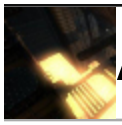




# Pause and Resume WebAssembly with Bysyncify

Sep 9, 2019

Pausing and resuming code can be useful for various things, like implementing coroutines, patterns like async/await, limiting how much CPU time untrusted code gets, and so forth. If you are customizing a WebAssembly VM then you have various ways to do this (direct jumps, longjmp, running in an interpreter, etc.). On the other and, if you want to do this in “userspace”, that is, if you are running inside of a standard wasm VM and you want to transform some wasm so that it can be paused and resumed, then Bysyncify provides a way to do that. For example, you can use it to pause and resume wasm that you run on the Web, like if you have some synchronous code that you don’t want to rewrite to be asynchronous, but need it to be - then Bysyncify can do that for you automatically!

Bysyncify is implemented as a Binaryen pass. It provides **low-level** support for pausing and resuming wasm by instrumenting the code and providing special functions that control **unwinding** and **rewinding** the call stack, preserving all the local state while doing so. We’ll see examples below of how to do that - basically, you can start with normal synchronous wasm code, run Bysyncify on it, and then easily control unwinding and rewinding the call stack from either wasm or JS.

Hopefully WebAssembly will support coroutines natively in the future. Another possibility here is to use threads (by blocking on another thread), but that can only work in browsers that support threads, and only in a worker. Bysyncify is another option in the meantime, which works everywhere but adds some amount of overhead. We’ll look into performance in depth later on - in many cases that overhead is surprisingly low!

Let’s start with three examples of how to use Bysyncify: in pure wasm, in JS plus wasm, and in C using Emscripten.

# Example: Pure wasm

To use Bysyncify in pure wasm, define imports to the `bysyncify.*` API. Bysyncify will turn those into direct calls to the implementations as it adds them. That API lets you control unwinding and rewinding the call stack:

- `bysyncify.start_unwind`: Starts to unwind the call stack. Receives a parameter to a data structure that will store the information about the call stack and local state, see below.
- `bysyncify.stop_unwind`: Stops unwinding the call stack. Must be called when you reach the end of the stack you want to unwind.
- `bysyncify.start_rewind`: Starts to rewind the call stack. Receives a parameter to the data structure used earlier to unwind, and rewinds to that exact position.
- `bysyncify.stop_rewind`: Stops rewinding the call stack. Must be called when you reach the top of the stack you want to rewind, that is, you finished returning to the previous position.

Here's a simple example:

```
;; input.wat
(module
  (memory 1 1)
  (import "spectest" "print" (func $print (param i32)))
  (import "bysyncify" "start_unwind" (func $bysyncify_start_unwind (
  (import "bysyncify" "stop_unwind" (func $bysyncify_stop_unwind))
  (import "bysyncify" "start_rewind" (func $bysyncify_start_rewind (
  (import "bysyncify" "stop_rewind" (func $bysyncify_stop_rewind))
  (global $sleeping (mut i32) (i32.const 0))
  (start $runtime)
  (func $main
    (call $print (i32.const 1))
    (call $sleep)
    (call $print (i32.const 3))
  )
  (func $sleep
    (if
```

```
      (i32.eqz (global.get $sleeping))
      (block
        ;; Start to sleep.
        (global.set $sleeping (i32.const 1))
        (i32.store (i32.const 16) (i32.const 24))
        (i32.store (i32.const 20) (i32.const 1024))
        (call $bysyncify_start_unwind (i32.const 16))
      )
      (block
        ;; Resume after sleep.
        (call $bysyncify_stop_rewind)
        (global.set $sleeping (i32.const 0))
      )
    )
  )
  (func $runtime
    ;; Call main the first time, let the stack unwind.
    (call $main)
    (call $bysyncify_stop_unwind)
    ;; We could do anything we want around here while
    ;; the code is paused!
    (call $print (i32.const 2))
    ;; Set the rewind in motion.
    (call $bysyncify_start_rewind (i32.const 16))
    (call $main)
  )
)
```

This little example uses the `spectest` import for logging, which works in the official wasm test suite, and so it should work in any interpreter compatible with that, like the spec interpreter or Binaryen's `wasm-shell`. Here is how you can build and run it:

```
$ wasm-opt input.wat --bysyncify -O --print > output.wat
$ wasm-shell output.wat
BUILDING MODULE [line: 1]
(i32.const 1)
(i32.const 2)
(i32.const 3)
```

First we process the file with `wasm-opt`, running the Bysyncify pass and also optimizing (optimizing is very important here for size and speed; see details later). We then print out the result in text form, since that's what `wasm-shell` expects (otherwise we could do `-o output.wasm` to get a binary). Then if we run it in the shell we get some logging from that tool (`BUILDING ..`), and then the expected three logged numbers: `main` prints `1`, then while it is sleeping we print `2`, and then after we resume `main` continues and prints `3`.

Note how even though we call `main` twice (once to start it, once to continue after sleeping), it only executes once (split into two parts), and therefore we only see `1` and `3` printed once. For comparison, if we ran the original uninstrumented `input.wat` (after removing the `bysyncify_*` calls from it), then we'd get this: `(i32.const 1) (i32.const 3) (i32.const 2) (i32.const 1) (i32.const 3)` When not instrumented `main` can't be paused once it starts to run, and it prints `1` and `3` every such time, giving us no opportunity to print `2` in the middle.

It may be confusing that we need to call `main` twice. The first time is obvious; the second is to start the rewinding of the call stack - we need the wasm VM to end up in the exact spot it was before, and the wasm VM manages the call stack, so we have to recreate the call stack by executing the same calls. So we start at the same place, and then the instrumented code makes sure to follow the right code paths in each function to unwind properly.

That also determines how `sleep` works, which is the function `main` calls to pause itself. `sleep` will be called twice, exactly like `main`: once when we run the program and we decide to sleep, and once when we finish rewinding the call stack up to where it was before - which is inside `sleep`! To handle such multiple calls a useful code pattern is used here, to check whether we are sleeping or not. If we aren't, that is the first call and we start to unwind; if we are then that is the second call and we finish the rewind, allowing normal execution to proceed.

An important detail here is the data structure that we pass a pointer to in `bysyncify_start_unwind` and `bysyncify_start_rewind`. This plays a similar role to the `jmp_buf` used with setjmp/longjmp, but it's pretty simple even if you're not familiar with that: it's basically a place to store information while we unwind and read it back while we rewind so that we can return to the exact same location and state. The `i32` passed to those methods must refer to a region in linear memory containing such a structure, which contains two fields that must be initialized before calling

`bysyncify_start_unwind`:

- `i32` at offset `0`: the index in linear memory of the start of the "bysyncify stack", a region of memory allocated for us.
- `i32` at offset `4`: the index of the end of that stack region. If the size of the stack is too small, the `bysyncify_*` functions will execute a wasm `unreachable` instruction (they check for a stack overflow). If you see such a trap happen from `bysyncify_*` then you need to increase the size here.

(This and other details are documented in the pass source.) In the example above the data structure starts at `16`, and the stack starts right after those two fields, at `24` (the end of the stack is at `1024`, but in this tiny example we'll need only a small fraction of it). In this example we have just one such structure, since that's all we need to pause and resume a single execution; to implement something like coroutines you would use one data structure for each.

One thing you may notice if you read the instrumented code is that `runtime` and `sleep` are not instrumented by Bysyncify, as it assumes any function calling its API is "runtime" code - the code that controls when to unwind and rewind, and in particular, unwinding must stop when it reaches there! This is necessary for using Bysyncify in pure wasm, as otherwise all the wasm would be instrumented and unwinding would return to the place that called into the wasm. In the next section we'll see how to unwind and rewind from JavaScript, in which case the "runtime" is outside the wasm.

# Example: JavaScript

// TODO: ensure a Binaryen tag

Controlling unwinding and rewinding from JavaScript is very easy: Bysyncify exports the four API methods, and you can use them in a similar way as in the last section. Just for fun in this example we'll show how to do everything in JS, including running Binaryen to call Bysyncify, which we can do thanks to binaryen.js. You can get it with `npm install binaryen`. Then run this JavaScript in `node`:

```
// example.js
```

```
var binaryen = require("binaryen");

// Create a module from text.
var module = new binaryen.parseText(`
  (module
    (memory 1 1)
    (import "env" "before" (func $before))
    (import "env" "sleep" (func $sleep (param i32)))
    (import "env" "after" (func $after))
    (export "memory" (memory 0))
    (export "main" (func $main))
    (func $main
      (call $before)
      (call $sleep (i32.const 2000))
      (call $after)
    )
  )
`);

// Run the Bysyncify pass, with (minor) optimizations.
binaryen.setOptimizeLevel(1);
module.runPasses(['bysyncify']);

// Get a wasm binary and compile it to an instance.
var binary = module.emitBinary();
var compiled = new WebAssembly.Module(binary);
var instance = new WebAssembly.Instance(compiled, {
  env: {
    before: function() {
      console.log('before!');
      setTimeout(function() {
        console.log('(an event that happens during the sleep)');
      }, 1000);
    },
    sleep: function(ms) {
      if (!sleeping) {
        // We are called in order to start a sleep/unwind.
```

```
        console.log('sleep...');
        // Fill in the data structure. The first value has the stack
        // which for simplicity we can start right after the data st
        view[DATA_ADDR >> 2] = DATA_ADDR + 8;
        // The end of the stack will not be reached here anyhow.
        view[DATA_ADDR + 4 >> 2] = 1024;
        exports.bysyncify_start_unwind(DATA_ADDR);
        sleeping = true;
        // Resume after the proper delay.
        setTimeout(function() {
          console.log('timeout ended, starting to rewind the stack')
          exports.bysyncify_start_rewind(DATA_ADDR);
          // The code is now ready to rewind; to start the process,
          // first function that should be on the call stack.
          exports.main();
        }, ms);
      } else {
        // We are called as part of a resume/rewind. Stop sleeping.
        console.log('...resume');
        exports.bysyncify_stop_rewind();
        sleeping = false;
      }
    },
    after: function() {
      console.log('after!');
    }
  }
});
var exports = instance.exports;
var view = new Int32Array(exports.memory.buffer);

// Global state for running the program.
var DATA_ADDR = 16; // Where the unwind/rewind data structure will l
var sleeping = false;

// Run the program. When it pauses control flow gets to here, as the
// stack has unwound.
exports.main();
```

```
console.log('stack unwound');
exports.bysyncify_stop_rewind();
```

Here is the output from running that code:

```
before!
sleep...
stack unwound
(an event that happens during the sleep)
timeout ended, starting to rewind the stack
...resume
after!
```

The key thing here is that we start a sleep, unwind the stack, and can then handle an event - that event would not arrive if we were not running asynchronously! After that, we rewind the stack, and proceed normally.

Most of the details here are direct parallels to the pure wasm example from earlier: `sleep` is called more than once, we use a similar data structure, and so forth.

# Example: Emscripten

// TODO: ensure an Emscripten release

The first two examples showed how to use Bysyncify at a low level, basically implementing your own runtime. Let's see a higher-level example now where the runtime is already provided: writing C code using Emscripten.

Emscripten needs something like Bysyncify because the native APIs that Emscripten supports (POSIX file reading, etc.) are often synchronous, while Web APIs are generally asynchronous. For that reason Emscripten has had the Asyncify and Emterpreter-Async features, which help codebases be ported to the Web that otherwise would need a massive refactoring. Emscripten can use Bysyncify as a third-generation solution here. To do so, simply build with

```
emcc -s BYSYNCIFY [..]
```

That will run Bysyncify itself, and enable synchronous versions of various APIs like `emscripten_sleep, emscripten_wget`, etc. For example, you can write code like this (note: we use EM_JS to make it convenient to mix JS and C):

```
// example.cpp
#include <emscripten.h>
#include <stdio.h>

// start_timer(): call JS to set an async timer for 500ms
EM_JS(void, start_timer, (), {
  Module.timer = false;
  setTimeout(function() {
    Module.timer = true;
  }, 500);
});

// check_timer(): check if that timer occurred
EM_JS(bool, check_timer, (), {
  return Module.timer;
});

int main() {
  start_timer();
  // "Infinite loop", synchronously poll for the timer.
  while (1) {
    if (check_timer()) {
      printf("timer happened!\n");
      return 0;
    }
    printf("sleeping...\n");
    emscripten_sleep(100);
  }
}
```

This contains an “infinite loop” that you normally can’t do on the Web - no event (including the `setTimeout`) will happen until you return to the main event loop. But if you compile that with `emcc example.cpp -s BYSYNCIFY` and run `nodejs a.out.js`, then you’ll see something like

```
sleeping...
sleeping...
sleeping...
sleeping...
sleeping...
timer happened!
```

With Bysyncify those sleeps are actual returns to the main event loop!

Implementing something like `emscripten_sleep` is very simple using Emscripten's JS runtime support: it's basically just this:

```
function emscripten_sleep(ms) {
  Bysyncify.handleSleep(function(wakeUp) {
    setTimeout(wakeUp, ms);
  });
}
```

`handleSleep` handles the "double call" issue from before automatically: you just provide it with the code to run (here, a `setTimeout`), and you call `wakeUp` at the right time in the future, and everything just works!

You don't need to look at the implementation of handleSleep in order to use the API in C, but it may be interesting if you're thinking of implementing support for Bysyncify in another language outside of Emscripten. (If so, let me know if I can help!)

Note that Emscripten only supports Bysyncify in the new LLVM wasm backend path; it won't work with the older "fastcomp" backend, which is where Asyncify and the Emterpreter work. In other words, if you use Asyncify or the Emterpreter then you will need to upgrade to the wasm backend and to Bysyncify at the same time.

# How Bysyncify works

You don't need to understand how Bysyncify works in order to use it. If you're not interested in that you can skip this section.

The basic capabilities we need in something like Bysyncify are to unwind and rewind

the call stack, to jump back to the right place in the middle of the function in each case, and to preserve locals while doing so. Saving and reloading locals is fairly straightforward; what is trickier is to get back to the right place in the middle of control flow. There are many possible ways to do this. As mentioned earlier Emscripten has had the Asyncify and Emterpreter features for this, and Bysyncify tries to improve on them, so it's interesting to briefly summarize the history here.

## Asyncify

Asyncify works on LLVM IR. It adds new control flow branches as needed, so that the entry to the function can reach all possible places we may resume at, and checks for unwinding after each call. An advantage here is that we can let the LLVM optimizer run, so it may be able to improve that control flow in some ways. A disadvantage though is that the transformed code is fairly pathological, as it can contain irreducibility like branches directly into a deeply nested inner loop, which wasm can't represent. The process of fixing up such complex control flow may end up unoptimal.

Local state is also a problem for code size, as Asyncify by necessity operates on LLVM's SSA registers. There are usually many more such registers than there are locals in the final wasm, and worse, each change to control flow can cause more locals to be needed due to merges and phis. In practice we saw huge code size increases sometimes which limited usability.

Note that Asyncify has no relation to LLVM coroutines (LLVM added them in 2016, Asyncify is from 2014). LLVM coroutines avoid the above problem with local state by not doing the full lowering at the IR level, which is good, but also means that each backend must support them, and the LLVM wasm backend doesn't yet. It may be interesting to implement coroutines there; one option might be to use Bysyncify for that.

## Emterpreter

The Emterpreter is a little interpreted VM implemented in asm.js that runs a custom bytecode (which asm.js is compiled into). As a VM, it can easily pause and resume execution, since the locals are already on the stack, and there is an actual program counter! This also has a guarantee of not increasing code size, since the bytecode is smaller than asm.js or wasm, and the VM itself is negligible in anything but a trivially

small program.

The obvious problem of course is that as an interpreter this is quite slow. The main solution to that is *selective interpreting*: by telling the Emterpreter which code to convert and which to leave at full speed, you can keep the important code at full speed. That works if whenever you unwind/rewind there is only emterpreted code on the stack (as we can't unwind/rewind anything else). In other words, if you have something like this (in JS-like pseudocode):

```
function caller() {
  var x = foo();
  sleep(100);
  return x;
}
```

Then if `foo` can't unwind the stack, you can run it at full speed. You only need to emterpret `caller`, which may be fine if doesn't take a significant amount of time itself anyhow.

# Bysyncify

As mentioned earlier Bysyncify tries to improve on those earlier approaches. The first design decision is that it operates on wasm. That avoids the problem with many SSA registers that we mentioned earlier, and also Bysyncify integrates with the Binaryen optimizer to reduce the number of locals as much as possible.

The big question is then what to do about control flow. Bysyncify does something like this to that last code snippet (again, in JS-like pseudocode):

```
function caller() {
  var x;
  if (rewinding) { .. restore x .. }
  if (normal) {
    x = foo();
  }
  if (normal || .. check call index ..) {
    sleep(100);
```

```
    if (unwinding) {
      .. save call index and x ..
      return;
    }
  }
  return x;
}
```

This may look confusing at first, but it's really pretty simple. If we are rewinding, we start by restoring the local state. We also we guard most code with checks on whether we are running normally - if we are rewinding, we must skip that code, since we've already executed it. When we reach a call that might unwind the stack, we have a "call index" for it so that we know which call to return to inside each function. If the call starts an unwinding, we save the call index and local state and exit immediately. Or if we are rewinding, then the call will set the state to normal, and then we will simply continue to execute on from the right place.

The key principle here is that we don't do any complex CFG transformations. Instead, Bysyncify's approach is to **skip code while rewinding**, always moving forward so that we eventually get to the right place. Importantly, we can put those ifs around whole clumps of code that can't unwind, like a loop without a call:

```
if (normal) {
  for (var i = 0; i < 1000; i++) {
    total += i;
  }
}
```

The entire loop is inside the if, which means that it runs at full speed! This is a big part of why Bysyncify is faster than you'd expect. Also, those ifs are well-predicted so modern CPUs aren't slowed down by them much, assuming unwind/rewinding are fairly rare events. Another optimization Bysyncify does is a whole-program analysis to see what can unwind the stack, which is why we didn't check for unwinding after calling `foo`. Bysyncify will also not modify a function at all if it sees it doesn't need to. And unlike Asyncify, Bysyncify avoids any unpredictable overhead from irreducibility since it never creates any.

In summary, the general idea is to keep skipping code while rewinding. This may be

less efficient than a direct CFG branch straight to the right place, but it is simple as well as predictable both in the code we generate and its runtime performance; and by analyzing the entire program we only add that overhead where it is actually needed. Next we'll see performance numbers for this.

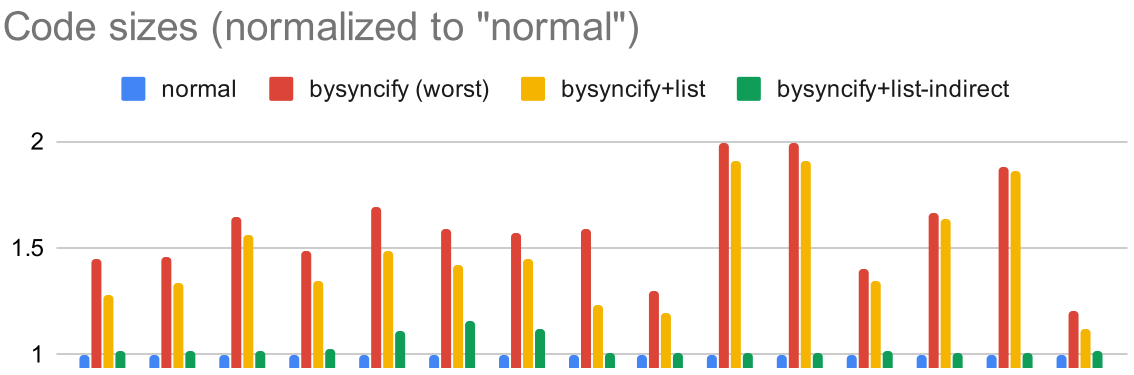
# Measurements

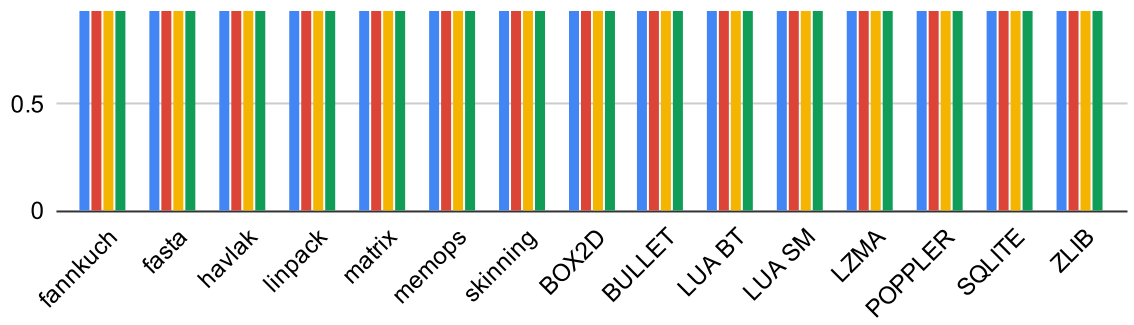
## General overhead

Bysyncify's checks about unwinding and rewinding the stack are expected to add overhead since they add work, and they can affect the liveness and interferences of locals etc. How expensive is this?

We'll focus on code size here; see notes later down on speed. There are 4 interesting measurements to make here, that we'll see in the chart below:

- **normal** - The benchmark compiled normally.
- **bysyncify (worst)** - The benchmark compiled with bysyncify in the most naive, worst way. Here we assume that *any* call to an import may unwind/rewind the stack (we consider imports because Emscripten controls unwinding/rewinding in JS, and not from inside wasm).
- **bysyncify+list** - Same as the last, but also with a list of the only imports that can unwind/rewind the stack (using the option `bysyncify-imports` to Bysyncify; see the pass source for details).
- **bysyncify+list-indirect** - Same as the last, but also assuming indirect calls can't lead to an unwind/rewind of the stack (using the option `bysyncify-ignore-indirect`).






Code size measurements in bytes (lower is better), compiled with Emscripten.

The first thing to note is that the size overhead is in the 1-2x range, that is, the binary is around 50% larger on average. Even on the realistic macrobenchmarks on the right (in ALL CAPS) it barely exceeds 2. In other words, there is significant overhead, but it's fairly predictable and not extreme. That this is what we see on the worst case, where we instrument all the code, is encouraging!

Looking more in depth, if we compare "bysyncify (worst)" to "bysyncify+list", then there is a noticeable improvement. For example, on Box2D it's about 25% smaller. What's going on here is that if we tell Bysyncify which imports can start an unwind/rewind and which can't, then its whole-program analysis can figure out that a lot of code doesn't need to be instrumented at all.

However, this fails on larger programs for a specific reason: **indirect calls**. Once you have enough of them it becomes very hard to statically analyze control flow. Poppler is the largest benchmark here, and it hardly benefits from the list of imports for this reason.

If we compare "bysyncify+list" to "bysyncify+list-indirect", where we ignore indirect calls, then almost all the overhead vanishes, on every single benchmark! Of course, "bysyncify+list-indirect" is the best possible case: we tell Bysyncify exactly what can unwind/rewind, and we also don't actually have any unwind/rewind operations here, since these are normal computational benchmarks - they don't actually call `sleep`! The worst case was still realistic because Bysyncify thinks any import can unwind/rewind, so it instruments lots more code than realistically necessary even if we had `sleep` calls. And the point of the best case is that it shows that with the right information we can remove all the overhead.

So far we talked about code size. The numbers for speed are mostly similar, or better - that is, if the binary is 20% bigger, it tends to be at worst 20% slower. However, I did

notice one large outlier, SQLite, on which the slowdown is around 5x. That appears to be because of the huge interpreter-like function, which goes from 150K (already big!) of wasm binary to 300K. The larger problem is probably with the locals: they go from 25 to 27, which doesn't seem so bad (Binaryen works hard to keep that number low!) but they hide the fact that all the extra branches and the saving/restoring code for the locals increases their live ranges dramatically - for many of them, across the whole function, which means a lot more interference, spilling and so forth; VMs may also limit compilation to the baseline tier on such pathological code. A 5x slowdown is not that surprising on such an extreme case.

**In summary**: The general overhead of Bysyncify can be limited to occur only where it is actually needed (if you can avoid indirect calls being in the way) and where it does occur it should do no worse than double size / halve speed for most code. However, extremely massive functions may end up with larger slowdowns.

## More on indirect calls

Using a list of imports is probably possible for most use cases. In Emscripten for example we pass it `emscripten_sleep` and other relevant sync APIs and syscalls. However, ignoring indirect calls is less obvious - some programs simply do have indirect calls on the call stack for an unwind/rewind operation, and the overhead there may approach the worst case from before. In such cases the true solution for maximal performance is probably a new WebAssembly spec proposal for coroutines, as mentioned earlier.

On the other hand, it is also common to only need that support in the main event loop, something like this:

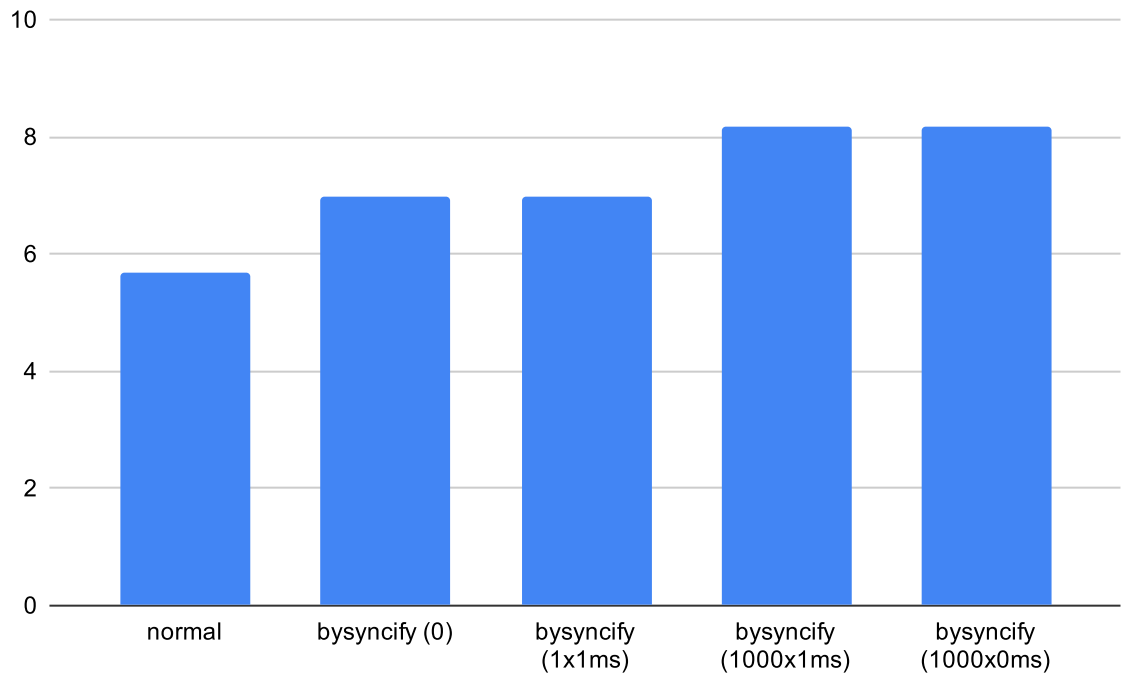
```
int main() {
  startUp();
  while (1) {
    renderFrame();
    handleEvents();
    sleep(timeToNextFrame());
  }
}
```

This type of "infinite loop" is common in games and other things. If that is the only call

to sleep, then it is perfectly safe to ignore indirect calls in Bysyncify, and when passing that flag it should only end up instrumenting `main()` itself. In that case, the overhead should be almost zero! Overall, if you can ignore indirect calls when using Bysyncify, it can be extremely helpful. To do so with `wasm-opt` pass `--pass-arg=bysyncify-ignore-indirect`, or in Emscripten use `-s BYSYNCIFY_IGNORE_INDIRECT`.

## Unwind/rewind speed

The measurements before looked at the general overhead: how much bigger code becomes and how much slower it is when running normally. Now let's take a look at how fast we can "context switch", that is, unwind and rewind the stack. Here are some numbers on the fannkuch benchmark, modified to sleep in the most inconvenient place (the innermost loop).




Unwind/rewind measurements on fannkuch, in seconds (lower is better).

The first two bars show that just enabling Bysyncify adds some overhead (22%), even without actually sleeping - that's the general (worst-case) overhead we measured before. The other bars show what happens when we do actually sleep: 1 time for 1ms, 1000 times for 1 ms, or 1000 times for 0ms. A single sleep's impact is so small it's basically impossible to measure, which is good! A thousand sleeps of 1ms should add 1 second (the total of the time spent sleeping); in practice it adds 1.18 seconds, with

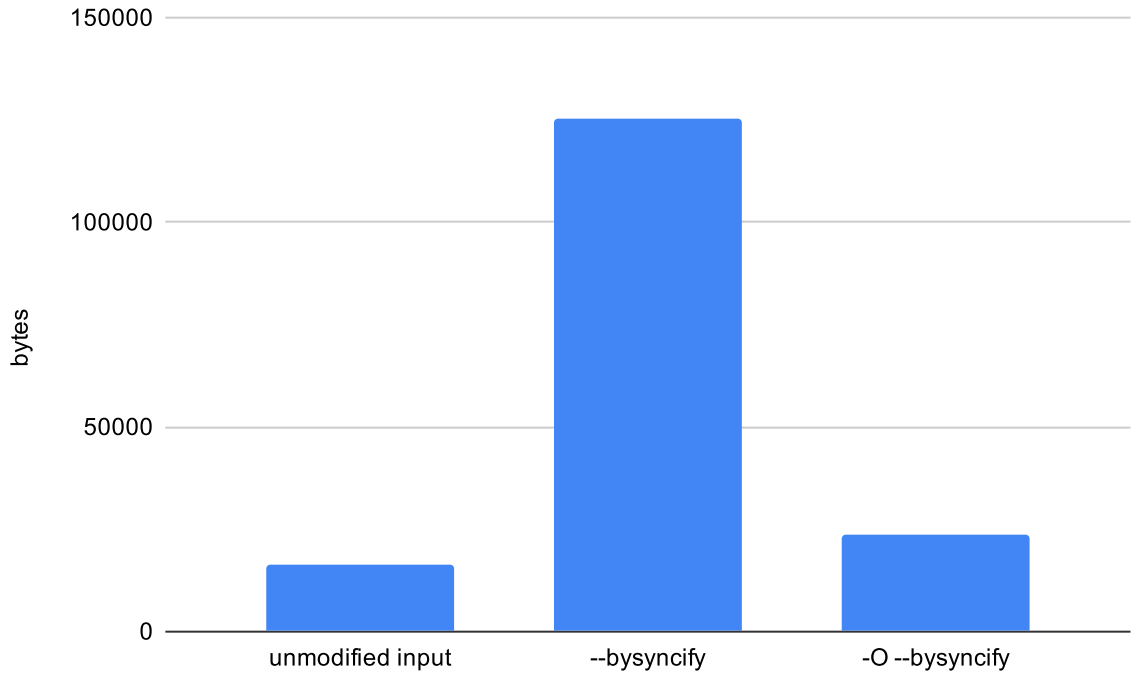
some noise (not pictured). That it's close to the theoretical minimum is good, and the noise suggests the overhead is related to the accuracy of the timer used in setTimeout. Indeed, doing the same amount of sleeps for 0ms (which should immediately resume in the next event loop without waiting at all) takes about the same time, so the event loop itself is adding much of that extra overhead.

**In summary**: The unwind/rewind overhead of Bysyncify is basically negligible if you are using it to do anything asynchronous. (It would also be interesting to measure something non-asynchronous, like swapping between two coroutines, but I don't have a good benchmark for that and this post is quite long already!)

## The importance of optimization

In the examples earlier we told Binaryen to optimize while it ran Bysyncify. Binaryen doesn't optimize by default, because that keeps things as modular as possible - each pass, like Bysyncify, does the least possible by itself, and it's simpler to write such passes if we assume that the other passes will optimize for us.

It is **very important** to optimize while running Bysyncify, for both code size and speed, as the following numbers on the Fannkuch benchmark show:




Fannkuch code size measurements (bytes; lower is better).

Running `wasm-opt --bysyncify` *without* optimizations leads to huge code sizes, while `-O --bysyncify` (which uses Binaryen's default optimization level) produces code sizes like what we'd expect given the data from before. Remember to optimize!

## Final thoughts

Hopefully Bysyncify is useful for people that have synchronous wasm code they want to run asynchronously, or to pause and resume, etc. If you do something cool with it, or you find a bug, let me know!