Weighing Continuations for Concurrency

Kavon Farvardin

University of Chicago

MS Presentation

March 31, 2017

Continuations

```
fun fact 0 = 1
  | fact n = let
    val n_1 = fact (n-1)
    in
      n * n_1
    end
```

Continuations

```
fun fact (0, return) = return 1
  | fact (n, return) = let
        fun retK n_1 = return (n * n_1)
      in
        fact (n-1, retK)
      end
```

CPS Factorial Visualized

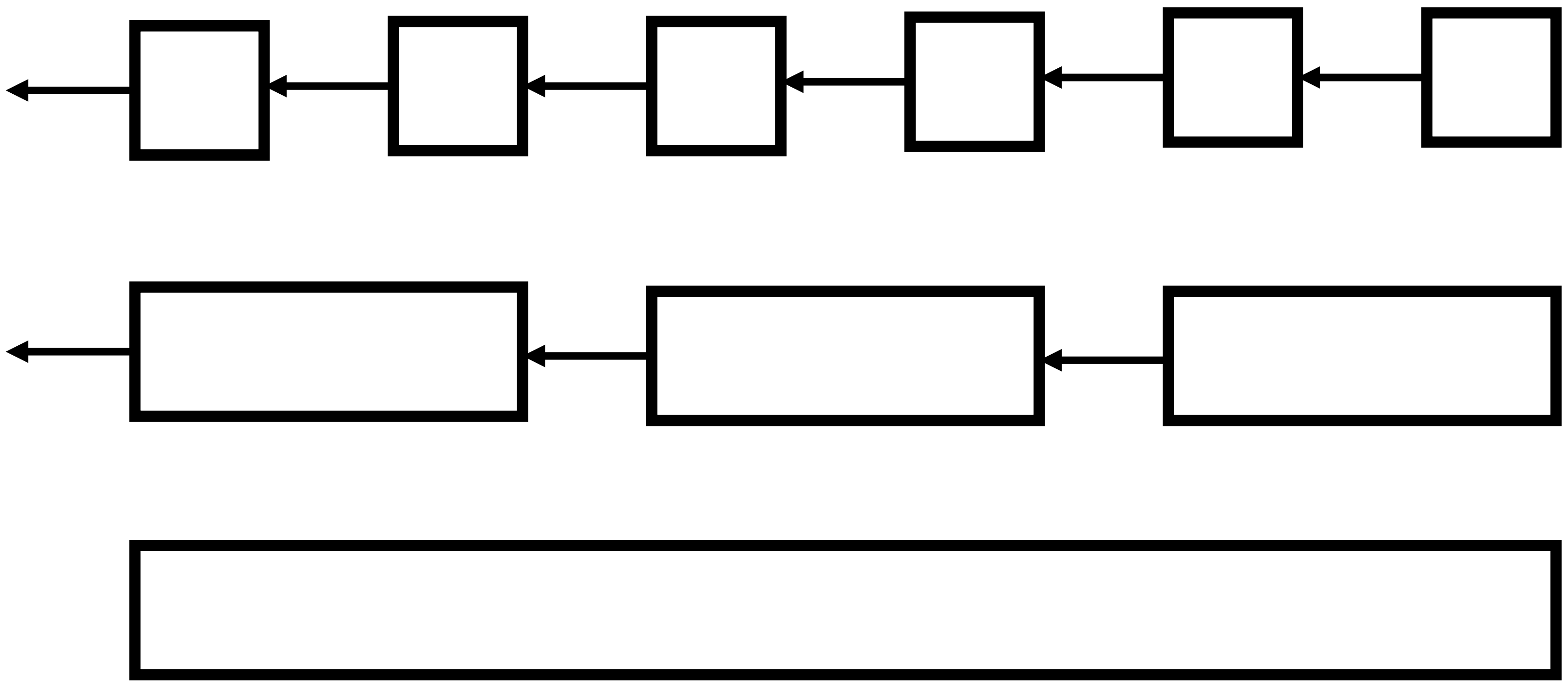
fact(3, `n = 4  | return =`)

fact(2, `n = 3  | return =`)

fact(1, `n = 2  | return =`)

fact(0, `n = 1  | return =`)

Implementation Strategies



Language support for continuations

```
type ‘a cont

callcc : (‘a cont -> ‘a) -> ‘a
throw  : ‘a cont -> ‘a -> ‘b
```

First-class Continuations

```
val ans = callcc (fn bind => let
                    fun prod (hd::tl) =
                      if hd = 0
                        then throw bind 0
                        else hd * (prod tl)
                  in
                    prod nums
                  end)
```

