Web Assembly Part - I Report

COP 701: Software Systems Lab

(MCS192568 and MCS192574)

Introduction

WebAssembly [1] (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Deliverables - Part 1

WebAssembly vs. Native Code

Experiment 1 - Relative Execution Time:

Run the PolyBench suite:

We have run all 30 programs of polyBench which are mentioned in benchmark_list file in Polybench.

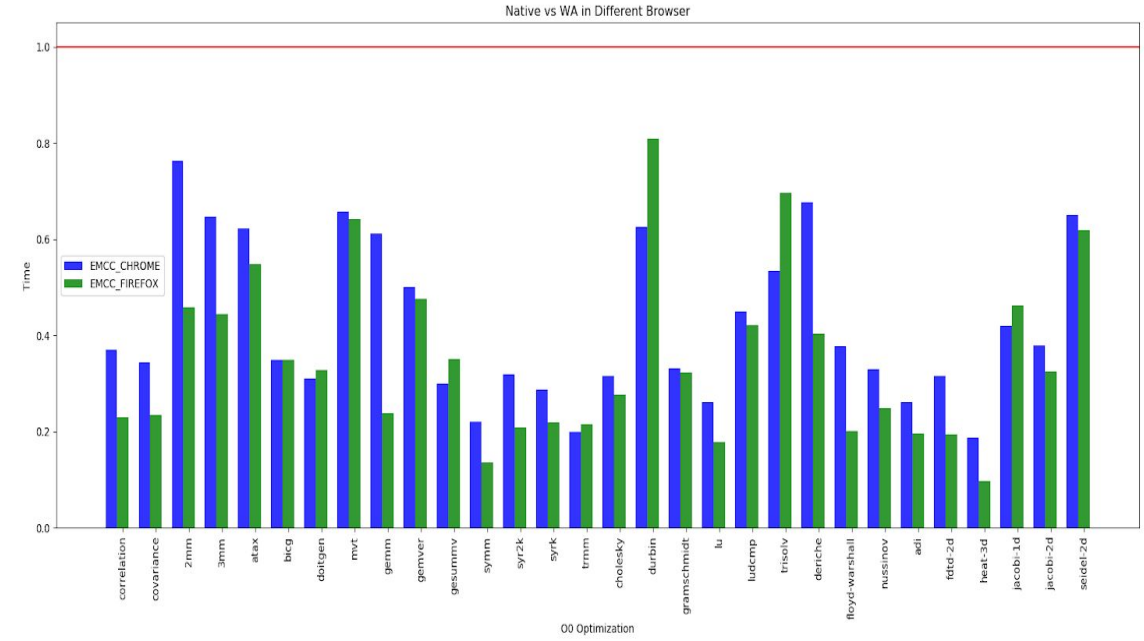
For native code we have used python scripts using **GCC** compiler

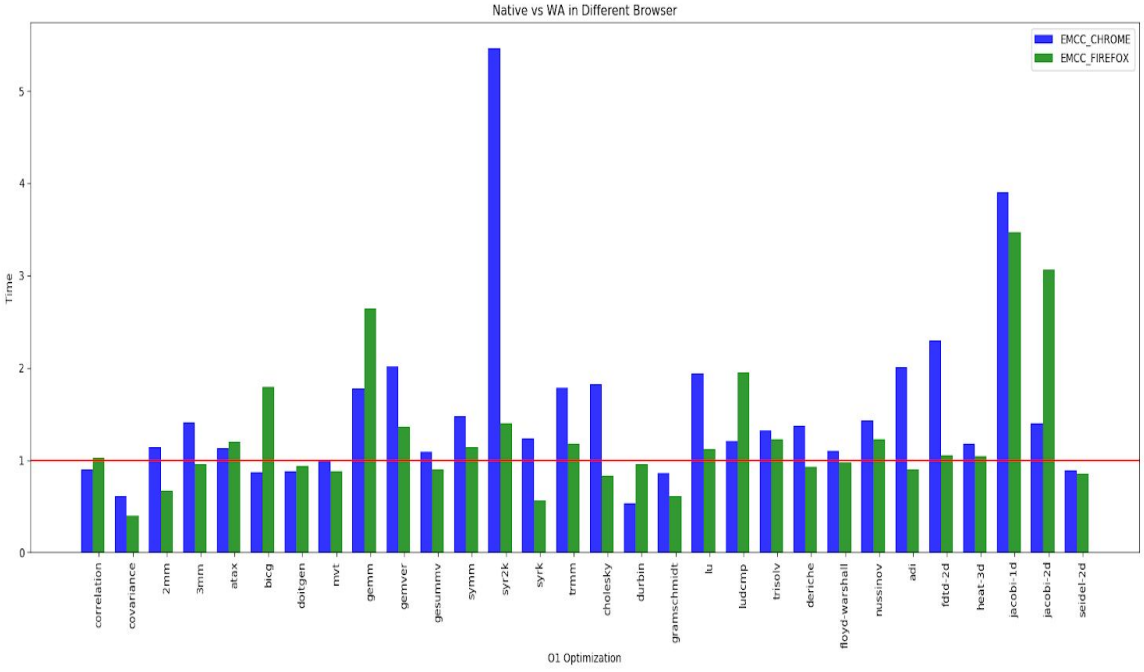
For Web assembly code we have used python scripts using **Emscriptem** to generate the required html file to run on web.

