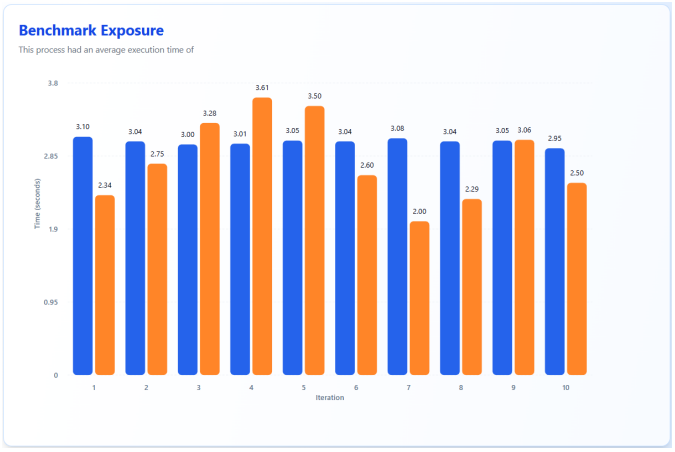
E. *Pengujian menggunakan gambar FHD dan 4K*

Pengujian ini sama seperti pengujian pertama yaitu *time execution*, namun dari pengujian tersebut membatasi ukuran gambar yang digunakan dengan resolusi HD namun pada pengujian ini akan menggunakan resolusi gambar FHD (1920px × 1080px) dan 4K (3840px × 2160px).

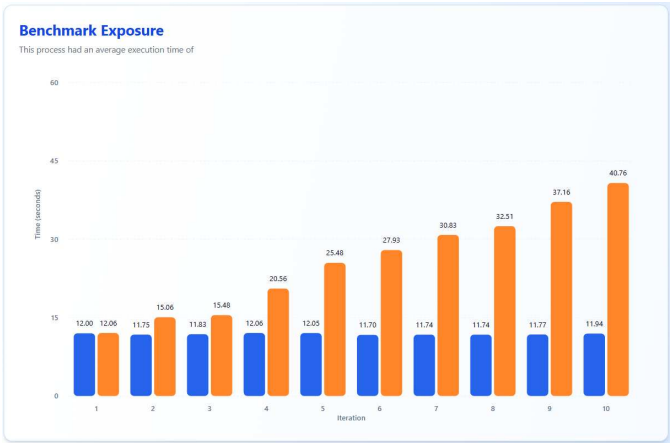
Gbr. 13 Hasil *Time Execution* Transfer color dengan gambar resolusi FHD.

Gbr. 14 Hasil *Time Execution* Transfer color dengan gambar resolusi 4K.

Dari Gbr 13 dan Gbr 14 menunjukkan hasil dari waktu pemrosesan dari algoritma *transfer color* dengan gambar resolusi FHD dan 4K, menunjukkan hasil bahwa implementasi WebAssembly membutuhkan waktu lebih lama dalam melakukan pemrosesan, hal ini dapat terjadi karena pemrosesan dari WebAssembly berada pada bagian client sehingga tugas yang dijalankan akan lebih berat dan lebih lama dibanding implementasi Native yang memiliki resource lebih banyak dan dapat melakukan pemrosesan *multi threads*. Selain itu implementasi WebAssembly juga memiliki lingkungan yang berbeda dimana pada Native code akan langsung melakukan proses pada perangkat atau CPU, sedangkan pada implementasi WebAssembly akan membangun sandbox atau lingkungan runtime pada browser, dimana hal tersebut akan menimbulkan Overhead pada saat validasi antara kedua lingkungan runtime WebAssembly dan browser[10][9], sehingga waktu pemrosesan akan lebih banyak ditambah lagi pemrosesan *transfer color* merupakan pemrosesan dua gambar sekaligus.



Gbr. 15 Hasil *Time Execution* Exposure dengan gambar resolusi FHD.



Gbr. 16 Hasil *Time Execution* Exposure dengan gambar resolusi 4K.

Berbeda dari pemrosesan *transfer color* yang merupakan algoritma *non-sequence*, pemrosesan *exposure* yang termasuk algoritma *sequence* ditunjukkan pada Gbr 15 dan Gbr 16 dimana nilai yang didapatkan cukup tinggi pada implementasi Native, sedangkan implementasi WebAssembly memberikan hasil yang lebih konsisten dengan lonjakan waktu pemrosesan yang tidak terlalu berbeda. Hal ini terjadi dikarenakan pada implementasi Native terjadi *stacking task* yang mana server diberi tugas tambahan ketika tugas yang sebelumnya belum benar-benar diselesaikan dan pada bagian UI pada client akan terjadi seperti aplikasi tidak berjalan namun proses pada server tetap berjalan dengan semestinya, pada implementasi WebAssembly juga terdapat kendala karena implementasi ini berjalan pada client sehingga ketika mendapatkan tugas yang besar aplikasi akan mengalami *blocking UI* sehingga akan mengganggu pengalaman pengguna.

