Analyzing Redundant Memory Bounds Checks for Wasm2c

By: Allen Jue and David Lu

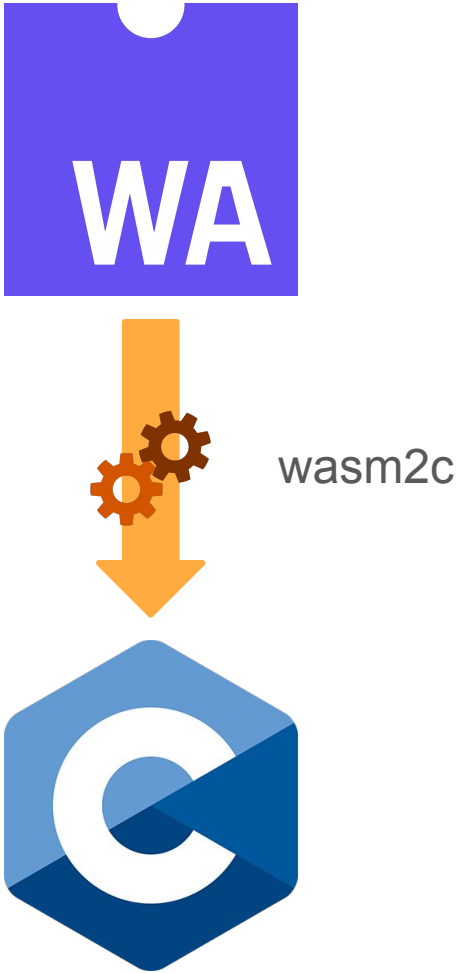
WebAssembly (wasm)

- Wasm is increasingly popular in the wild
- Wasm has security overheads[1]
- Wasm is a low-level language[1]

[1] *WebAssembly Security*. WebAssembly. (n.d.). https://webassembly.org/docs/security/

Wasm2c

- Part of the Wasm Toolkit (wabt)
- Decompiles Wasm into a C source and header
- **Why decompilation?**
  - “Is This the Same Code?”[2]
  - Help understand the code
  - Existing tools to verify, analyze, and debug C code



[2] Wu et. al. (2024).

Wasm Overheads

- Wasm has two goals[3]:
  - Security
  - Speed
- For security, guard pages and bounds checking
- Up to **650% overheads** from memory bounds checks[4]
  - Cholesky Decomposition (20%)
  - GEMM (220%)

[3] Rossberg et al., 2018

[4] Szewczyk, 2022

Wasm Memory Layout

- A single large array of bytes
  - Can be grown in size as needed by the memory.grow instruction
- 4 primitive types supported:
  - 32/64 bit floating point
  - 32/64 bit integer