Continuations and Concurrency

```
type thread = unit cont

new : (unit -> unit) -> thread
schedule : thread -> unit

(* thread actions *)
exit : unit -> ‘a
yield : unit -> ‘a
```

Continuations and Concurrency

```
fun new f =
  callcc(fn ret => (
      callcc(fn thd => throw ret thd);
      f();
      exit()
    )
  )
```

Binding an Escape Continuation

```
cont k () =
  A
in
  B
```

- Binds k in the scope of “B”.
- Captures the current continuation.
- Can only be thrown once.

```
cont k () =
  return 7
in
  if cond
  then return 5
  else throw k ()
```

Implementing Continuations for Concurrency

Goals:

- Low sequential overhead.
- Extremely cheap capture and throw.
- Easy to integrate into a compiler.

Which strategy should we use?

Prior Evaluations

- Clinger et al. compared implementations of callcc
  - Direct measurements performed nearly 30 years ago
  - Callcc is expensive for concurrency
- Appel and Shao analyzed using simulation & theory
- Bruggeman et al. had a short evaluation.

Fact or Fiction?

- Rust and Go developers reported a “segment bouncing” problem.
- MultiMLton and Guile avoided segmented stacks due to the reports.
- Bruggeman et al. described a solution for the bouncing in 1996.

Are segmented stacks *actually* slow?

This Work: An Empirical Evaluation

- What are the trade-offs of various strategies for continuations?
- Implementation details are crucial.
- Strategies are implemented and measured with a *single compiler*.
- Provide an empirical analysis of trade-offs.

Implementation

Manticore and CPS

- Manticore is a compiler for parallel functional programming.
- Continuation-passing style (CPS) is used for optimization & codegen.
- Continuations are heap-allocated; made up of immutable frames.

Stack-allocating Continuations

- Manticore makes no use of a “stack”; all calls are in tail position.
- Native codegen for an efficient stack is a pain.
- LLVM already supports stack allocation; Manticore can use LLVM.
- How can a CPS-based compiler use LLVM with a stack?

Undoing CPS

Key Observation*

Most continuations created by CPS are well behaved.

* by Danvy, Kelsey, *etc.*

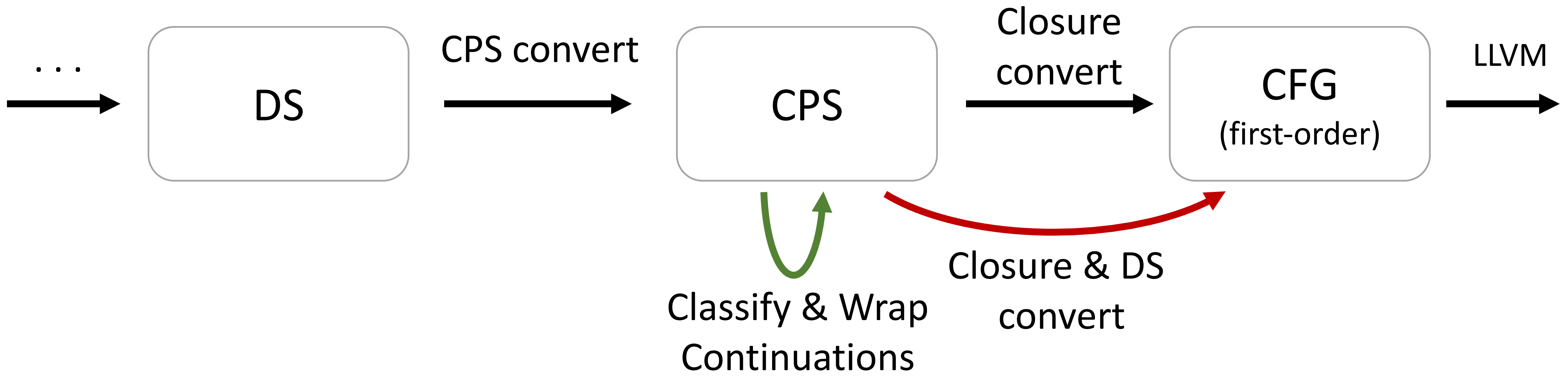
Undoing CPS

It starts with a good intermediate representation:

- Continuations and functions are different.
- Continuation parameters added by CPS are distinguished.

```
cont k () = A in B       <->  throw k ()

fun f (x, y / k) = A    <->  f (1, 2 / k')
```

Noninvasive Compiler Upgrades



Classifying Continuations

Higher-order DS

```
fun g x = x
fun f x y = if x > 10 then h((g x) + y) else h x
```

```
fun g (x / k) = throw k x            ← Return throw
fun f (x, y / k) =
  cont doH z = h (z + y / k) in
  if x > 10
    then g (x / doH)                 ← Non-tail call
    else h (x / k)                   ← Tail call
```

CPS

Return continuations are only ever used or passed from the same function.