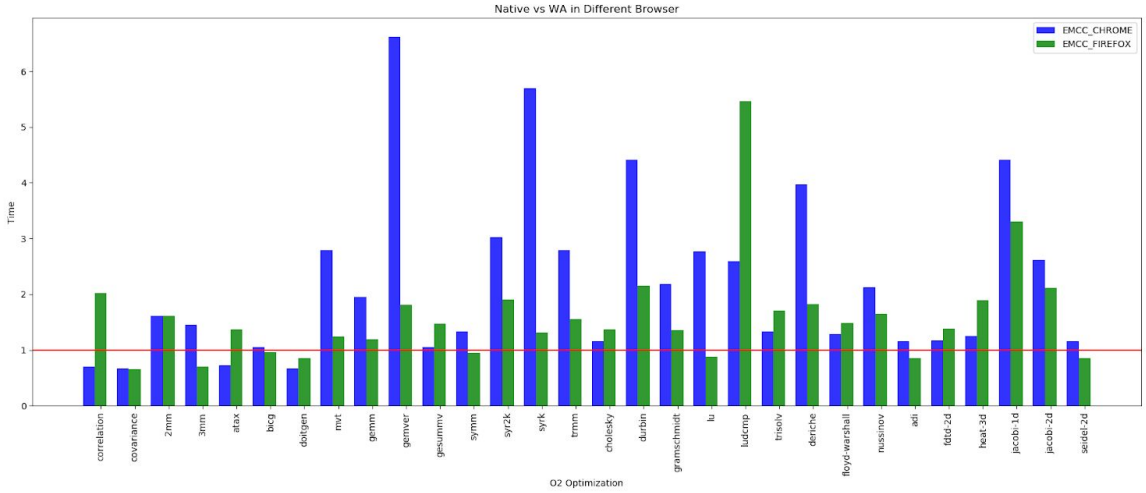
We ran all files on **Chrome** and **FireFox** browser and noted the running time.

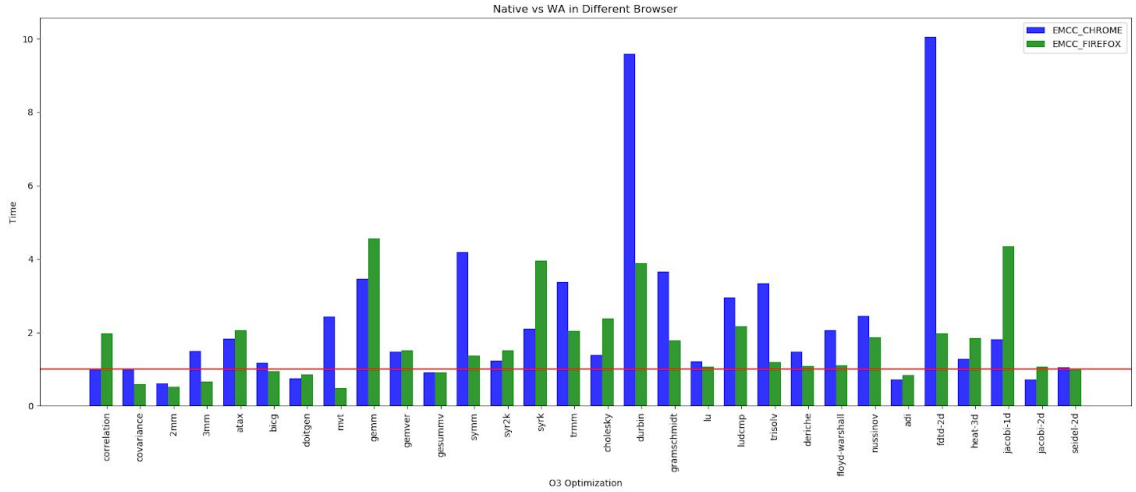
We stored the result in **PICKLE** format and used it in **plot1.py** to plot the graph using matplotlib. Pickles used :: **emcc.pickle , gcc.pickle**

Plot1.py is the python script to plot the graph using matplotlib









<u>Conclusion</u>:

We can observe from the following graph the drastic improvement in GCC optimization from O0→ O3

It can be seen that as we optimize EMCC and GCC there is not much variance in run time of EMCC where as GCC shows a sharp drop in running time.

TABLE DATA : NATIVE VS WA_CHROME VS VS WA_FIREFOX

FileName	Native	Chrome	Firefox
correlation	1.036447	0.385	0.238
covariance	1.023256	0.353	0.24
2mm	0.217544	0.166	0.1
3mm	0.294644	0.191	0.131
atax	0.067357	0.042	0.037
bicg	0.094306	0.033	0.033
doitgen	9.053019	2.822	2.975
mvt	0.115363	0.076	0.074
gemm	0.142328	0.087	0.034
gemver	0.203771	0.102	0.097
gesummv	0.039934	0.012	0.014
symm	0.23491	0.052	0.032
syr2k	0.45239	0.145	0.094
syrk	0.128161	0.037	0.028
trmm	0.120602	0.024	0.026
cholesky	0.180052	0.057	0.05
durbin	0.038307	0.024	0.031
gramschmidt	0.204757	0.068	0.066
lu	0.629148	0.164	0.112
ludcmp	0.311057	0.14	0.131
trisolv	0.018677	0.01	0.013
deriche	1.700097	1.152	0.688
floyd-warshall	0.435825	0.165	0.088
nussinov	2.19042	0.722	0.544
adi	0.890928	0.233	0.175
fdtd-2d	0.421735	0.133	0.082
heat-3d	1.00217	0.188	0.098
jacobi-1d	0.023774	0.01	0.011
jacobi-2d	0.24264	0.092	0.079
seidel-2d	0.494345	0.322	0.306