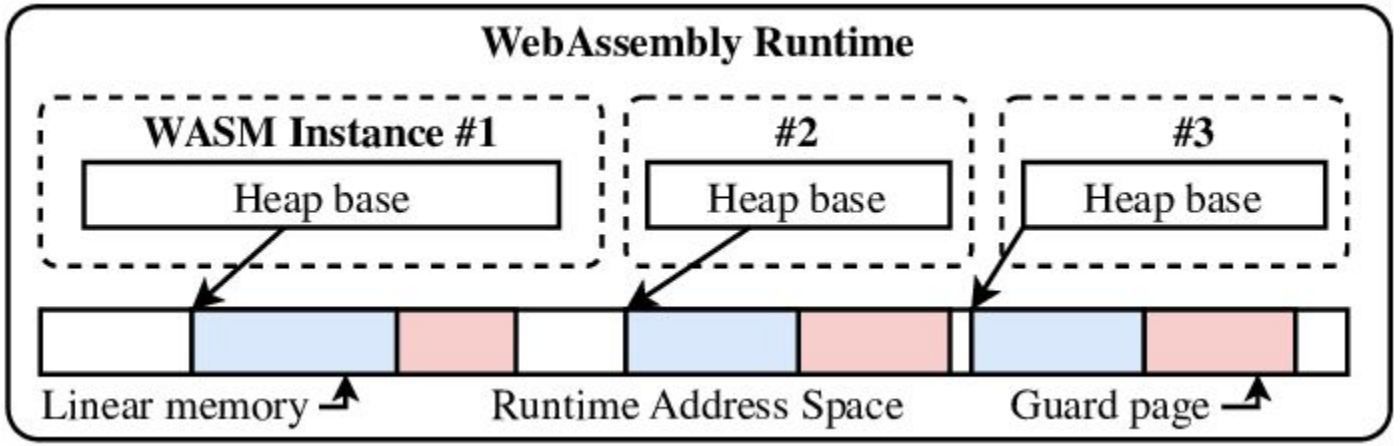


Image source: Fink et al., 2024

Wasm Sandboxing

- Sandboxing to help security
- Typically done in 2 implementations:
  - Explicit bounds checks (validate every memory access beforehand)
  - Guard Pages (mark pages outside of instance as inaccessible)

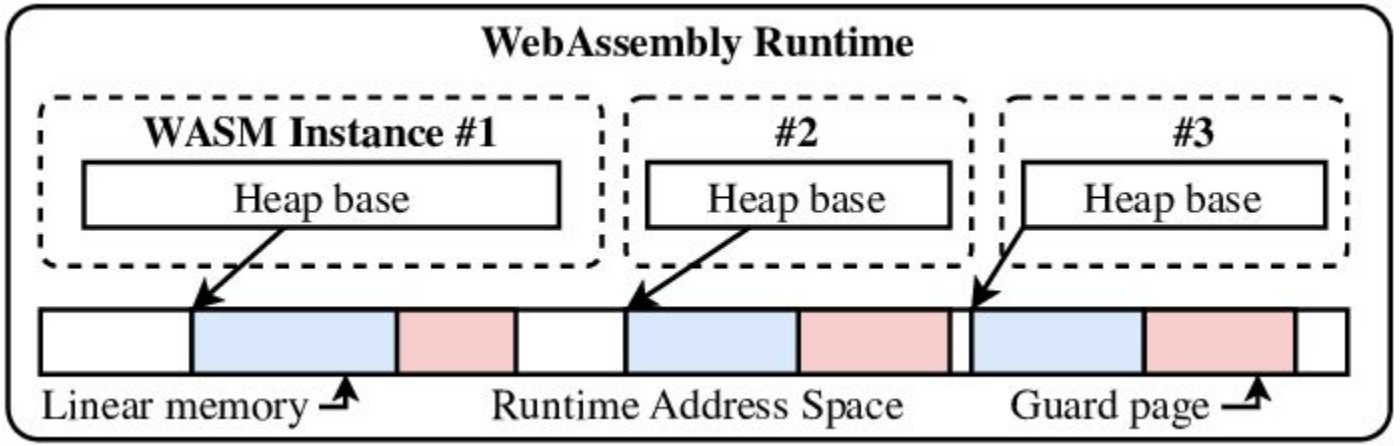


Image source: Fink et al., 2024

Guard Pages

- Used in wasm runtimes to detect out-of-bounds memory accesses
  - Are implemented as unmapped memory regions placed outside WebAssembly linear memory
- Traps if a pointer out of bounds is accessed

In 64-bit Wasm, requires a specific configuration:

- 64-bit platforms
- MMAP allocation strategy
- No 64-bit memories
- No big-endian architecture

Will fall back to bounds checks if not met

Guard Pages - Overhead

- 32 bit guard pages: Essentially 0 overhead
- Two-level guard pages: 12.7% overhead
- Single-level 64-bit guard pages: 76-142% overhead

Bounds Checks

- Used in wasm runtimes to detect out-of-bounds memory accesses
- Will cause a trap (termination) on OOB accesses
- In 64-bit environments, are usually implemented instead of guard pages
    - Incur heavy overhead compared to guard pages