Wrapping Escape Captures

```
fun foo(_ / retK) =
  cont k (x) =
    A
  in
    B
```

```
fun foo(_ / retK) =
  cont k’ (x) =
    A
  in
    cont landingPad(ret, x) =
      if ret
      then throw retK x
      else throw k’ x
    in
      fun bindK(k ..) = B
      in
        callec (bindK / landingPad)
```

Converting to Direct Style

Higher-order CPS

```
fun g (x / k) = throw k x
fun f (x, y / k) =
  cont doH z = h (z + y / k) in
  if x > 10
    then g (x / doH)
    else h (x / k)
```

First-order DS

```
fun g (_, x) = return x
fun f (ep, x, y) =
  block doH (ep, z, y) =
    tailcall h (z + y)
  if x > 10
    then z = call g x
           goto doH (ep, z, y)
    else tailcall h x
```

Generating Assembly with LLVM

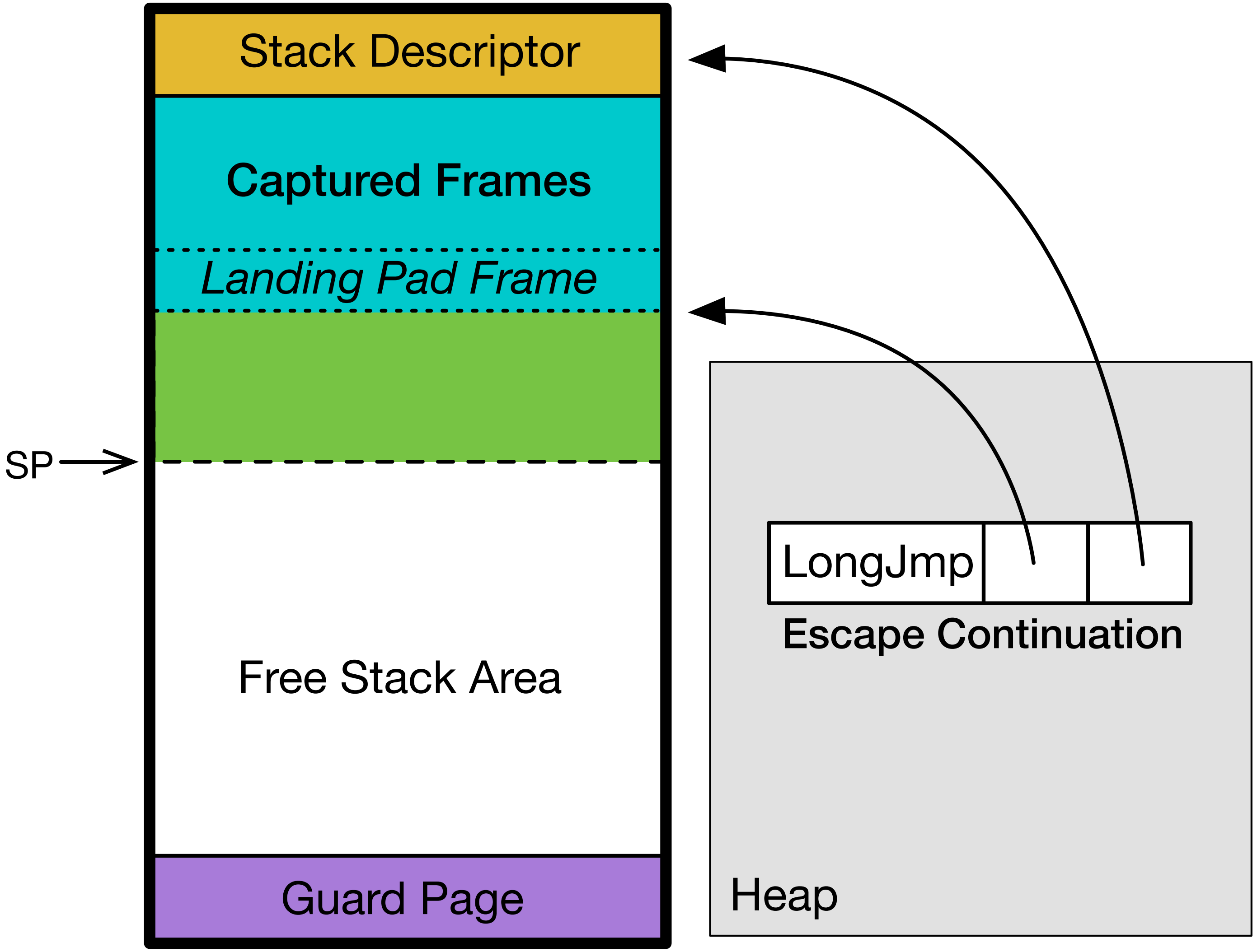
We modified LLVM to generate code for different continuation strategies.