FileName	Native	Chrome	Firefox
correlation	0.314698	0.284	0.321
covariance	0.43711	0.269	0.175
2mm	0.101363	0.116	0.068
3mm	0.125874	0.177	0.12
atax	0.031783	0.036	0.038
bicg	0.02172	0.019	0.039
doitgen	3.046613	2.687	2.848
mvt	0.049865	0.049	0.044
gemm	0.033279	0.059	0.088
gemver	0.066835	0.135	0.091
gesummv	0.010057	0.011	0.009
symm	0.0298	0.044	0.034
syr2k	0.060021	0.328	0.084
syrk	0.026625	0.033	0.015
trmm	0.026329	0.047	0.031
cholesky	0.065887	0.12	0.055
durbin	0.033482	0.018	0.032
gramschmidt	0.08672	0.075	0.053
lu	0.113472	0.22	0.127
ludcmp	0.05554	0.067	0.108
trisolv	0.010583	0.014	0.013
deriche	0.691137	0.951	0.639
floyd-warshall	0.122582	0.135	0.12
nussinov	0.325597	0.465	0.401
adi	0.200365	0.402	0.181
fdtd-2d	0.074208	0.17	0.078
heat-3d	0.09666	0.114	0.101
jacobi-1d	0.002308	0.009	0.008
jacobi-2d	0.033601	0.047	0.103
seidel-2d	0.300143	0.267	0.256

FileName	Native	Chrome	Firefox
correlation	0.231351	0.163	0.467
covariance	0.22522	0.151	0.148
2mm	0.045797	0.074	0.074
3mm	0.081769	0.119	0.058
atax	0.037178	0.027	0.051
bicg	0.020807	0.022	0.02
doitgen	1.689575	1.128	1.44
mvt	0.038663	0.108	0.048
gemm	0.025122	0.049	0.03
gemver	0.052417	0.347	0.095
gesummv	0.009474	0.01	0.014
symm	0.04421	0.059	0.042
syr2k	0.066058	0.2	0.126
syrk	0.015268	0.087	0.02
trmm	0.022573	0.063	0.035
cholesky	0.032887	0.038	0.045
durbin	0.009279	0.041	0.02
gramschmidt	0.042037	0.092	0.057
lu	0.082995	0.23	0.073
ludcmp	0.044794	0.116	0.245
trisolv	0.005267	0.007	0.009
deriche	0.633365	2.513	1.154
floyd-warshall	0.10943	0.141	0.163
nussinov	0.308352	0.655	0.51
adi	0.188833	0.218	0.162
fdtd-2d	0.063153	0.074	0.087
heat-3d	0.091511	0.115	0.173
jacobi-1d	0.004531	0.02	0.015
jacobi-2d	0.032148	0.084	0.068
seidel-2d	0.297752	0.347	0.256

FileName	Native	Chrome	Firefox
correlation	0.162554	0.161	0.32
covariance	0.204879	0.201	0.121
2mm	0.079362	0.049	0.041
3mm	0.086416	0.129	0.058
atax	0.016999	0.031	0.035
bicg	0.021131	0.025	0.02
doitgen	1.794551	1.344	1.542
mvt	0.080292	0.196	0.039
gemm	0.011829	0.041	0.054
gemver	0.054802	0.081	0.083
gesummv	0.009787	0.009	0.009
symm	0.034428	0.144	0.047
syr2k	0.10422	0.128	0.158
syrk	0.016662	0.035	0.066
trmm	0.013661	0.046	0.028
cholesky	0.025218	0.035	0.06
durbin	0.007197	0.069	0.028
gramschmidt	0.038742	0.142	0.069
lu	0.091777	0.112	0.099
ludcmp	0.055867	0.165	0.121
trisolv	0.00748	0.025	0.009
deriche	0.586128	0.867	0.637
floyd-warshall	0.108747	0.225	0.12
nussinov	0.289256	0.709	0.54
adi	0.195417	0.139	0.163
fdtd-2d	0.033311	0.335	0.066
heat-3d	0.081371	0.104	0.15
jacobi-1d	0.002757	0.005	0.012
jacobi-2d	0.061438	0.044	0.066
seidel-2d	0.251114	0.263	0.255

Experiment 2 - Instruction Mix Analysis:

Pin Tool:

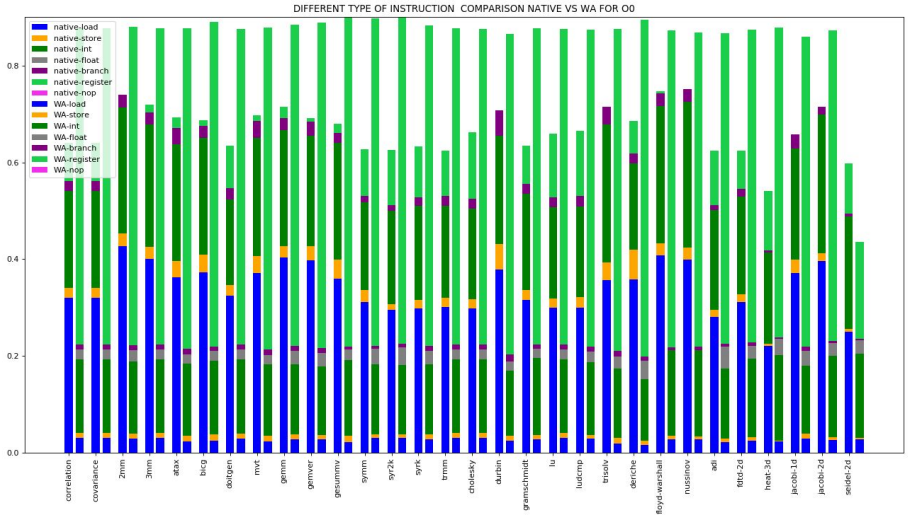
We used the pin tool ver-3.11 to benchmark the native code.
For Instruction breakup we modified the **inscount0.cpp**
Pin2_3.py is the name of python script used to run the benchmark
The namespaces mentioned in PIN manual are used to count different breakups of instruction are present in inscount file.
The result observed is stored as PICKLE format to be used in **plot2.py** python script which generates bar graph.

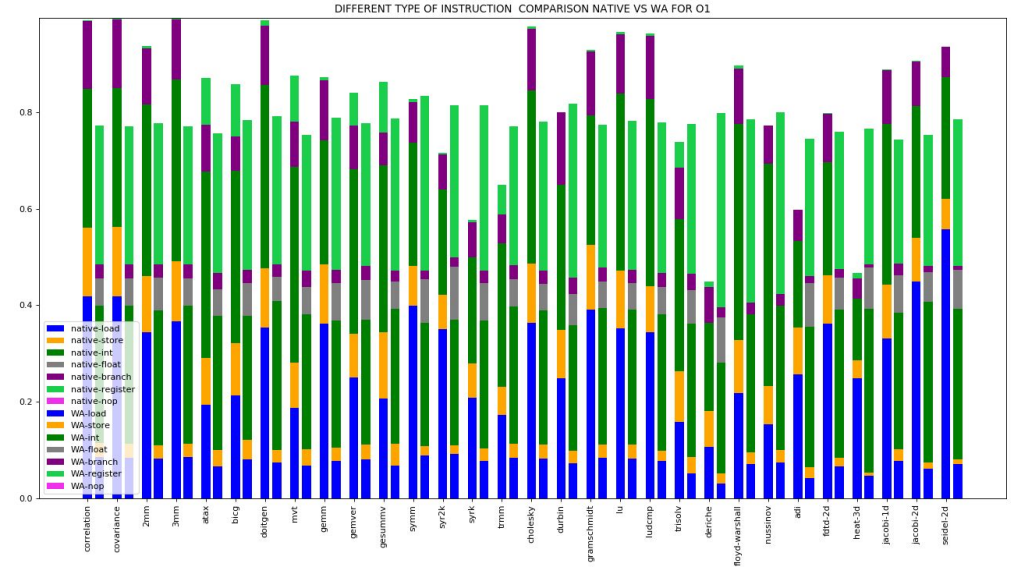
Wasabi tool

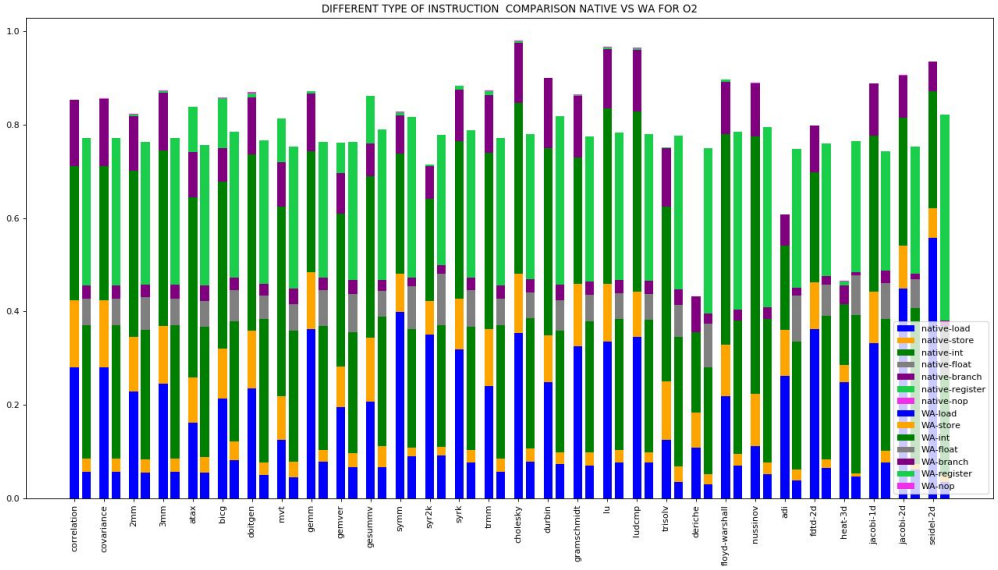
[5] to instrument the WebAssembly code to obtain the instruction breakup. Types: load, store, arithmetic/logical-int, arithmetic/logical- oat, branch, register transfer, And Nop