Bounds Check Elimination

Dynamic type-level

- Uses runtime knowledge of types and ranges to determine safe memory accesses dynamically.
- Ex: Statistical check elimination and caching redundant checks

Static type-level

- Compiler infers information about parameter types, sizes, and access patterns.
- Reason about program without running it

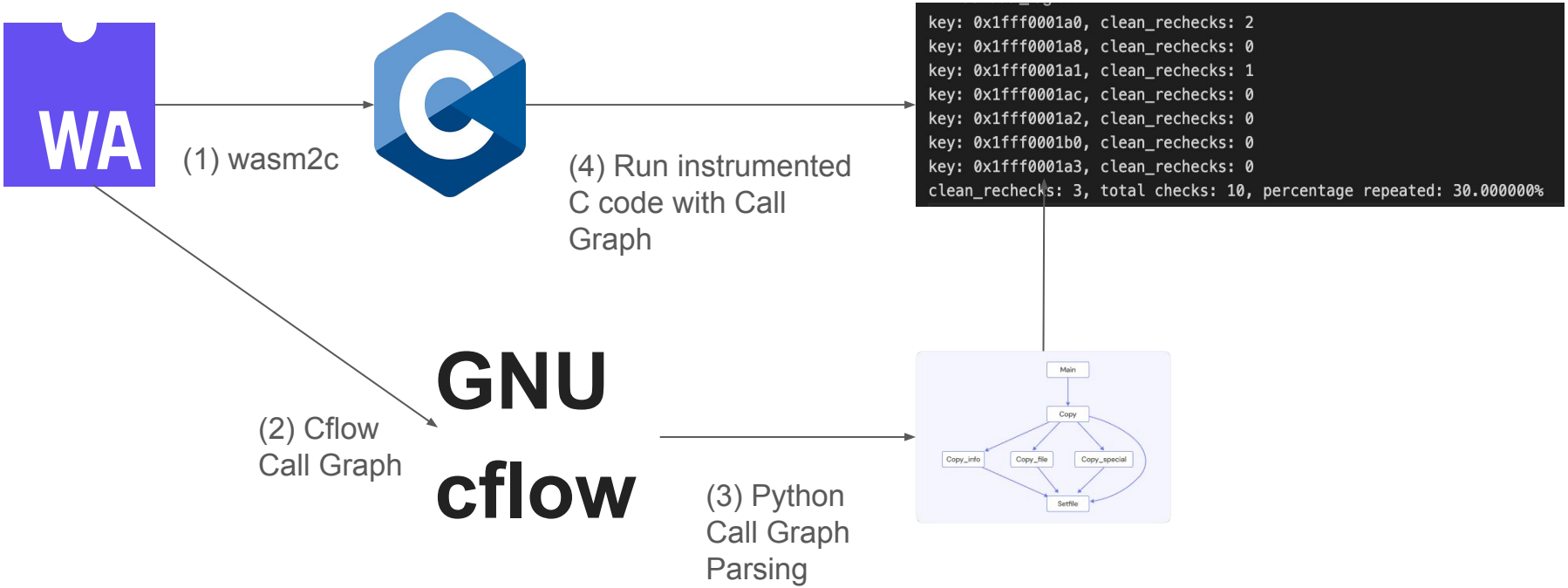
Solution

We want to create a dynamic tool that tries to identify redundant bounds checks that will likely be identified by a static checker.

Criterion

1. Look at functions 1 frame away
2. Repeat bounds checks
3. Have pointer arguments passed in

Solution


WA
(1) wasm2c
(4) Run instrumented C code with Call Graph
key: 0x1fff0001a0, clean_rechecks: 2
key: 0x1fff0001a8, clean_rechecks: 0
key: 0x1fff0001a1, clean_rechecks: 1
key: 0x1fff0001ac, clean_rechecks: 0
key: 0x1fff0001a2, clean_rechecks: 0
key: 0x1fff0001b0, clean_rechecks: 0
key: 0x1fff0001a3, clean_rechecks: 0
clean_rechecks: 3, total checks: 10, percentage repeated: 30.000000%
(2) Cflow Call Graph
GNU cflow
(3) Python Call Graph Parsing
Main
Copy
Copy_info
Copy_file
Copy_special
Setfile

CFlow Example

```
int fibonacci(int n) {
        if (n <=1) return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
}

int main (int argc, char **argv) {
        return fibonacci(5);
}
```

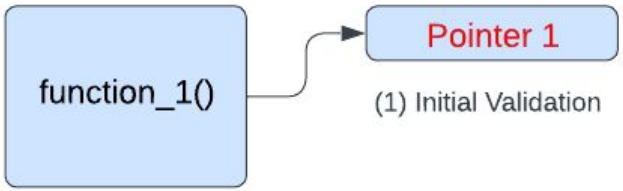
```
main() <int main (int argc,char **argv) at fib.c:26>:
        fibonacci() <int fibonacci (int n) at fib.c:10>:
                fibonacci() <int fibonacci (int n) at fib.c:12>
                fibonacci() <int fibonacci (int n) at fib.c:12>
```

Implementation

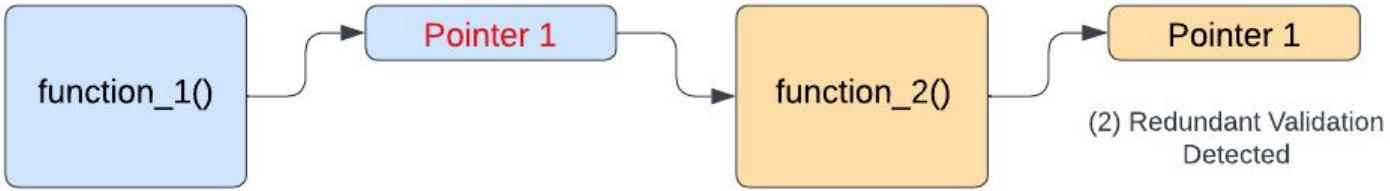
- MemoryInfo metadata object stored in a map that tracks bounds checks for all pointers
  (1) Track Caller-Callee relationships when pointers are validated
  (2) Identify repeated bounds checks
  (3) Track pointers that are passed in as parameters*
- When a pointer is checked
  1. Verified if a pointer meets all 3 criterion.
  2. Otherwise, need to revalidate pointer

Implementation

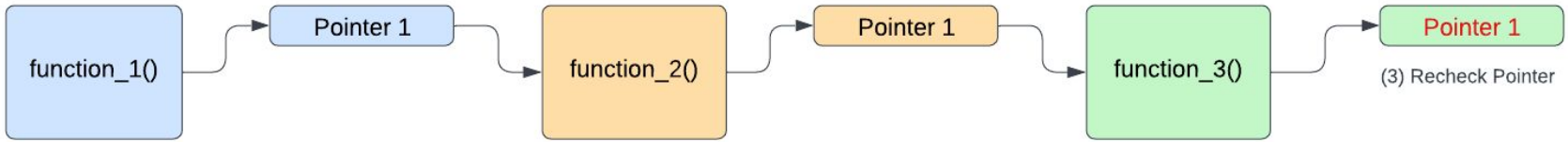
Last checked depth: [**1**]



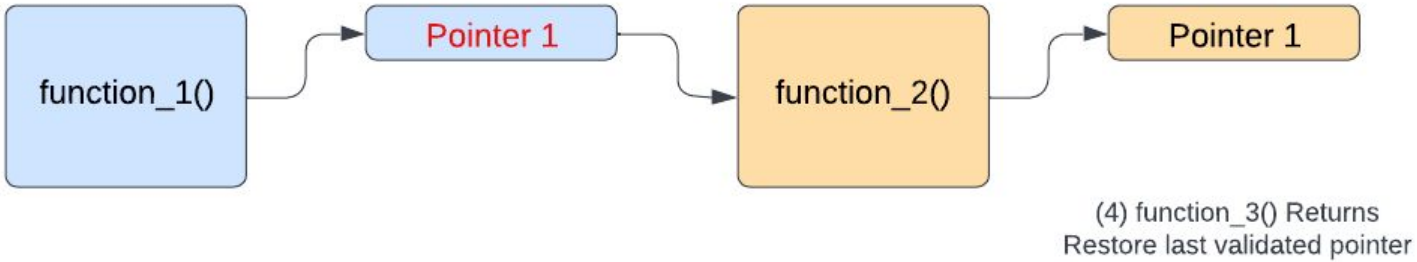
Last checked depth: [**1**]



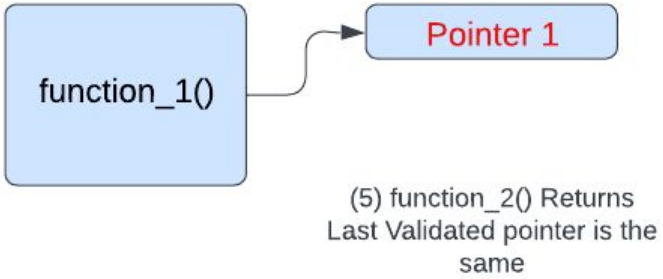
Last checked depth: [1, **3**]



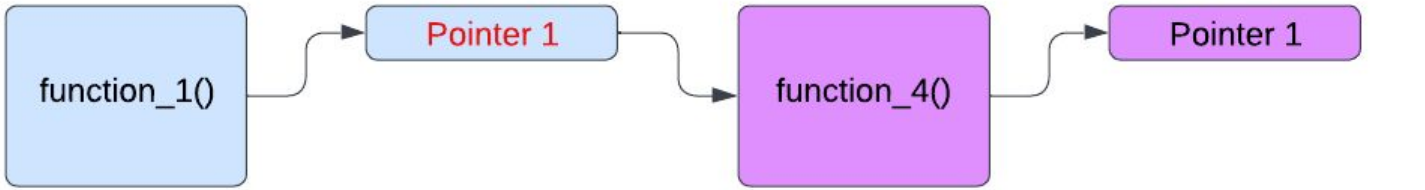
Last checked depth: [**1**]



Last checked depth: [**1**]



Last checked depth: [**1**]