Using LLVM, Manticore now supports:

- Contiguous Stacks
- Segmented Stacks
- Heap-allocated, Immutable Control Stacks

Contiguous Stack

(Stacks grow downwards)



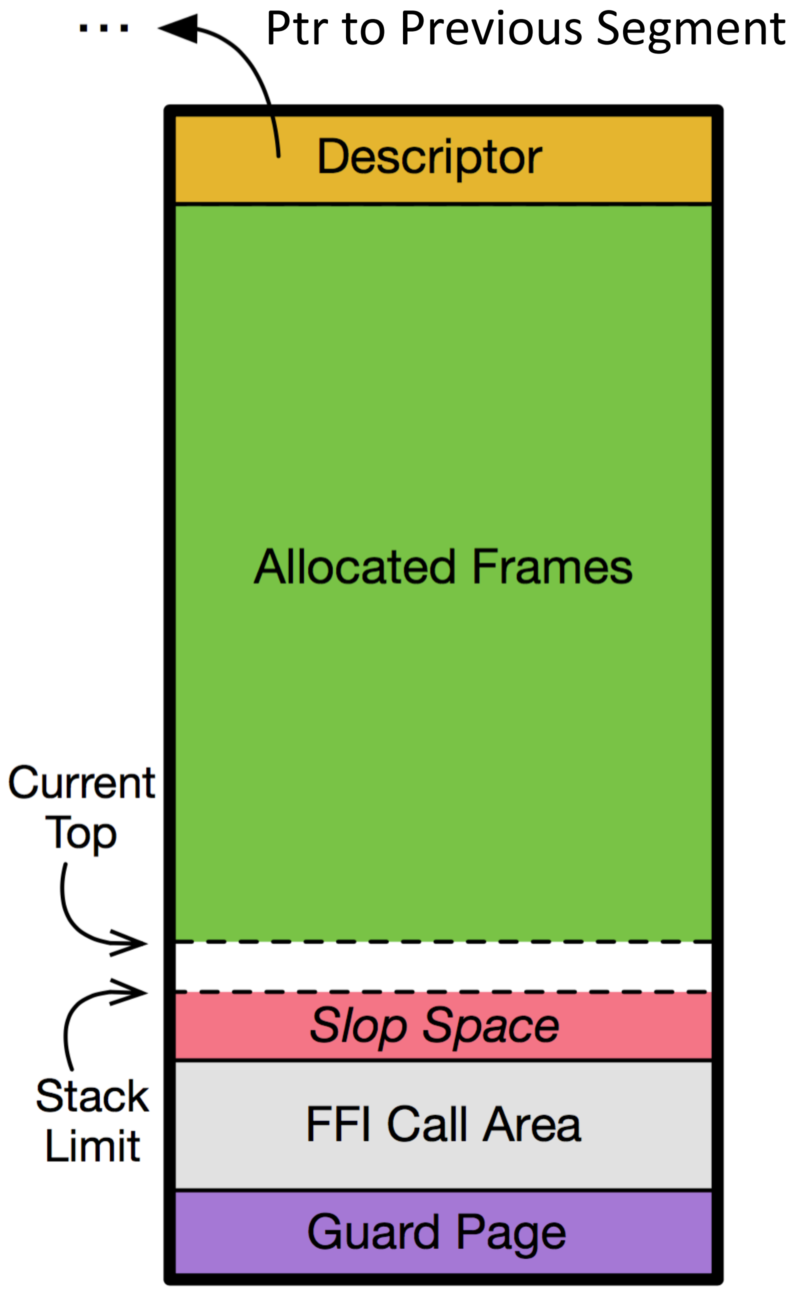
Allocating Frames on a Stack

```
_some_function:
  # prologue
  subq    $SpillSz, %rsp
  pushq   $0    # watermark

  ...

  # epilogue
  addq    $(SpillSz+8), %rsp
  retq
```

Frame Reuse for Non-tail Calls

```
# initialize return continuation for func1
movq    %rax, 24(%rsp)    # slot 3
movq    %rcx, 16(%rsp)    # slot 2
movq    %r14, 8(%rsp)     # slot 1
callq   _func1
movq    8(%rsp), %rdi   # use of val
# initialize return for func2
movq    %rax, 8(%rsp)   # reuse slot
callq   _func2
# reload live vals for use
movq    24(%rsp), %rax
movq    16(%rsp), %rcx
```

Segmented Stack

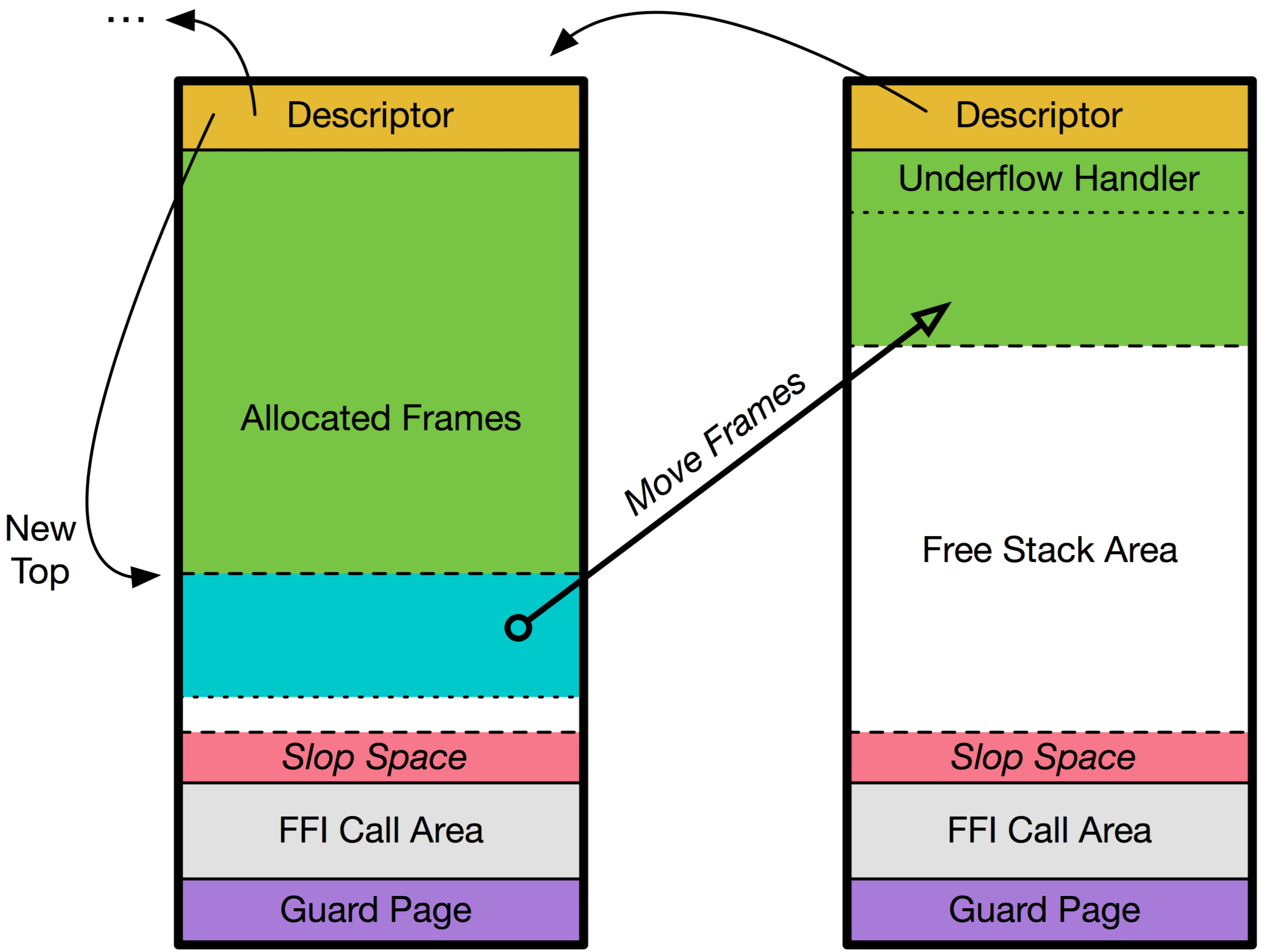


(This is an individual segment)