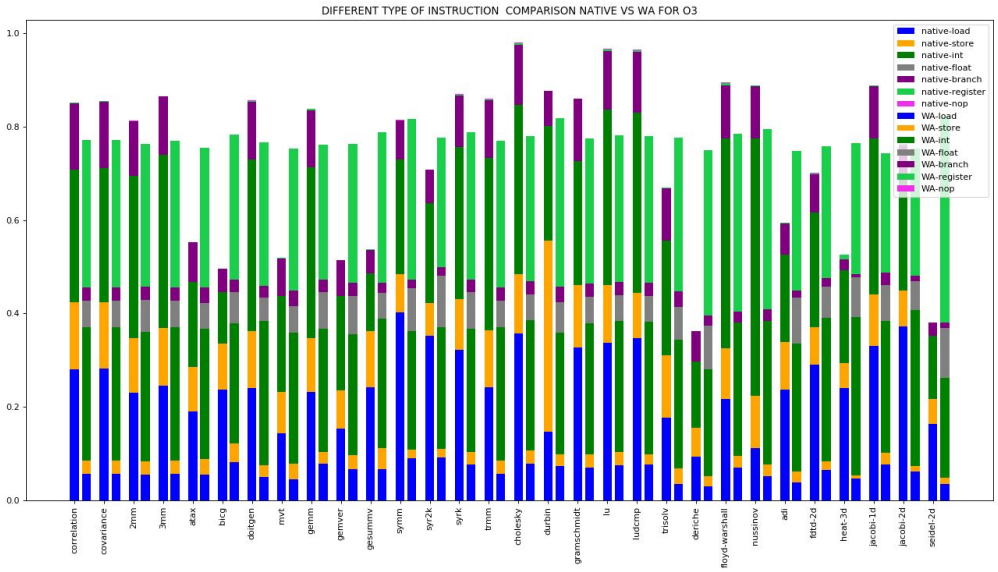
We use all the hooks to get total instruction count as well as different types of instructions.
We have made a thread script which parallely use the emcc for different browsers and gcc for native machine in question 1 and parallely spawns emrun thread for question 2.

We modified the js file **instruction-mix.js** to find total instruction count, breakup instruction count and frequency of function executions. To prevent manual intervention we have used a POST call to the emrun server to post the output on server console and use subprocess.Popen of python to get the data.
Plot2.py is the python script to plot the graph using matplotlib


DIFFERENT TYPE OF INSTRUCTION COMPARISON NATIVE VS WA FOR O0
native-load
native-store
native-int
native-float
native-branch
native-register
native-nop
WA-load
WA-store
WA-int
WA-float
WA-branch
WA-register
WA-nop
0.0
0.2
0.4
0.6
0.8
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d


DIFFERENT TYPE OF INSTRUCTION COMPARISON NATIVE VS WA FOR O1
native-load
native-store
native-int
native-float
native-branch
native-register
native-nop
WA-load
WA-store
WA-int
WA-float
WA-branch
WA-register
WA-nop
0.0
0.2
0.4
0.6
0.8
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d





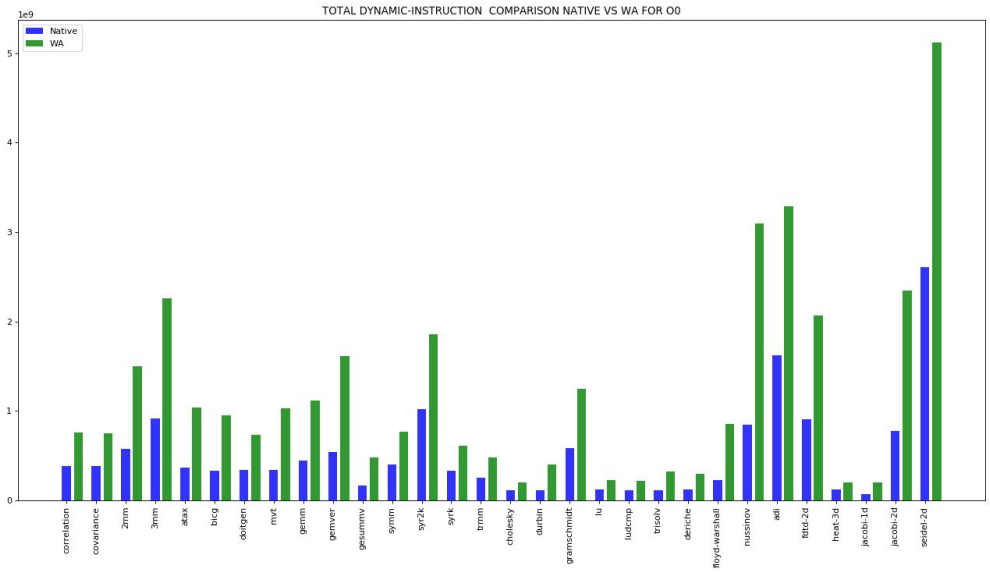
Conclusion:

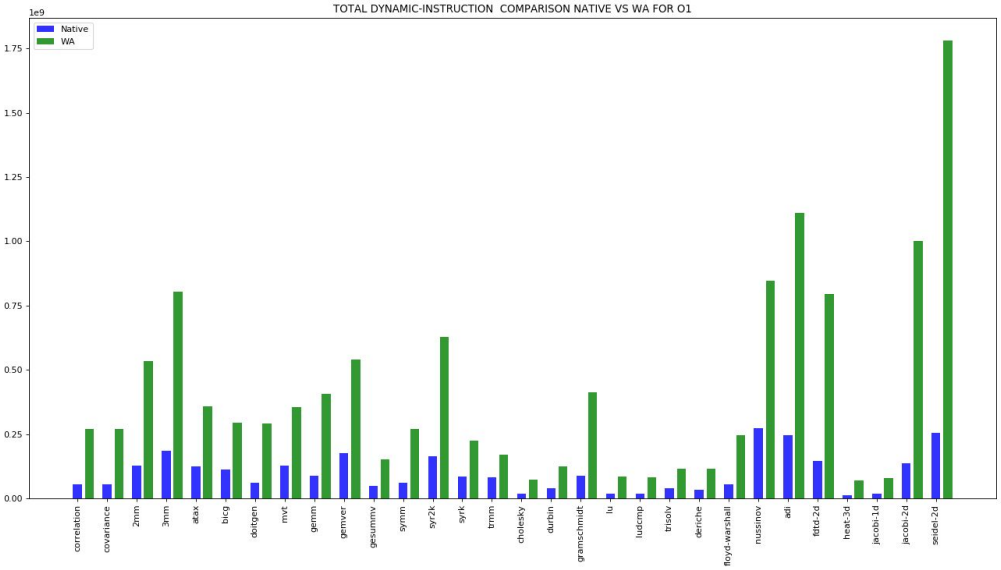
Improvements can be observed in both NATIVE and WEBASSEMBLY calculated using PIN and WASABI tool for instrumentation.
In native machine it is observed that as we increase the optimization, more instructions are incorporated into the above mentioned type.

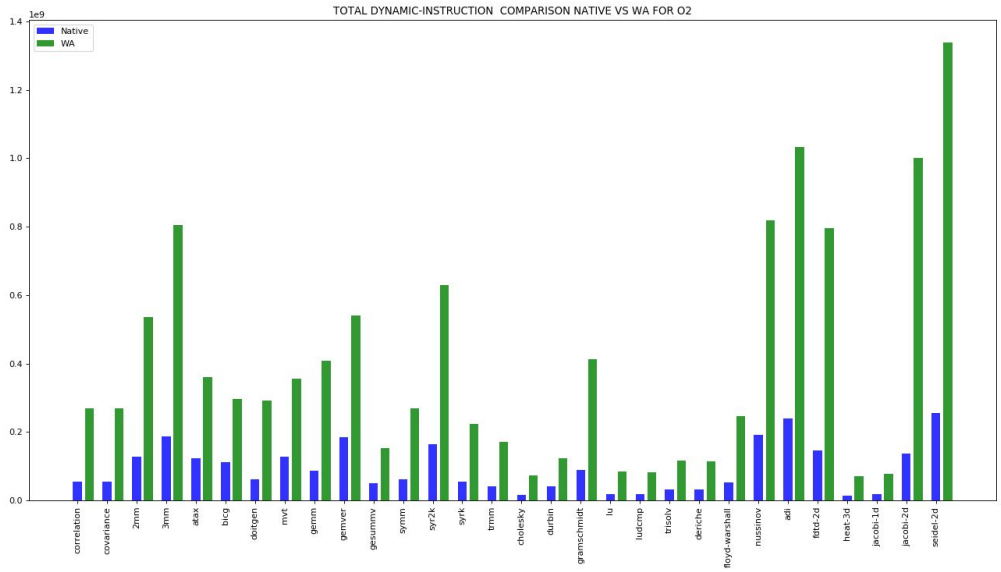
The convergence of type of instruction into our used format is comparatively slow in WEBASSEMBLY.

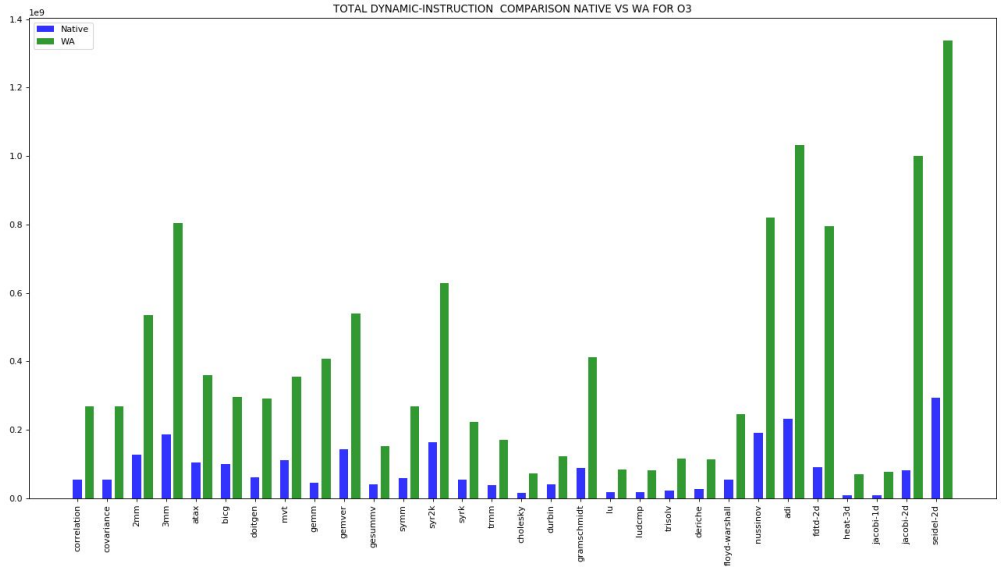
Experiment 3 - Instruction Count:
For native code we again used PIN tool to count no of dynamic instruction count using the following namespace:
Pin2_3.py is the name of python script used to run the benchmark
Plot3.py python script is used to plot the graph showing relative
Plot a bar graph showing the
relative number of dynamic instructions for WebAssembly vis-a-vis Native Code.