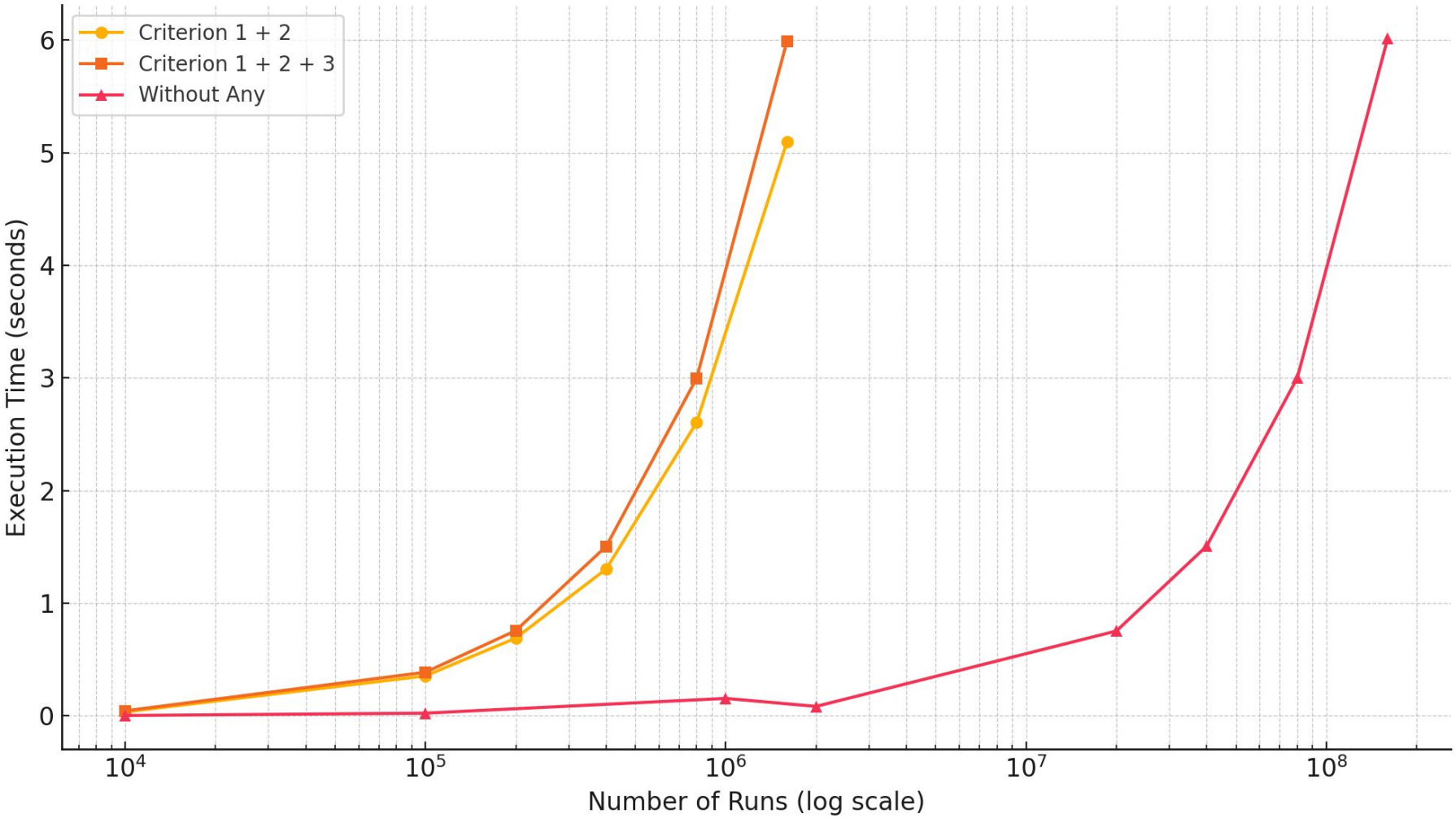
(6) Redundant check correctly detected

Dhrystone Metrics

Dhrystone is a synthetic computing benchmark that tests programs meant to mimic common processor usages.

Criterion 1+2	
Clean rechecks	552,080,105
Total memory checks	675,178,741
Percentage repeated	**81.77%**
Overhead	**5,917%**

Criterion 1+2+3	
Clean rechecks	21,770,370
Total memory checks	332,778,455
Percentage repeated	**6.54%**
Overhead	**9972%**


Execution Time vs Number of Runs
Criterion 1 + 2
Criterion 1 + 2 + 3
Without Any
Execution Time (seconds)
0
1
2
3
4
5
6
10^4
10^5
10^6
10^7
10^8
Number of Runs (log scale)

clean_rechecks: 552080105, total
checks: 675178741, percentage
repeated: 81.767993

one run through Dhrystone: 2.262293
econd: 442029.428938
= 251.581917

one run through Dhrystone: 0.037596
econd: 26598295.747448
= 15138.472252

one run through Dhrystone: 3.786496
econd: 264096.392542
= 150.310980

21770370, total checks: 332778455,
ated: 6.542001

10000 runs 0.034150 seconds
100000 runs 0.353827 seconds
200000 runs 0.689923 seconds
400000 runs 1.303460 seconds
800000 runs 2.604929 seconds
1600000 runs 5.098361 seconds