F. *Analisa dan Rekomendasi*

Dari hasil pengujian yang telah dipaparkan sebelumnya diketahui bahwa implementasi WebAssembly memiliki keunggulan pada bagian *time execution* terutama pada kelompok algoritma *sequence*, dibanding dengan implementasi Native yang memiliki waktu tambahan untuk melakukan *request* dan menunggu *response* server. Namun

dari kelebihan tersebut terdapat kekurangan pada bagian pengalaman pengguna ketika tugas yang diberikan cukup banyak, hal ini akan menyebabkan *blocking UI threads* Ketika proses yang berat dilakukan atau ukuran gambar sangat besar, sedangakan pada implementasi Natve tidak terjadi *blocking UI threads* dikarenakan pemrosesan dilakukan pada server secara menyeluruh, namun ketika pemrosesan cukup lama pada halaman pengguna tidak akan mengalami perubahan pada gambar untuk beberapa saat.

Alasan mengapa *blocking UI threads* bisa terjadi pada implementasi WebAssembly dikarenakan pada saat pemrosesan WebAssembly membangun suatu *sandbox* yang bertujuan untuk lingkungan pemrosesan, yang menyebabkan *overhead* karena validasi berulang antara browser dan lingkungan *runtime*. Alasan lain yaitu pemanggilan fungsi JavaScript yang cukup banyak menyebabkan *blocking UI threads*, karena pemanggilan proses kecil secara berkala membutuhkan resource yang lebih banyak daripada satu proses besar pada satu waktu.

Dari kedua implementasi tersebut dapat diambil Kesimpulan impelmentasi WebAssembly lebih cocok untuk aplikasi yang melakukan proses yang membutuhkan performa yang cukup besar namun tidak berkala, ataru pemrosesan kecil namun berkala dimana terbukti memiliki cakupan waktu yang rendah dan stabil pada setiap pemrosesan, contoh yang cocok diantaranya aplikasi editor gambar *real-time* atau aplikasi pemrosesan audio. Sedangkan implementasi Native cocok untuk diterapkan pada aplikasi yang membutuhkan pemrosesan pada server, pemrosesan yang besar, contohnya aplikasi editor gambar atau video secara *batch* (jumlah yang banyak).

IV. KESIMPULAN

Dari hasil pengujian yang dilakukan dapat diambil kesimpulan bahwa implementasi WebAssembly memiliki kelebihan pada pemrosesan algoritma *sequence* yang hanya memproses gambar satu persatu dengan waktu pemrosesan sebesar 0.16s-1.1s tiap pemrosesan algoritma, sedangkan implementasi Native dapat melakukan pemrosesan *transfer color* lebih cepat karena resource yang digunakan bisa lebih banyak dibanding dengan device client. Namun implementasi Native sangat dipengaruhi oleh kecepatan internet, karena harus ada komuniskasi antara device client dengan server yang bekerja, sedangkan pada implementasi WebAssembly ketika halaman sudah diinisiasi maka seluruh fungsi juga sudah dapat digunakan tanpa khawatir saat pengeditan dilakukan terjadi permasalahan internet.

Pada pengujian *load page* diketahui bahwa implementasi WebAssembly memiliki waktu yang lebih singkat disbanding implementasi Native yang harus menunggu *response* server terlebih dahulu. Pada pengujian penggunaan CPU dan pengujian menggunakan resolusi FHD dan 4K ditemukan bahwa ketika beban kerja pada pengeditan cukup tinggi aplikasi editor dengan implemnetasi WebAssembly bisa menimbulkan *blocking UI* dan waktu pemrosesan untuk *transfer color* bisa lebih lama, namun untuk algoritma *sequence* bisa lebih singkat walau terdapat *blocking UI* dan pada implementasi Native ketika beban kerja cukup tinggi akan menyebabkan *delay result* saat pengeditan, yang menyebabkan aplikasi editor tidak melakukan apa-apa padahal masih terdapat tugas yang menumpuk pada server.

REFERENSI

[1] ALEVÄRN, M. (2021, June 11). Server-side image processing in native code compared to clientside image processing in WebAssembly. KTH ROYAL INSTITUTE OF TECHNOLOGY SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE. Retrieved May 20, 2025, from https://www.diva-portal.org/smash/get/diva2:1587964/FULLTEXT01.pdf

[2] Web Framework benchmark. (n.d.). TechEmpower. Retrieved June 14, 2025, from https://www.techempower.com/benchmarks/#section=data-r21

[3] Golsch, L. (2019, November 29). WebAssembly: Basics.

[4] Rossberg, A. (Ed.). (2025). *WebAssembly Specification Release 2.0 (Draft 2025-05-15)* (2nd ed.). https://webassembly.org/.

[5] Gurgone, G., & Spiess, P. (2024, August 21). A real-world WebAssembly benchmark. Nutrient. Retrieved June 16, 2025, from https://www.nutrient.io/blog/a-real-world-webassembly-benchmark/

[6] *wasm-pack documentation.* (n.d.). Retrieved Oktober 29, 2024 from https://rustwasm.github.io/wasm-pack/

[7] *Actix web Documentation.* (n.d.). Retrieved June 1, 2025 from Actix Web: https://actix.rs/docs

[8] *What are browser developer tools? - Learn web development | MDN.* (2025, April 29). Retrieved June 3, 2025 from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Learn_web_development/Howto/Tools_and_setup/What_are_browser_developer_tools

[9] Pérez, C. A. (2019, September). Measuring Opencv.js performance with Wasm execution engine in desktop, embedded and mobile browsers. 51-57.

[10] *WebAssembly concepts - WebAssembly | MDN.* (2025, October 14). From MDN: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts