This is a manuscript submitted to ASPLOS'24. It was highly affirmed by the anonymous reviewers for its value in design of coroutine, as well as related components of systems software and computer architecture. It was, however, rejected due to lack of novelty: the key proposal, CACS, had already been implemented before in libfringe. So we give up the attempt to publish this manuscript as a research paper in ASPLOS, nor other academic conferences. And we upload it to Photon's project site as-is, hoping everyone gets the message that ***stackful coroutine has been made fast.***

Stackful Coroutine Made Fast

Submission #17 for ASPLOS'24

Huiba Li, Rui Du and Windsor Hsu
Alibaba Cloud

Abstract

Stackful coroutine, also known as user-space cooperative thread, offers the promise of more intuitive and accessible concurrent programming. With the growing demand for highly concurrent programs, stackful coroutine has gained increasing interest in recent years. It has, however, been much maligned for poor performance compared to stackless coroutine because of its heavy reliance on context switching. In this paper we perform in-depth measurement and analysis of several advanced implementations of stackful and stackless coroutine. Our analysis indicates that although current implementations of stackful coroutines are indeed significantly slower than their stackless counterparts, stackful coroutine is not intrinsically slow. Rather, stackful coroutine performs poorly mainly because the current implementations do not fully leverage the fact that control flow is cooperatively passed among stackful coroutines. Based on this observation, we propose context-aware context switching (CACS) among stackful coroutines. Instead of a full set of registers, CACS saves (restores) only the minimum necessary set of registers according to the caller (callee) context. It also enables the branch to be inlined at the caller site so that branch prediction is more accurate. We have implemented CACS in Photon, a highly optimized libOS based on coroutine. Performance measurements show that with the optimizations proposed in this paper, stackful coroutine out-performs stackless coroutine in most cases, and ties in the generator paradigm which is an especially challenging scenario for stackful coroutine. We also suggest a few supporting changes in computer architecture, programming language, compiler and OS that can further improve the performance of stackful coroutine. Our work is open-sourced on GitHub[1].

1 Introduction

Modern servers may have network connectivity with bandwidth that is on the same order as that of its memory or CPU interconnect [28], and host dozens of SSDs each offering more than 10GB/s of throughput [35]. With such advances in the hardware I/O capability, software stacks must become both highly concurrent and efficient to unleash the growing performance potential of the modern server. Recent software improvements to this effect include the development of high-performance I/O frameworks, such as DPDK [6] and SPDK [11], that budget to process a single packet or request in terms of CPU cycles.

Coroutine has also been gaining increasing attention in recent years to handle concurrent programming efficiently and effectively. Programming languages that have or are embracing coroutine include C++20 [3], Rust 1.39 [2] in 2019, C# 5.0 [16] in 2012, Python 3.5 [20] in 2015, JavaScript ES2017 [9], Swift 5.5 [12] in 2021, Java 21 [8] in 2023, Dragonwell 8 [1] in 2019, etc. Golang [15] has provided coroutine (goroutine) as a first-class construct since its initial design.

There are two types of coroutine — stackless and stackful. The former shares a default stack among all the coroutines while the latter assigns a separate stack to each coroutine. With stackless coroutine, the code is transformed into event handlers at compile time, and driven by an event engine at run time, i.e. the scheduler of stackless coroutine. Transferring control of CPU to a stackless coroutine is merely a function call with an argument pointing to its context. Conversely, transferring CPU control to a stackful coroutine requires a context switch. This context switch is widely regarded as a heavy-weight operation when compared to the function call. In reality, this context switch is much more efficient than a kernel task switch because it does not incur the overhead of a round-trip transition from user-space to kernel-space, and it is also possible to perform optimizations by making use of the cooperative nature of the coroutines within a single program.

Nevertheless, due primarily to the perceived performance concern (among other issues), more and more systems are abandoning stackful coroutine for stackless coroutine, especially systems that emphasize performance such as C++20, Rust, C#, Swift, etc. There was once a heated debate [30, 38–40] in the C++ standards committee over stackless or stackful coroutine for C++20. Although stackful coroutine is generally easier to use, more compatible with existing codebases and more efficient in many scenarios, the proponents of stackless coroutine constructed a microbenchmark [18] to show that "fibers (stackful coroutines) have 20 times larger context switch overhead" than stackless coroutine [39]. The defendants of stackful coroutine did not give a direct response to the challenge [38], and ultimately the committee adopted stackless coroutine for C++20. We believe that the demonstrated outsized difference in overhead played an important role in this decision, as well as similar decisions for other systems.

In this paper we argue that stackful coroutine is not intrinsically slow. It just has not been implemented to fully exploit the cooperative nature of stackful coroutine, and there are opportunities to make context switching more efficient. We perform in-depth measurement and analysis of several current coroutine implementations. Based on the analysis, we propose context-aware context switching (CACS) to improve

[1] For blind review purpose, we have put related resources at https://github.com/for-blinded-review. We'll try to merge them to corresponding upstreams after review.

the efficiency of register saving, accuracy of branch prediction, and hit rate of CPU cache. CACS leverages the fact that context switches among stackful coroutines occur only at specific caller sites. With CACS, each context switching saves only the registers that will be needed after it switches back, and these registers are determined by the compiler at the caller site. CACS also enables the branching to be inlined at the caller site so that branch prediction is more accurate. We apply CACS to optimize stackful asymmetric coroutine by designing in-stack generator. With CACS, it performs as efficiently as the corresponding stackless coroutine implementation. We also introduce a new function calling convention named `preserve_none` as an expansion of CACS for all switching functions to further improve performance. With our work the cost of yield operation (scheduling and switching to the next coroutine) is greatly reduced. CACS is implemented in Photon [10], a sophisticated libOS based on coroutine. The proposed calling convention is implemented in Clang [4].

On the other hand, we demonstrate that stackless coroutine is inefficient in handling multi-level invocation, an inevitable pattern in real-world programs. It even incurs an overhead proportional to the length of the call chain when dealing with recursion. Despite there are optimizations that can reduce this overhead to a constant in some scenarios, it is still much higher than the corresponding overhead with stackful coroutine. Our results suggest that stackful coroutine is the better choice for efficient concurrency.

The contributions of this paper are as follows:

1. We conduct an in-depth performance characterization of state-of-the-art implementations of both stackless and stackful coroutines, analyzing the root causes for various measured differences.
2. We observe that there are untapped opportunities to improve the performance of stackful coroutine by exploiting the fact that coroutines are cooperatively scheduled user-space threads.
3. We propose and implement CACS to optimize the performance of stackful coroutine, and demonstrate that it can effectively eliminate the performance concern of context switching, thereby raising the performance upper bound of stackful coroutine. CACS makes stackful coroutine even feasible for challenging scenarios such as the generator paradigm. The overall result is promising: a single Xeon CPU core can perform a yield operation in ~1.52 ns or ~3.34 cycles, comparable to the cost of a function call, and out-performing state-of-the-art result by several times.
4. While we demonstrate promising results with CACS, the current implementation is still constrained by existing architecture, programming language, compiler and OS. We suggest several supporting changes in these areas that can further improve the performance of stackful coroutine.

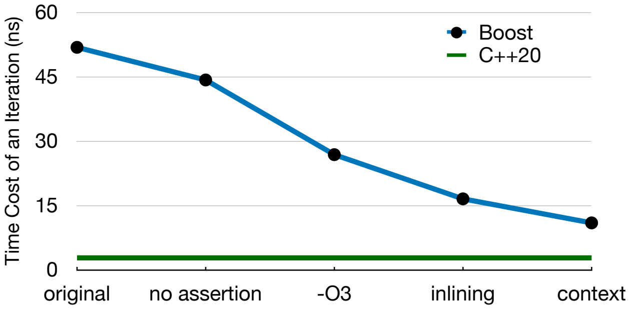


Figure 1. Breakdown of the ~20 times overhead of Boost stackful coroutine compared to C++20 stackless coroutine in the micro benchmark of sequence's sum.

2 Performance Measurement of Coroutines

In this section, we carry out in-depth measurement of coroutine implementations, trying to find out how such a big difference as "20 times" [39] in speed was formed, and whether there are opportunities for optimizations. We begin with the micro benchmark of sequence's sum that was used in the test, then expand its structure in two ways respectively: callee recursion (Hanoi) and caller nesting (write_fully) with concurrent execution. These expansions represents some common scenarios in real applications.

Besides the classification of stackful or stackless coroutine, they can also be classified as symmetric or asymmetric coroutine [37]. The former provides a single control-transfer operation that allows coroutines to explicitly pass control between themselves. The latter provides two control-transfer operations: one for invoking a coroutine and one for suspending it, possibly carrying return value(s) to the caller. Asymmetric coroutine can be resumed repeatedly to generate a series of return values. That's why it is also called generator.

2.1 Coroutines under the Microscope

We revisit the performance benchmark [18] that was used in C++ committee, and we are able to reproduce similar results in our environment, with latest compiler (Clang 15.0.3), latest dependent library (Boost 1.81), and trivial changes to the source code (e.g., removing the "experimental" namespace). The benchmark includes two cases, respectively for stackful coroutine implemented by Boost.Coroutine2 and stackless coroutine built-in C++20. Both of them consist of a generator that produces natural number from N down to 1, and a reducer that sums the numbers up.

We have managed to reduce the overhead of stackful coroutine down to ~4 times with some simple efforts such as: disabling assertion, raising up optimization level from -O2 to -O3, forced inlining all possible functions involved in the kernel loops, and re-implementing a third test case using Boost.Context directly. The overall result is depicted in Fig 1.

```
__attribute__ (( noinline ))
generator<uint64_t>
producer(uint64_t count) {
  for (; count!=0; --count)
    co_yield count;
}
```

```
__attribute__ (( noinline ))
generator<uint64_t>
producer(uint64_t count) {
  while(count)
    co_yield count--;
}
```

Figure 2. Stackless generator with for-loop and while-loop.

To find possible opportunities for further improvements, we analyze the implementation by disassembling its machine code. We find that a function jump_fcontext() contributes to most of the remaining overhead. It is a handcrafted assembly function consisting 24 instructions to perform context switching. It saves and restores a set of registers including `r12~r15`, `rbx`, `rbp`, x87 FPU control word (FCW), MMX/SSE control status register (MXCSR) and finally `rsp`, the stack pointer register. The list is determined by the function calling convention in use, and classic context switching functions save and restore all these registers in order to conform to the convention. But most of these registers are not used at all in such simple case as this, and it is a great waste to do it blindly. We believe it is better for the compiler to smartly determine what actually needs to be done before and after each context switching operation at the calling site, and do it all by itself. We have realized this optimization with some inline assembly code and a simple optional compiler extension. As a result, the cost of a context switching becomes similar to that of an ordinary function call. We'll describe this work in section 3.

It's also worth noting that, when we change the stackless producer()'s implementation from `for` loop to simpler and identical `while` loop, as shown in Fig. 2, its time cost increases unexpectedly by ~50% (see section 4 for details). We suspect it's due to the limitation of current compiler optimization for stackless coroutine, as it involves somewhat complex translation of the code.

2.2 Coroutines in the Telescope

Stackless coroutine can be inefficient when it yields some value to the original caller from a deep call chain of recursion, because the operation is designed to be targeting at the direct caller, and it may incur a high overhead of $O(n)$ to cross multiple levels. This scenario is depicted in Fig 3.

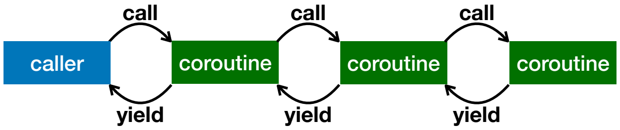


Figure 3. The recursive call chain of stackless coroutine implies a possible $O(n)$ complexity in most implementations.

```
void Hanoi(char n, char from, char to, char aux, Callback cb) {
  if (n==0) return;
  Hanoi(g, n-1, from, aux, to);
  cb(n, from, to); // move the n-th disk
  Hanoi(g, n-1, aux, to, from);
}
```

Figure 4. Hanoi using function recursion & callback.

Many important recursive algorithms fits this pattern, such as traversing a tree and yielding every node to the caller, or performing a depth-first-search in a graph (or other state spaces) and yielding each item found. Most coroutine implementations are subject to this problem, such as C#, Python, etc. Python provides "`yield from`" for this scenario to forward values from multi-level invocations, but only as a syntax sugar without any improvement to its performance. Rust has experimental support for generator, and it does not yet support multi-level invocations of generators. C++23 introduced an optimized implementation as `std::generator`, which achieves $O(1)$ delivery for multi-level invocations in stackless coroutines, by sophisticated manipulation of coroutine handles. But it still incurs a great overhead compared to generators based on stackful coroutine.

We study this problem by solving the classic puzzle Tower of Hanoi in both stackful and stackless coroutine. As depicted in Fig. 4, it is a simple recursive algorithm consisting only 4 lines of effective code. It is good a representative for recursion in real applications. We reimplement it respectively with C++20 stackless coroutine, C++23 stackless generator, and Boost stackful coroutine.

The results are shown in Fig 5, as relative time cost of coroutines compared to classic function recursion, so as to eliminate the exponential growth of the algorithm, and

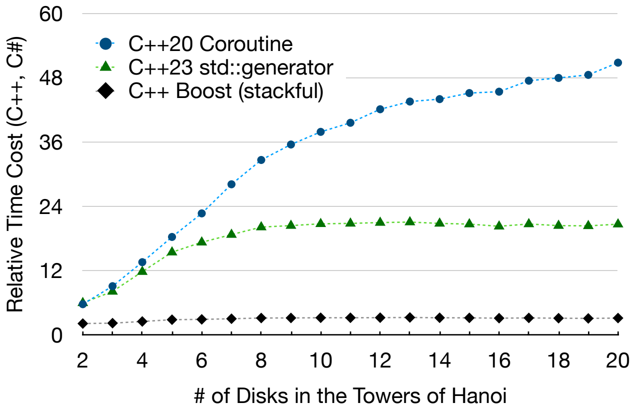


Figure 5. Time cost of coroutine recursion normalized to function recursion in their corresponding languages, solving the puzzle of Hanoi.

clearly reflect the overhead of coroutine yield and resume. The $O(1)$ and $O(n)$ overhead growths are clearly shown in the results, and the overhead of C++20 coroutine can even reach 50 times when the number of disks is 20. C++23 generator has $O(1)$ asymptotic growth, though, its overhead is still much higher than that of stackful coroutine with Boost. The results suggest that stackless coroutine is not efficient for recursion. And we'll also show in section 2.3 that it is not efficient either for non-recursive multi-level invocation, which is a common practice in non-trivial applications.

To further study C++23 std::generator (Gen23) and Boost, we use the `perf` tool to respectively monitor them solving Hanoi(1~20). We see that Gen23 has executed ~15 billions of instructions in ~5.8 billions of cycles (2.58 per cycle). And Boost has executed ~2.05 billions of instructions in 938 millions of cycles (2.19 per cycle). The former executes much more instructions at a higher IPC (Instructions Per Cycle) speed than the latter, indicating a more complex solution. We also notice that Boost has a 5.36% miss rate for branch prediction, which is much higher than that of Gen23 (0.76%). We believe it's due to the context switching function, jump_fcontext(), which jumps to a location different from that of the previous run, thus hard to predict. We'll solve this problem with context-aware context switching. See details in section 3.1.

2.3 Coroutines in the "Macroscope"

We measure how coroutines perform in a scenario of multiple conceptual threads of execution running concurrently. We simulate a server-type workload with both coroutine models, by creating 10 concurrent tasks, with each sending 100 MB of data through a network connection to its client. We mimic the non-blocking socket API that can send some number (>0) of bytes in each invocation if it is ready for write (there's some room in its internal buffer), and we have to repeat it in a loop until all data has been sent. In order that the measurement is repeatable and not too fast, we assume that every invocation to send_some() can send exactly 800 bytes, and is preceded by an invocation to `wait_for_ready()` that only switches execution to other tasks. This is effectively the pattern of non-recursive multi-level invocation. As the actual networking and event engine (e.g. epoll) are not the target of this test, they are not included in order that we focus on the coroutine part of the program.

Two implementations are involved in this measurement, C++20 stackless coroutine and Boost.Context stackful coroutine. We employ simple round-robin scheduling for both of them.

Fig. 6 shows the results of our measurement, including those from Rust and C# as references only. Boost achieves the best performance with stackful coroutine. We further study the cases in C++ with `perf` tool, in order to investigate any other reasons for such differences besides multi-level invocation. We find that C++20 coroutine incurs major overhead in memory allocation/deallocation for coroutine context. This indicates that memory pooling is very important for stackless coroutine, the ideal pooling might be a linear data structure for each conceptual thread of execution, and it grows with `co_await` and shrinks with `co_return`, as suggested in [38].

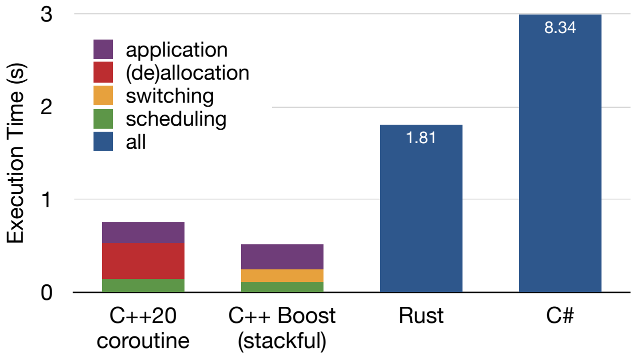


Figure 6. Execution time for simulation of concurrent write_fully(). Rust (based on tokio) and C# (compiled to native code) are included as references only.

We also find that the scheduler for C++20 coroutine incurs an overhead higher than that for Boost, despite we try to make both of them as simple and identical as possible. The question is answered with some extra counters: C++20 coroutine invokes scheduler several times more than Boost coroutine. It turns out that every `co_await` in the program makes the control return to the scheduler then calls the target coroutine from there. `co_return` and `co_yield` follow the reversed control flow. It is possible for stackless coroutine to apply an optimistic optimization: invoke the target first without returning to the scheduler, speculating that the target will complete without actually `co_yield`-ing or `co_await`-ing. This optimization can be applied repeatedly until it eventually has to go back to the scheduler, by successive returning. This pattern is useful for submitting asynchronous I/O operations that may complete immediately in some cases (e.g. there's enough internal buffer for write), so there is no need to wait for its completion in this case. We don't employ this optimization because we are focusing on the coroutine part of the program in the measurement, whereas this optimization avoids the part in the case of successful speculation.

We also notice with `perf` that the Boost-based case incurs a miss rate of 13.08% for L1 data cache load, which is much higher than the case with C++20 coroutine (0.01%). In such a micro benchmark, all data loads in the kernel loop should be cache-hit. We believe it's due to not-too-coincidental collisions in the set-associate algorithm of cache, and it should be caused almost deterministically by the default alignment of allocation for stack, combined with the homogenization of the running coroutines. And we believe that's why the application part of stackful case is slower than that of stackless case. The issue probably influences context switching too, because the context is also saved on stack. And the issue probably occurs in many other benchmarks and applications based on stackful coroutine or user-space threads, because this pair of conditions are easy to meet in real applications, especially on servers. We resolve this issue by simply introducing a random factor to the starting address of stack, making the stackful case significantly faster. See section 4 for further results.

3 Context-Aware Context Switching (CACS)

In this section, we present our proposals to make stackful coroutine fast. We first propose a novel approach to perform context-aware context switching (CACS), which saves and restores a least necessary number of registers decided by the compiler at the calling site. It makes use of inline assembly, and spreads out the indirect jump of CACS, possibly improving the rate of correct branch prediction. We then apply CACS to asymmetric coroutines by proposing in-stack generator, a new design that realize efficient yield() and resume(), as well as stack reuse in the case of non-concurrent generators. Thirdly we propose a new calling convention, preserve_none, for functions that are expected to switch context, so as to further improve performance. This is actually an expansion of CACS for caller functions to make sure the gains are not masked by them.

3.1 Design

As discussed in previous section, the classic context switching function saves and restores a fixed list of registers before making an indirect jump to the target location. The function is usually implemented in separate assembly code that is completely unaware of the functions that invoke context switching. All that it can do is saving and restoring every callee-saved register as specified in the function calling convention in use, treating every switching like a worst case function call.

To address the problem and make every cycle count, we propose a novel design of context-aware context switching (CACS), which saves and restores a least necessary number of registers decided by the compiler at the caller site. The compiler will generate instructions to save all necessary registers to stack before executing context switching, and restore them after switching. The compiler will also do a trade-off between saving / restoring a register and recomputing it.

We design CACS as a code template using standard inline assembly, as shown in Fig. 7. The instructions should be adapted to coroutine type and control structure definition. The key point is the bottom lines that define a list of all registers clobbered by the assembly code, so that the compiler will save and restore the registers that are needed after the code template. CACS is inlined at the caller site, making branch predictions for the jump easier to be correct. This is more useful for asymmetric coroutine (generator) where users directly make context switching without resorting to a scheduler. In contrast, classic context switching is typically implemented as a stand-alone assembly function, which is written and compiled without any knowledge of the particular contexts that make the invocation. If CACS is invoked at the end of a function, we can apply tail call optimization by using the function's return address as switch-back address, avoiding an extra indirect jumping.

```
__asm__ volatile (R"(
  lea 1f(%%rip), %%rax       # calculate switch−back address (label 1)
  push %%rax                 # store it to stack
  mov %%rsp, 0x10(%0)        # store sp to control struct `from`
  mov 0x10(%1), %%rsp.       # load sp from control struct `to`
  pop %%rax                  # load execution address from stack
  jmp *%%rax                 # jump to target coroutine
1:
)" : : "r"(from), "r"(to) : "rax","rbx","rcx","rdx","rbp","rsi",
"rdi","r8","r9","r10","r11","r12","r13","r14","r15"); # all regs
```

Figure 7. The code template for Context-Aware Context Switching (CACS) using inline assembly. The key point is informing the compiler that *all* registers will be clobbered by this code template, so the compiler itself will generate instructions before and after this code template to exactly save and restore necessary registers at this site.

3.2 In-Stack Generator with CACS

In the previous section stackful coroutine implemented with Boost remained ~4 times overhead compared to stackless coroutine in the microbenchmark of sum of sequence. According to our analysis, stackless coroutine transforms the code into event-driven form at compile time, and each invocation to the generator corresponds still to a function call. Whereas stackful coroutine creates a new stack for the generator, and each invocation corresponds to a context switching. We monitor the execution of both test cases with `perf` tool, and find that they show similar instruction per cycle (IPC): 2.29 for stackful and 2.49 for stackless. So their difference in performance must come primarily from the number of instructions executed in their kernel loops. By disassembling we find that, the context switching function, jump_fcontext(), consists 24 instructions, which gets executed twice in each iteration. While a pair of function call and return in this case costs only few instructions.

To resolve the problem, we (1) use CACS to greatly reduce the number of instructions for saving and restoring registers; we (2) reuse the current stack to further simplify switching (in addition to eliminating the allocation of a new stack); and (3) we exploit the fact that there exists a conceptual relationship of caller-callee, by defining a simple & efficient

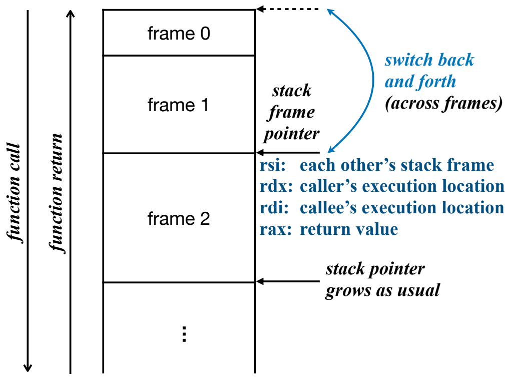

function call
function return
frame 0
frame 1
frame 2
⋮
stack frame pointer
switch back and forth (across frames)
rsi: each other's stack frame
rdx: caller's execution location
rdi: callee's execution location
rax: return value
stack pointer grows as usual

Figure 8. Stack layout and context (frame) switching of in-stack generator.

convention to ease the cooperation. We realize switchings with only 3 instructions, for both back and forth. The cost is similar to that of a function call / return, making stackful coroutine as performant as stackless coroutine.

We define new primitives to allow a function (generator) to switch the execution (yield) to the caller without terminating its execution, resulting a stack layout as shown in Fig. 8. The yield operation carries an application-defined value in register `rax` to the caller, as a form of "return value". The operation also carries additional information (frame pointer in `rsi` and instruction pointer in `rdx`), so that the caller can later resume the generator's execution. The resume operation is also a context switching, carrying additional information (frame pointer in `rsi` and instruction pointer in `rdi`) for the generator to yield back.

As shown in Fig. 9, an in-stack generator (`seq` or `hanoi`) is implemented like a normal function, with the first argument being GCTX*, which actually points to the caller's stack frame. And the function must create a GPromise object on entry, so as for the generator to yield results back to the caller.

```
u64 seq(GCTX* fp, u64 c) {
  GPromise gp(fp);
  while(c)
    gp.yield(c--);
  return 0;
}
u64 sum_seq(u64 c) {
  u64 sum = 0;
  Generator g(&seq, c);
  for (; g; g.resume())
    sum += g.value();
  return sum;
}
```

(a) Sum of Sequence

```
void _H(GPromise& g, char n,
        char f, char t, char a) {
  if (n == 0) return;
  _H(g, n-1, f, a, t);
  g.yield(n+(f<<8)+(t<<16));
  _H(g, n-1, a, t, f);
}

u64 hanoi(GCTX *fp, char n) {
  GPromise g(fp);
  _H(g, n, 'a', 'b', 'c');
  return 0;
}
```

(b) Hanoi (generator recursion)

Figure 9. Implementations with in-stack generator

The first yield jumps to the caller right after the invocation, making it seem as if it comes back from a normal return. The promise object gets destructed automatically on exit of the generator, and on this occasion it overrides the generator function's return address with an actual position where the caller has executed to. It also sets the resume address as NULL, indicating termination of the generator.

Fig. 10 shows a direct comparison of compiled kernel loops of the proposed stackful generator with CACS and those of C++20 stackless generator, using the sequence's sum test case. As a result the two implementations have exactly the same number of instructions, and they show similar performance numbers (see section 4 for details).

3.3 Calling Convention for Switching Functions

The default calling convention (CC) is sub-optimal for functions that are expected to switch context, either directly or indirectly. The default CC is designed for general cases, in which some registers are expected not to be modified by the callee function. This condition is not met if the execution of callee clobbers all registers, and context switching is one of such cases.

The optimal strategy to call a switching function is to expect all registers will be clobbered, and define all registers as caller-saved, so as to avoid unnecessary saving and restoring. We have designed such a new CC named as preserve_none, meaning that the function will preserve none of the registers,

```
// generator's kernel loop
<+32>: movq %rax, -8(%rbp)
<+36>: movq -8(%rbp), %rax // ??
<+40>: leaq 5(%rip), %rdx ;<+52>
<+47>: xchgq %rbp, %rsi
<+50>: jmpq *%rdi
<+52>: movq -8(%rbp), %rax
<+56>: decq %rax
<+59>: jne 0x7d0; <+32>
```

```
// sum()'s kernel loop
<+32>: addq %rax, %rcx
<+35>: movq %rcx, -0x8(%rbp)
<+39>: leaq 5(%rip), %rdi ;<+51>
<+46>: xchgq %rbp, %rsi
<+49>: jmpq *%rdx
<+51>: movq -8(%rbp), %rcx
<+55>: testq %rdx, %rdx
<+58>: jne 0x9e0 ;<+32>
```

(a) Stackful with CACS

```
// generator's kernel loop
<+0>:  cmpb $0x0, 0x28(%rdi)
<+4>:  je 0x4d7c ;<+28>
<+6>:  movq 0x20(%rdi), %rax
<+10>: decq %rax
<+13>: je 0x4d85; <+37>
<+15>: movq %rax, 0x20(%rdi)
<+19>: movq %rax, 0x10(%rdi)
<+23>: movb $0x1, 0x28(%rdi)
<+27>: retq
```

```
// sum()'s kernel loop
<+48>: movq 0x10(%r14), %rbx
<+52>: movq %r14, %rdi
<+55>: callq *(%r14)
<+58>: addq %rbx, %r15
<+61>: cmpq $0x0, (%r14)
<+65>: jne 0x1770 ;<+48>
```

(b) Stackless with C++20

Figure 10. The disassembly code of the kernel loops of in-stack generator (a) and stackless generator (b). They have identical number of instructions. Note that there is a redundant instruction in stackful generator at <+36>, due to limitation of compiler optimization.

```
extern Coroutine* CURRENT;
#define CO \
    __attribute__ ((preserve_none))
CO void yield () {
  auto from = CURRENT;
  auto to = from->next;
   if (from == to) return;
  CURRENT = to;
  from->state = READY;
  to->state = RUNNING;
  CACS_tail(from, to);
}
```

```
<+0>:  movq 0x1799(%rip), %rsi
<+7>:  movq (%rsi), %rdi
<+10>: cmpq %rsi, %rdi
<+13>: je 0x199f ;<+47>
<+15>: movq %rdi, 0x178a(%rip)
<+22>: movl $0x0, 0x18(%rsi)
<+29>: movl $0x1, 0x18(%rdi)
<+36>: movq %rsp, 0x8(%rsi)
<+40>: movq 0x8(%rdi), %rsp
<+44>: popq %rax
<+45>: jmpq *%rax
<+47>: retq
```

Figure 11. The yield() function with `preserve_none` calling convention (left) is compiled to 12 instructions (right) .

and the callers must spill all necessary registers themselves. Applying this new CC to a switching function is either beneficial (if the caller is simple) or not worse (if the caller is complex), because the number of spilled registers in the new CC can not be greater than the total number of caller-saved registers and callee-saved registers in the default CC. Note that non-switching functions, such as `malloc`, should use the default CC as before.

The new CC acts as an expansion of CACS for the caller functions to unleash the full potential. It is important for simple switching functions. Take the yield() function in symmetric coroutine for example, which does simple scheduling (e.g. round-robin) then switch the context. It doesn't save / restore any registers by itself, but adopting CACS will make it so for all necessary registers to comply to the default CC. As a result, there would be hardly performance gain, because the cost is only teleported from context_switch() to yield().

As shown in Fig. 11, a possible yield() function with preserve_none CC could be compiled to only 12 machine instructions, containing no register saving at all. The callers will save and restore registers according to their real needs. The implementation also makes use of the fact that the switching is the tail of the function, by combining "jumping to target context" and "returning to the caller" as a single step — jmpq *%rax, without resorting to the final retq[2]. If yield() uses the default CC, its compiled code will have 12 more instructions for spilling 6 registers as required, and the tail-call optimization will become infeasible either.

Applying the new CC is simply to add a new attribute to the target function, and the syntax can be further simplified by using the macro CO, as shown in Fig 11. The burden of applying the new CC is much less than that of applying stackless coroutine, which forces to change the return types of all involving functions, as well as the syntax of every invocation. Whereas the proposed CC is only an option to further improve performance.

The new CC is also applicable to in-stack generators, because they switch context too, and the switching functions, yield() and resume() in our work, clobber all registers as well. This behavior makes preserving registers useless for them.

3.4 Implementation

We have implemented CACS for Clang/GCC compilers, x86-64 architecture, SysV (AMD64) ABI. The implementation is a library of 2 header files, respectively for symmetric and asymmetric coroutine (generator). The former is ~40 lines, including tail-call optimization; and the latter one is ~90 lines, including support for both sides of the generators. CACS doesn't require modifications to compiler tool chain. The implementation is based on Photon [10], assuming control struct of symmetric coroutine defined in it. Specifically, we store switch-back address at top of stack, and stack pointer (SP) in the control struct by an offset of 0x10.

We have implemented in-stack generator in the library too, using the core concept of CACS. The differences here are: (1) we switch stack frames instead of the whole stack; (2) a value is passed in a register along with each switching from the generator to its caller; (3) as the switching target is fixed and known at compile time for both sides, so we can pass the switch-back address to the opposite side via a register, in order to exploit a chance to avoid storing it to memory. The implementation is a header file of ~90 lines.

We have implemented the new CC `preserve_none` in Clang and its backend LLVM, primarily in 3 steps. Firstly we define the new CC in LLVM's header file CallingConv.h by adding a new enum member PreserveNone. And we define the list for its callee-saved registers in the TableGen file X86CallingConv.td, by inheriting the class CalleeSavedRegs and setting the list as CSR_NoRegs, a predefined constant in LLVM representing an empty list. Then we implement the new CC's behaviors in LLVM. Such as, extending the function getCalleeSavedRegs() in class X86RegisterInfo so that it recognizes the new CC as argument, and returns an empty list of registers for callee to save. We also update the instruction selector's function canGuaranteeTCO() in X86ISelLowering.cpp so that tail-call optimization can be applied to invocations with the new CC. Finally we extend the frontend clang to accept the new CC as an attribute of function. So that if a function is tagged as "__attribute__((preserve_none))", all invocations to that function will be using the proposed CC.

We have also implemented some helper tools in Clang/LLVM compiler tool chain to assist applying the proposed CC to large applications without modifying the source code. First, we add an option for Clang to accept a list of functions from a file. All functions that occur in the list during compilation will be implicitly applied with the proposed CC. Second, we add another option for the compiler to automatically

[2] jmpq is faster than retq in this case, because its general-purpose branch predictor has a somewhat higher hit rate than that of retq, which uses Return Stack Buffer (RSB) for branch target prediction, and it is supposed to miss in this case due to the switching of stack.

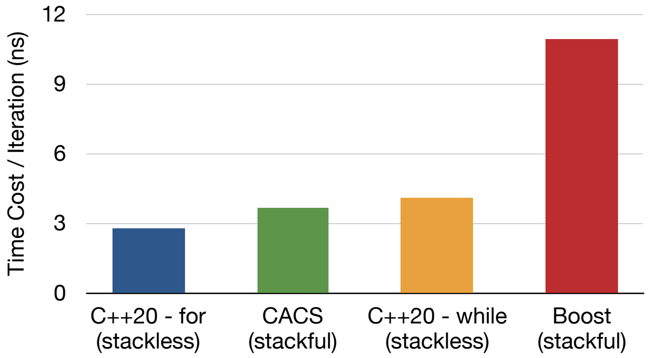


Figure 12. CACS (in-stack generator) performs equally well with C++20 stackless coroutine in sum of sequence.

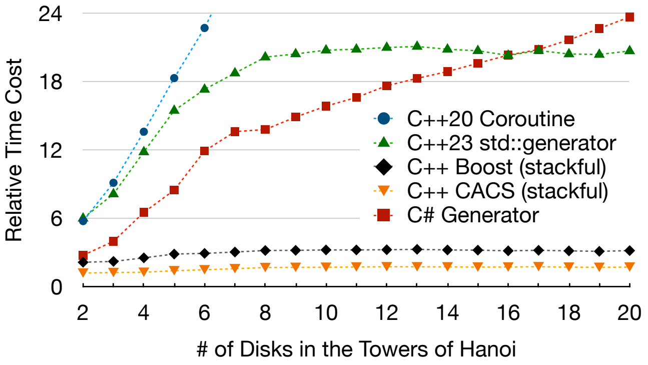


Figure 13. Time cost of coroutine recursion normalized to function recursion + callback, solving the puzzle of Hanoi. The proposed CACS (in-stack generator) performs best.

applying the preserve_none CC (APN) to proper functions. When a function `foo` with default CC calls another function with preserve_none CC, we infect `foo` as preserve_none if (1) its address is never taken, not even by a virtual function table; (2) its symbol is not exported in the final outcome (the main function is considered exported). If these functions do need preserve_none, we must mark it explicitly in the source code. APN is realized as an extra pass in LLVM, just before instruction selector (PreISel). We first create a work list including all functions that are explicitly marked as preserve_none. Then for each function in the list, we enumerate its callers to determine whether it is possible to infect the caller as preserve_none, and if so, we add it to the work list as well after infection. The process repeats until the work list becomes empty.

4 Evaluation

In this section we evaluate stackful coroutine equipped with all the proposed optimizations, denoted as CACS in a unification. Because in-stack generator is the application of CACS in asymmetric coroutine, and the preserve_none CC is an expansion of CACS for the caller functions to unleash the full potential of CACS. The CC seems to be useful only with CACS together. We also include stack address randomization in the evaluation, although it is an independent technique and applicable to other systems. We compare the results with Boost and C++20 stackless coroutines. Our testbed is a physical server with dual Intel Xeon CPU @ 2.2GHz and 128GB of memory.

4.1 Sum of Sequence

We evaluate CACS (in-stack generator) using sum-of-sequence benchmark, and compare it with other implementations. We also include a derived edition of the test case in C++20, by replacing the for-loop in producer() with a seemingly equivalent while-loop.

Fig. 12 shows the results. It is as speculated in previous section that CACS performs similarly well with C++20 coroutine. It is unexpected that the two editions in C++20 perform differently. This suggests that compiler optimization has a room for improvement, and this may also suggest that measurement at nano-second-scale is difficult, as slight changes may lead to noticeable difference.

We anticipate that CACS and in-stack generator would perform better in non-trivial applications, because its target code has less branches and memory accesses than those in stackless coroutine (4+4 vs 5+7, as shown in Fig. 10). These instructions tend to have higher miss rate in reality, due to resource contentions in TLB, branch predictor, multiple levels of caches, etc.

4.2 Hanoi

We evaluate CACS (in-stack generator) using the puzzle of Hanoi, and compare it with other implementations. Fig. 13 shows the results of relative time costs of coroutines normalized to that of function recursion + callback. CACS is roughly twice as fast as Boost, and they are both much faster than stackless coroutines. C# performs close to C++ in function recursion, so we are able to include its result as a reference. (Rust currently doesn't support yielding from recursive generator.)

And we also see with `perf` tool that CACS reduces the number of branch-misses from ~11.8M in Boost to ~1.62M in CACS, exhibiting a significant improvement, though it is still lightly worse than function recursion and callback. The reason is that the switching of context clobbers all registers on both sides, whereas the the callback can preserve some registers on the caller's side. We believe there is room for further improvement on this issue, and the optimal performance should be very close to that of function recursion.

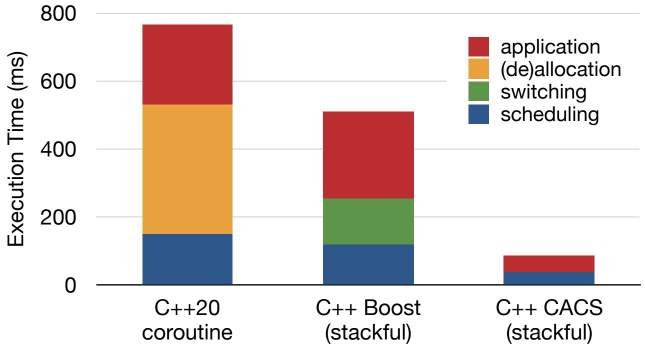


Figure 14. CACS performs better in write_fully() than C++20 coroutine and Boost.

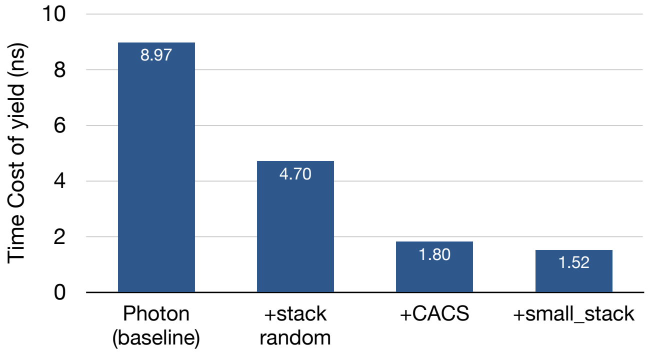


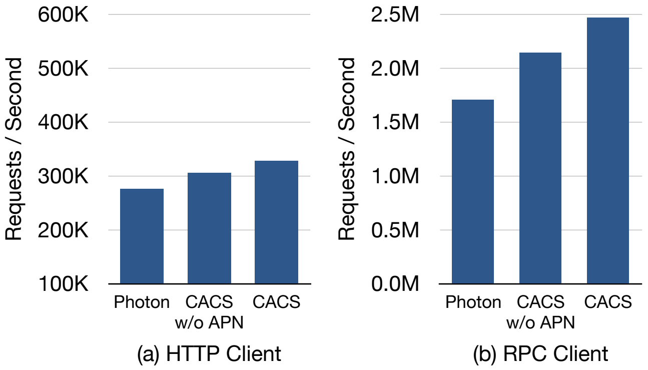
Figure 15. Breakdown of proposed optimizations in yield()

4.3 write_fully()

We evaluate CACS using write_fully() in symmetric coroutines, and compare it with other implementations. Fig. 14 shows the results. CACS is much faster than Boost and C++20 stackless coroutine. And the breakdown shows that CACS is fast in both scheduling (together with context switching) and application, we believe it is due to stack entry randomization that avoids coincidental cache collisions between the concurrent coroutines. And we confirm this speculation with `perf` tool. It shows that the miss rate of L1 data cache drops from 15.83% of Boost to ~0.00% of CACS.

4.4 yield()

We evaluate CACS with yield() operation, by making 10 coroutines yielding to one another in a loop. We repeat the loop many times and calculate the average time cost of a single yield. We then subtract the cost of an empty loop from the result, because it can not be ignored in the measurement at nano-second-scale. We have managed to avoid any compiler optimizations applying to the empty loop, and we also emulate the effect of losing all registers after calling to a preserve_none function, with a piece of volatile inline assembly code.



(a) HTTP Client

(b) RPC Client

Figure 16. Performance improvements of CACS in HTTP and RPC client respecitvely. In the cases of "CACS w/o APN" we don't enable the automatic applying of preserve_none, thus only the coroutine library has that CC explicitly. The two cases of CACS improves performance respectively by 11.2% and 18.8% for HTTP client; and 26.2% and 45.2% for RPC client.

In order to break down the optimizations, we first evaluate vanilla Photon without these optimizations. Then we apply them one by one, including a special case of small stack. We coincidentally find that stack size can influence performance, and a small one that is not aligned[3] to special numbers can often result in better performance. In this benchmark we use 4,112 bytes for the case of small stack.

Fig. 15 shows the results. The time cost of a single yield is reduced dramatically from 8.97ns to 1.52ns.

4.5 HTTP and RPC Clients

We evaluate CACS with HTTP and RPC client respectively, which can represent close-to-reality workloads. We use those components also from Photon [10], because they allow us to easily mock their networking layers so as for the results are not affected and bounded by the physical network hardware. We have managed to avoid most of the data-copy (except the header of HTTP response), effectively assuming modern offloaded zero-copy network transport. We run the benchmarks with a single CPU core, as parallelism is orthogonal to our proposals. The RPC framework avoids excessive string parsing operations, making every cycle count for the payload. It is especially useful for high-performance I/O systems. The test cases are both sending requests to obtain 1MB of data in each response. The RPC messages are real ones used in our distributed storage system, deployed at scale in our production environments. As shown in Fig. 16, CACS clearly improves real-world HTTP and RPC clients.

[3] but required to be 16-byte-aligned

5 Related Work

Coroutine is a form of generalized subroutine that dates back to decades ago. There are two types of coroutine: stackful and stackless. The former is also known as user-space thread. It assigns a separate stack to every coroutine. Whereas the latter makes the default stack shared by all coroutines. It realizes so usually by transforming the code into event-driven form at compile time. Both coroutine types depend on underlaying asynchronous interface to achieve the illusion of synchronous I/O (storage, network, etc.). There has been a long-standing debate over which model is better for concurrent programming, events or threads. The revival of coroutine in recent years may give us a different view on this problem.

5.1 Stackful Coroutine and User-Space Threading

One of the most influential stackful coroutine implementation is perhaps the cooperatively scheduled threads and processes in Windows 3.x, which dominated PC market along with the success of Windows. This form of threading was easier to implement and more efficient to run in resource-constrained devices, like PCs at that time. However, the model requires each thread / process (coroutine) to cooperate with others by voluntarily yielding its control of CPU to others at appropriate occasions. Otherwise a buggy (non-cooperative) application can block the entire system from responding. So this cooperative threading got deprecated since Windows 95 in preference of preemptive threading. As a matter of fact, the non-cooperation problem is tolerable within a single process, and concurrent programming nowadays is becoming increasingly important, thus we have the revival of stackful coroutine (stackless too), with Go [15] being a representative.

Go provides coroutine (goroutine) as a first-class construct since its first language design. Goroutine is Go's answer to concurrent programming. It supported only one CPU core (no parallelism) at first, and multiple cores later. It was cooperatively scheduled at first, and strengthened recently with preemption [7]. C/C++ alway have numerous libraries of coroutine. Such as state-thread, originally developed by Netscape as part of NSPR library, and later maintained by SGI, Yahoo. Boost.Coroutine 1, 2 and Boost.Context [13] are a set of authoritative coroutine libraries for modern C++. There are many other languages also have (some sort of) support for stackful coroutine, such as Java (since v19) [8], Lua [17], PHP (swoole) [19], etc.

While most coroutines are cooperatively scheduled, which requires a task explicitly yields its control of CPU so that one of the others can get executed. There are work to add preemption to coroutines. Most proposals [22, 25, 36] are based on timer signal, and this technique has been applied in Go since 1.14 [7]. This approach is limited due to the fact that, some functions are not safe to get preempted by signals, such as malloc(). [43] proposed another approach called KLT-switching to overcome this issue.

Stackful coroutine assigns a separate stack to every coroutine. The stack must be large enough for maximal possible usage, so it may be a waste of memory. Go initially uses segmented stack to save stack memory. New stack segments are automatically allocated as function invocations need. They are linked for ease of management and traverse. Go switched to continuous stack that can grow and shrink after V1.3. The growing is realized with memory copy, thus incurs some overhead, and is incompatible with C/C++, because it's impossible to update references to stack-allocated objects. Some C/C++ compilers, such as GCC [14] and Clang [4], also support segmented stack, but this feature have yet to become the mainstream. We want to solve this problem with the help of modern hardware. As modern 64-bit address-space is big enough for the allocation of many large continuous stacks, we can make use of madvise() to request the OS kernel to release unused physical memory pages in the stacks, so that they consume only address space. When the space is accessed again in the future, OS kernel will implicitly allocate new pages via page fault trap to meet the needs. We believe this is a sweet point for the trade off between memory consumption, execution efficiency and solution compatibility. And we call for kernel support for more efficient reclamation of the pages.

Ref. [27] shares a common goal with CACS to minimize the number of spilled registers when switching context. It proposed a new primitive in compiler backend (LLVM) specifically for context switching called `SwapStack`. It also managed to expose the primitive to users throughout the whole compiling pipeline. On the other hand, CACS constitutes of a few functions in templatized inline assembly code of tens of lines as (part of) a library. And it's only a matter of syntax adjustment for CACS to support other compilers (e.g. clang, gcc, cl, etc.) and platforms (e.g. Linux, Windows, macOS, etc.). In addition, we propose in-stack generator for efficient asymmetric coroutine, which was not addressed in [27]. And we also propose preserve_none calling convention for further improvements of all switching functions. It is especially important to simple functions, as shown in Fig. 11, the yield() function would have been compiled to twice more instructions (24 vs 12), if not with preserve_none.

This paper focuses on performance measurement and optimizations of coroutine, specifically for stackful coroutine in compiled languages. Our work should be applicable for JIT compilation of high-level languages. And we believe it is also enlightening for dynamically typed languages.

5.2 Stackless Coroutine and Event-Driven Paradigm

Stackless coroutine in compiled languages is translated into event-driven code during compilation, and there exists some work [23, 24, 29, 31, 34, 42] that does similar (source-to-source) translation before the advent of stackless coroutine in these languages. They usually provide new keyword(s) that act(s) like `co_await` in C++20 stackless coroutine, such as `tamed` in [31] and `async` in [29].

The translations are introduced for dealing with the so-called "stacking-ripping" [21] problem that arises in asynchronous even-driven code. They proposed to automatically rip the stacks with (pre)compilers instead of doing it manually, as it is complex and error-prone. Stackless coroutine can be regarded as the lastest evolution of compiler translation in order to get rid of this problem.

Stackful coroutines internally deals with events, too. Take Photon [10] for example, it has several event engines to interoperate with asynchronous & event-driven APIs provided by host OS kernel, such as libaio or io_uring, etc. When an event occurs, Photon switches to the handling coroutine by loading its stack address and jumping to its execution address. This is very similar to invoking a callback function in event-driven paradigm, in the perspective of machines. The primary difference is that we load the address of context into rsp (effectively r7) instead of rdi (effectively r5; the 1st argument by convention).

5.3 Others

It has been proved in [32] that events and threads are duals, though, the debate [21, 41, 44–46] for the better one still seems endless. Today's debate between stackless and stackful coroutines is essentially a continuation of the former one. We believe both events and threads have their best-fit scenarios. In addition to the optimizations for stackful coroutine, Photon also supports scheduling of stackless coroutine in conjunction with stackful coroutine, and it provides an optimal memory pool for efficient allocation of contexts in stackless coroutine.

Minimizing the number of registers to spill during context switching is also a major goal in preemptive threading, such as [33]. It is completely different compared to CACS, due to their preemptive nature.

6 Limitations and Future Work

In this paper, we propose efficient techniques to make stackful coroutine fast. Although we have demonstrated promising results, there exists limitations that need further improvements.

We call for compiler / language support for in-stack generator. In section 3.2 we have proprosed in-stack generator that makes efficient use of registers to pass information forth-and-back. When it comes to generator recursion, we have to resort to shared memory for passing information about the promise object between the frames of recursion. The object actually has a small size of 16 bytes, and it's better to share it in a pair of registers. If the users create more than one concurrent generator, it's better for the compiler to detect it and create additional stack(s) as needed. Compilers can further inline a whole generator if proper conditions are met.

We call for compiler and runtime support for coroutine-local variables, in correspondence to thread-local variables. And for an optimization that automatically applies the proposed calling convention to suitable functions.

We call for OS kernel support for more efficient page reclamation. Stackful coroutine depends on madvise() to request the OS kernel to reclaim unused physical memory pages in the stacks (but not releasing the virtual addresses). This incurs an overhead to flush involved TLB entries in all threads of current process. And the overhead increases with the number of threads / CPU cores increasing. There is possibility for more efficient flushing policy. For example, current thread can flush TLB immediately in the madvise() syscall, whereas others try to postpone the flushing until later they enter the kernel themselves. There would be an inconsistent view of the process of freeing the unused pages. This should be tolerable in many scenarios. And the reclamation should take place in an order from the coldest pages to the hottest pages.

We call for architectural support for explicit management of Return Stack Buffer (RSB). Modern CPUs generally incorporate an implicit RSB to predict target addresses of return instructions, as a special form of branch predictor. RSB is supposed to be, after a context switching, entirely invalid and inferior to the general predictor in this case. That's why classic context switching functions prefer the pop-and-jmp instruction pair to the usual ret. If it's possible to directly clear RSB, or better switch RSB as part of the context, that would be very helpful, not only for user-space context switching, but also for kernel-space security. There have been reports of vulnerabilities regarding RSB, and the kernel developers tried very hard to effectively clear RSB [26], at the cost of an "impressive" amount of memory.

This work is carried out on the x86-64 architecture with SysV (AMD64) ABI, which is our primary environment. We expect higher speedups for other ABI (e.g. Microsoft x64 ABI) or architecture (e.g. AArch64), as they have more registers to spill during a classic context switching.

7 Conclusion

In this paper we carry out in-depth measurement on coroutine implementations, finding out key issues regarding performance. And we propose effective optimizations for stackful coroutine, making its weak cases no longer weak, and strong cases even stronger. We also suggest some future work for possible further improvement. We implement the proposed optimizations in Photon and Clang, respectively. The source code is open-sourced on GitHub for blinded review. We'll try to merge our work to corresponding upstreams after review.

References

[1] Alibaba Dragonwell8 Extended Edition Release Notes. https://github.com/dragonwell-project/dragonwell8/wiki/Alibaba-Dragonwell8-Extended-Edition-Release-Notes, Accessed: 2023-11-28.

[2] Announcing Rust 1.39.0. https://blog.rust-lang.org/2019/11/07/Rust-1.39.0.html, Accessed: 2023-11-28.

[3] C++20 standard. https://isocpp.org/std/the-standard, Accessed: 2023-11-28.

[4] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, Accessed: 2023-11-28.

[5] Clobbers and Scratch Registers (in inline assembly code template). https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Clobbers-and-Scratch-Registers, Accessed: 2023-11-28.

[6] Data Plane Development Kit. https://www.dpdk.org/, Accessed: 2023-11-28.

[7] Go 1.14 Release Notes. https://golang.org/doc/go1.14, Accessed: 2023-11-28.

[8] Java 21 Release Notes. https://www.oracle.com/java/technologies/javase/21-relnote-issues.html, Accessed: 2023-11-28.

[9] JavaScript ES2017 Specification. https://262.ecma-international.org/8.0/, Accessed: 2023-11-28.

[10] Photon libOS. https://photonlibos.github.io/, Accessed: 2023-11-28.

[11] Storage Performance Development Kit. https://spdk.io/, Accessed: 2023-11-28.

[12] Swift 5.5 Release Notes. https://www.swift.org/blog/swift-5.5-released/, Accessed: 2023-11-28.

[13] The Boost libraries. https://www.boost.org/users/history/version_1_81_0.htm/, Accessed: 2023-11-28.

[14] the GNU Compiler Collection. https://gcc.gnu.org/, Accessed: 2023-11-28.

[15] The Go Programming Language. https://go.dev/, Accessed: 2023-11-28.

[16] The history of C#. https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history, Accessed: 2023-11-28.

[17] The Lua programming language. https://www.lua.org/, Accessed: 2023-11-28.

[18] The micro benchmarks once used by C++ committee to demonstrate the advantage of stackless coroutine over stackful coroutine. https://wandbox.org/permlink/J2xY7U4Hf6rryeCr, Accessed: 2023-11-28.

[19] The Swoole Extension of PHP. https://www.php.net/manual/en/book.swoole.php, Accessed: 2023-11-28.

[20] What's new in Python 3.5. https://docs.python.org/3/whatsnew/3.5.html, Accessed: 2023-11-28.

[21] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[22] Aravindh Anantaraman, A Mahmoud, Ravi Venkatesan, Yifan Zhu, and Frank Mueller. Edf-dvs scheduling on the ibm embedded powerpc 405lp. In *Proceedings of the IBM P= ac2 Conference*. Citeseer, 2004.

[23] Alexander Bernauer and Kay Römer. A comprehensive compiler-assisted thread abstraction for resource-constrained systems. In *Proceedings of the 12th international conference on Information processing in sensor networks*, pages 167–178, 2013.

[24] Alexander Bernauer, Kay Römer, Silvia Santini, and Junyan Ma. Threads2events: An automatic code generation approach. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, pages 1–5, 2010.

[25] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477, 2020.

[26] Jonathan Corbet. Stuffing the return stack buffer. https://lwn.net/Articles/901834/, Accessed: 2023-11-28.

[27] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[28] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, et al. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 610–622. IEEE, 2020.

[29] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: composable asynchronous io for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

[30] Oliver Kowalke. fibers without scheduler. In *C++ commiittee conference*, page P0876R0, 2018-02-11.

[31] Maxwell N Krohn, Eddie Kohler, and M Frans Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, pages 87–100, 2007.

[32] Hugh C Lauer and Roger M Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[33] Zhen Lin, Lars Nyland, and Huiyang Zhou. Enabling efficient preemption for simt architectures with lightweight context switching. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908. IEEE, 2016.

[34] William P McCartney and Nigamanth Sridhar. Stackless multithreading for embedded systems. *IEEE Transactions on Computers*, 64(10):2940–2952, 2014.

[35] Memblaze. PBlaze®7 7940 Series NVMe™ SSD: PCIe 5.0, High Performance for any Workload. https://www.memblaze.com/en/product/pblaze7/658.html, Accessed: 2023-11-28.

[36] Malcolm S Mollison and James H Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 283–292. IEEE, 2013.

[37] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[38] Oliver Kowalke Nat Goodspeed. Response to "fibers under the magnifying glass". In *C++ commiittee conference*, page P0866R0, 2019-01-06.

[39] Gor Nishanov. Fibers under the magnifying glass. In *C++ commiittee conference*, page P1364R0, 2018-11-20.

[40] Gor Nishanov. Response to response to "fibers under the magnifying glass". In *C++ commiittee conference*, page P1520R0, 2019-03-08.

[41] John Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5, pages 33–131. San Diego, CA, USA, 1996.

[42] János Sallai, Miklós Maróti, and Ákos Lédeczi. A concurrency abstraction for reliable sensor network applications. In *Reliable Systems on Unreliable Networked Platforms: 12th Monterey Workshop 2005, Laguna Beach, CA, USA, September 22-24, 2005. Revised Selected Papers 12*, pages 143–160. Springer, 2007.

[43] Shumpei Shiina, Shintaro Iwasaki, Kenjiro Taura, and Pavan Balaji. Lightweight preemptive user-level threads. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 374–388, 2021.

[44] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[45] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review*, 37(5):268–281, 2003.

[46] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for scalable, well-conditioned internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18), Lake Louise,*

Canada, 2001.