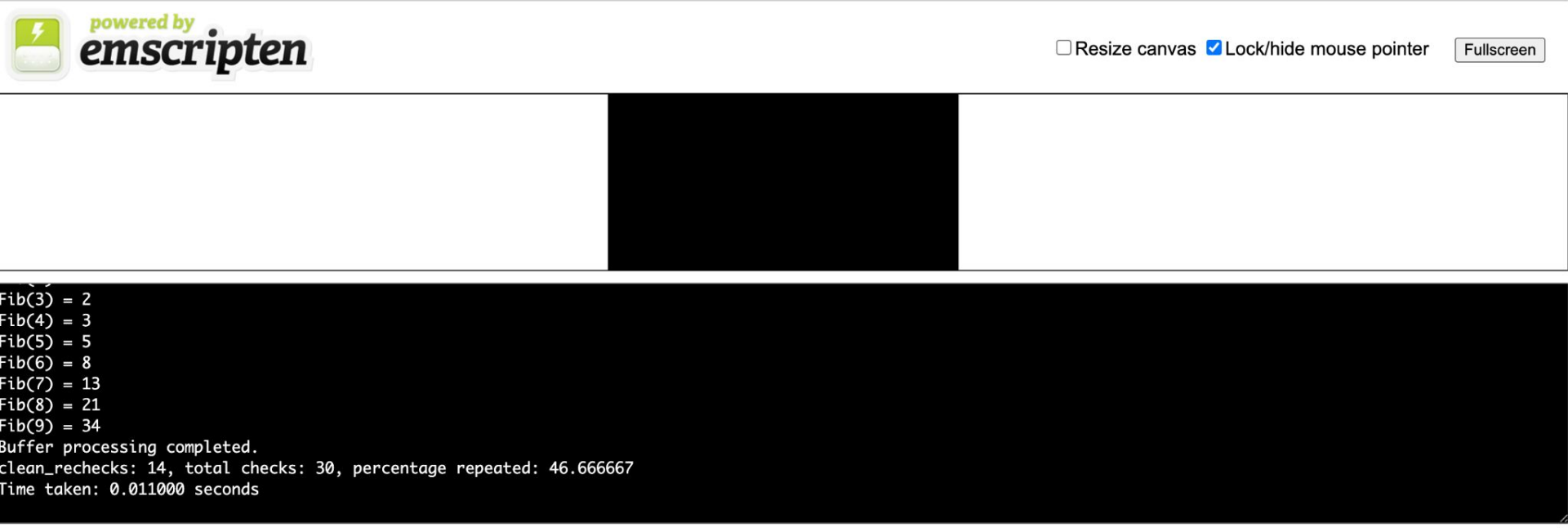
10000 runs 0.043790 seconds
100000 runs 0.386574 seconds
200000 runs 0.756544 seconds
400000 runs 1.505169 seconds
800000 runs 2.997267 seconds
1600000 runs 5.993094 seconds

10000 runs 0.002308 seconds
100000 runs 0.023105 seconds
1000000 runs 0.153953 seconds
2000000 runs 0.083223 seconds
20000000 runs 0.752998 seconds
40000000 runs 1.508573 seconds
80000000 runs 3.000045 seconds

Other metrics (fibonacci and matrix multiplication?)

Emscripten (back to Wasm, how is it?)

- Convert C module back to a Wasm module with minimal changes
- Demonstrates flexibility of compiling back and forth



Further work

- Returning pointers may be redundant
- Create a basic static analyzer to see if performance benefits can be realized
- Working on converting more complicated C modules back to wasm

Conclusion

- For common programs, we can expect to eliminate an upper bound of 6.54% of bounds checks
- Modules created from wasm2c can be converted back to wasm modules with Emscripten

References