Core Stack Switching Implementation Strategies (discussion)

Daniel Hillerström

The University of Edinburgh, UK

May 19, 2025

WebAssembly Stacks Subgroup

https://dhil.net

Implementation strategies

- Stack as a linked list of frames (e.g. Danvy (1987))
- Stack copying and pasting (e.g. Leijen (2017))
- Virtual memory mapped stacks (e.g. Leijen and Sivaramakrishnan (2021))
- Segmented stacks (e.g. Bruggeman, Waddell, and Dybvig (1996), Alvarez-Picallo et al. (2024))
- Whole-program transformations (e.g. Hillerström (2022), Zakai (2019))

Implementation strategies

- Stack as a linked list of frames (e.g. Danvy (1987))
- **Stack copying and pasting** (e.g. Leijen (2017))
- **Virtual memory mapped stacks** (e.g. Leijen and Sivaramakrishnan (2021))
- **Segmented stacks** (e.g. Bruggeman, Waddell, and Dybvig (1996), Alvarez-Picallo et al. (2024))
- Whole-program transformations (e.g. Hillerström (2022), Zakai (2019))

Methodology

Setup

- Source language: C with a bespoke fiber library
  - Stack copying: Libhandler backend (Leijen 2017)
  - Virtual memory stacks: Libmprompt backend (Leijen and Sivaramakrishnan 2021)
  - Segmented stacks: Libseff backend (Alvarez-Picallo et al. 2024)
- Disclaimer: mostly off-the-shelf configurations
- Three CPU-bound microbenchmarks:
  - Iterative summing
  - Recursive tree summing
  - Prime sieve computation
- Disclaimer: not particularly representable of ‘real-world’ workloads
- Using `hyperfine` to compare benchmarks end-to-end

Source code: github.com/dhil/core-stack-switching-experiments

Fibers interface in C

```
/** The signature of a fiber entry point. **/
typedef void* (*fiber_entry_point_t)(void*);
/** The abstract type of a fiber object. **/
typedef struct fiber* fiber_t;

/** Allocates a new fiber with the default stack size. **/
fiber_t fiber_alloc(fiber_entry_point_t entry);
/** Reclaims the memory occupied by a fiber object. **/
void fiber_free(fiber_t fiber);

/** Yields control to its parent context. **/
void* fiber_yield(void *arg);

/** Possible status codes for ‘fiber_resume‘. **/
typedef enum { FIBER_OK, FIBER_YIELD, FIBER_ERROR } fiber_result_t;

/** Resumes a given ‘fiber‘ with argument ‘arg‘. **/
void* fiber_resume(fiber_t fiber, void *arg, fiber_result_t *result);
```

Iterative sum (1)

Characteristics

- 2 coroutines, repeatedly yielding control in a tight loop
- Shallow call stack depth

Benchmark essence

```
for (size_t i = 0; i < N; i++) {
  result += fiber_yield(&i);
}
```

Iterative sum (2)

		Implementations		
		Stack copying	Virtual stacks	Segmented stacks (baseline)
N	10000	4.11	1.71	1.0
	100000	8.85	2.45	1.0
	1000000	11.15	2.86	1.0

Lower is better

Tree sum (1)

Characteristics

- 2 coroutines
  - 1 coroutine walks the tree recursively
  - 1 coroutine yields in a tight loop
- Deep call stack depth

Benchmark essence

```
void walk_tree(node_t *node) {
  if (node->tag == LEAF) {
    fiber_yield(node->val);
  } else {
    walk_tree(node->left);
    walk_tree(node->right);
  }
}
```

Tree sum (2)

		Implementations		
		Stack copying	Virtual stacks	Segmented stacks (baseline)
N	21	11.22	2.81	1.0
	22	11.32	2.83	1.0
	23	11.59	2.86	1.0

Lower is better

Prime sieve (1)

Characteristics

- Dynamically spawns coroutines (up to N)
- Each coroutine holds a unique prime number
- Shallow call stack depth

Benchmark essence

```
bool filter(int32_t my_prime) {
  bool divisible = false;
  int32_t candidate = fiber_yield(NULL);
  while (candidate > 0) {
    divisible = (candidate % my_prime) == 0;
    candidate = fiber_yield(divisible);
  }
  return false; // dummy value
}
```

Prime sieve (2)

		Implementations		
		Stack copying	Virtual stacks	Segmented stacks (baseline)
N	1000	9.36	3.87	1.0
	4000	8.09	2.87	1.0
	8000	7.79	2.65	1.0

Lower is better

References I

Danvy, Olivier (1987). “Memory allocation and higher-order functions”. In: *PLDI*. ACM, pp. 241–252. DOI: 10.1145/29650.29676. URL: https://doi.org/10.1145/29650.29676.

Bruggeman, Carl, Oscar Waddell, and R. Kent Dybvig (1996). “Representing Control in the Presence of One-Shot Continuations”. In: *PLDI*. ACM, pp. 99–107. DOI: 10.1145/231379.231395. URL: https://doi.org/10.1145/231379.231395.

Leijen, Daan (2017). “Implementing Algebraic Effects in C - "Monads for Free in C"”. In: *APLAS*. Vol. 10695. Lecture Notes in Computer Science. Springer, pp. 339–363. DOI: 10.1007/978-3-319-71237-6_17. URL: https://doi.org/10.1007/978-3-319-71237-6_17.

Zakai, Alon (2019). *Pause and Resume WebAssembly with Binaryen’s Asyncify.* Accessed 2022-10-27. URL: https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html.

Leijen, Daan and KC Sivaramakrishnan (2021). *libmprompt: Robust multi-prompt delimited control and effect handlers in C/C++.* https://github.com/koka-lang/libmprompt.

Hillerström, Daniel (2022). “Foundations for Programming and Implementing Effect Handlers”. PhD thesis. The University of Edinburgh, UK. DOI: 10.7488/era/2122. URL: http://dx.doi.org/10.7488/era/2122.

References II

Alvarez-Picallo, Mario et al. (2024). “Effect Handlers for C via Coroutines”. In: *Proc. ACM Program. Lang.* 8.OOPSLA2, pp. 2462–2489. doi: `10.1145/3689798`. url: `https://doi.org/10.1145/3689798`.