Plot3.py is the python script to plot the graph using matplotlib




TOTAL DYNAMIC-INSTRUCTION COMPARISON NATIVE VS WA FOR O1
1e9
Native
WA
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d


TOTAL DYNAMIC-INSTRUCTION COMPARISON NATIVE VS WA FOR O2
1e9
Native
WA
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d



Conclusion

As we Optimise our GCC there is significant drop in total number of Dynamic instructions which supports the observations of experiment 1 that total time reduces in majority cases.
In output of WebAssembly instrumented by wasabi is not that impressive as native machine.

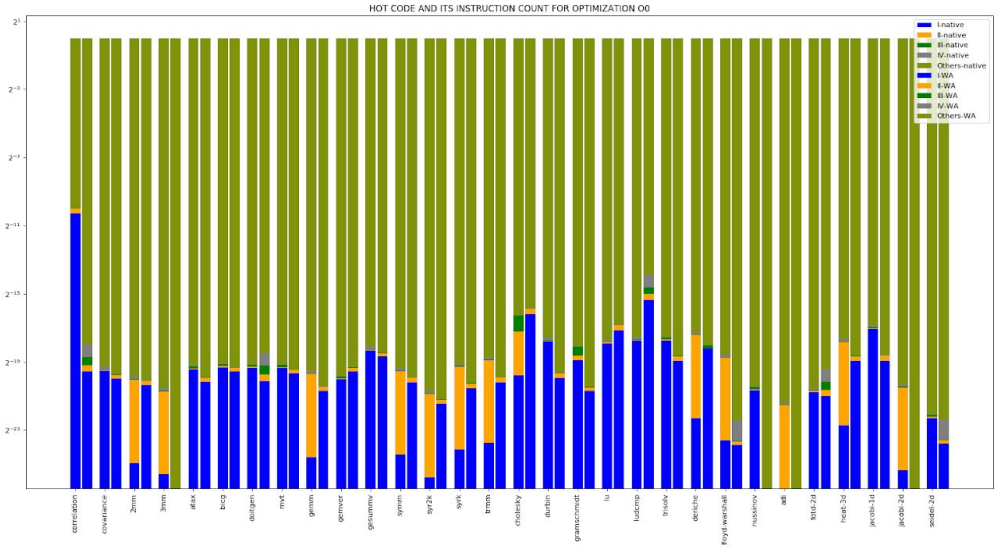
• **Experiment 4 - Hot Code:** Identify the functions in the code that are run frequently and plot their dynamic instruction count.
PIN: Tool is used to count the Hot code by sorting the functions in descending order according to number of calls per function.
The name and ratio of count of INstruction of each function is sent as output.
The result is stored in pickle format as **nativePin4.pickle**

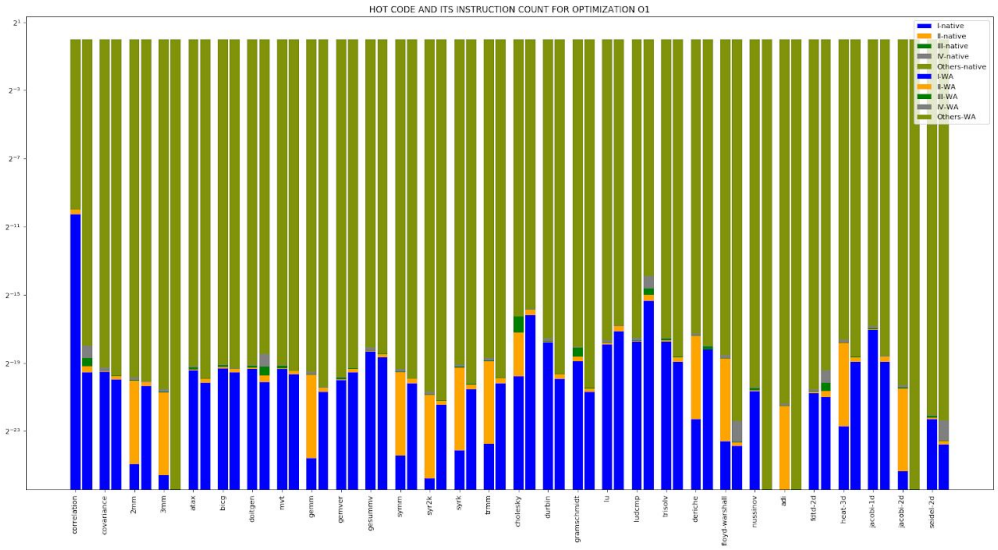
Proccount.cpp file of PIN tool is used for this purpose.
A Vector< Pair <FunName,InsCount>, CallCount > is used to calculate the required instruction.

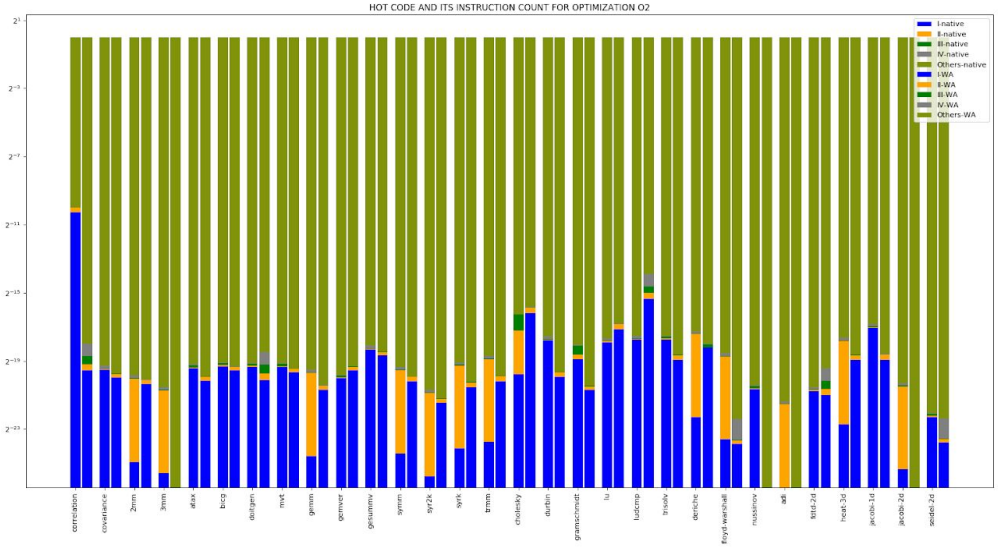
WASABI:

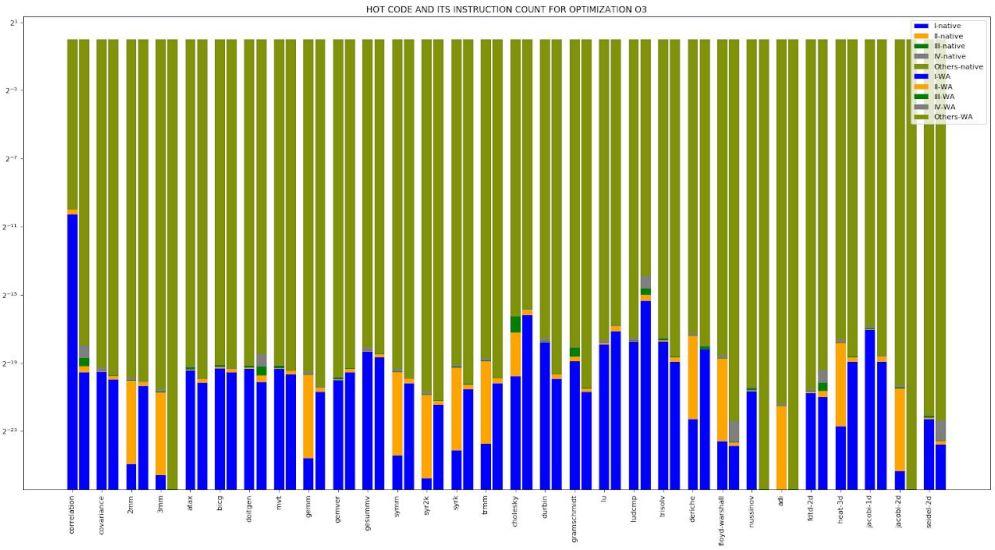
In the hot code section we have used stack implementation to keep track of all the dynamic instructions inside the function. Additionally we made sure that if a function A gets called inside function B then instruction count of function A should not include instruction count of function B. Additionally we used wasabi.wast file which is generated by using -g3 flag in emcc command to map function name to its function number.

The call_pre hook was used to detect the start of a function whereas call_post was used to detect the end of a function. So we pushed a function on stack at call_pre hook and popped at call_post hook and in between all the instructions were used to increment instruction count of the function on top of the stack.




HOT CODE AND ITS INSTRUCTION COUNT FOR OPTIMIZATION O1
I-native
II-native
III-native
IV-native
Others-native
I-WA
II-WA
III-WA
IV-WA
Others-WA
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d


HOT CODE AND ITS INSTRUCTION COUNT FOR OPTIMIZATION O2
I-native
II-native
III-native
IV-native
Others-native
I-WA
II-WA
III-WA
IV-WA
Others-WA
correlation
covariance
2mm
3mm
atax
bicg
doitgen
mvt
gemm
gemver
gesummv
symm
syr2k
syrk
trmm
cholesky
durbin
gramschmidt
lu
ludcmp
trisolv
deriche
floyd-warshall
nussinov
adi
fdtd-2d
heat-3d
jacobi-1d
jacobi-2d
seidel-2d



Conclusion

The above bar graph plots the relative hotcode in each function.

Since we count the total Instruction of top 4 Hotcodes and rest are included in others hence relative value of “others count” to “hot codes count” is very huge sometimes tending to 99% hence we have scaled the y axis in bar graph to log.

Since we sort the functions as per their calls rather than Instruction code, hece function with large instruction count is included in others. Making it the most dominant category even though it does not contain top hot codes.

Native Hardware

DRAM Size	12 GB DDR3
CPU Frequency	Intel(R) Core(TM) i3-5010U CPU @ 2.10GHz (i3 - 5th Generation)
Cache Size	3072 KB
CPU cores	2
OS Version	Ubuntu 16.04.6 LTS (codename : Xenial) with GNOME Shell 3.18.5
GCC Version	gcc (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609
Python Version	Python 3.7.3
Google Chrome	Google Chrome 77.0.3865.120
Mozilla Firefox	Mozilla Firefox 69.0.1
PolyBench	polybench-c-4.1