Segmented Stack Prologue

```
_some_function:
  cmpq    120(%r11), %rsp
  jge     allocFrame
  callq   ___manti_growstack
allocFrame:
  subq    $56, %rsp
  pushq   $72  # frame size
  pushq   $0   # watermark
  # ... body ...
```

Segment Overflow



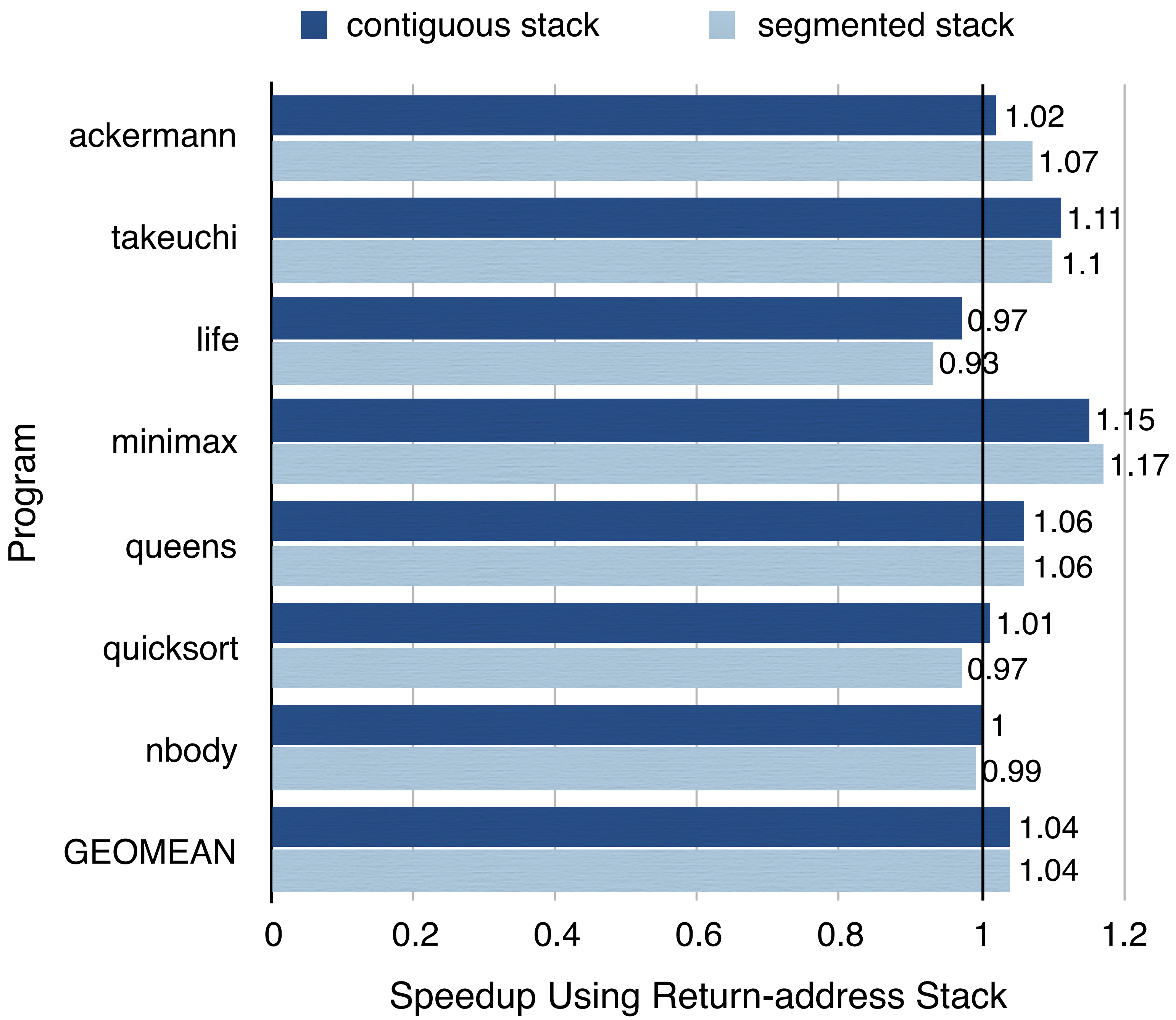
Evaluation

“The real performance cost of first class continuations is the time and money required to implement them.” – Clinger et al. (1988)

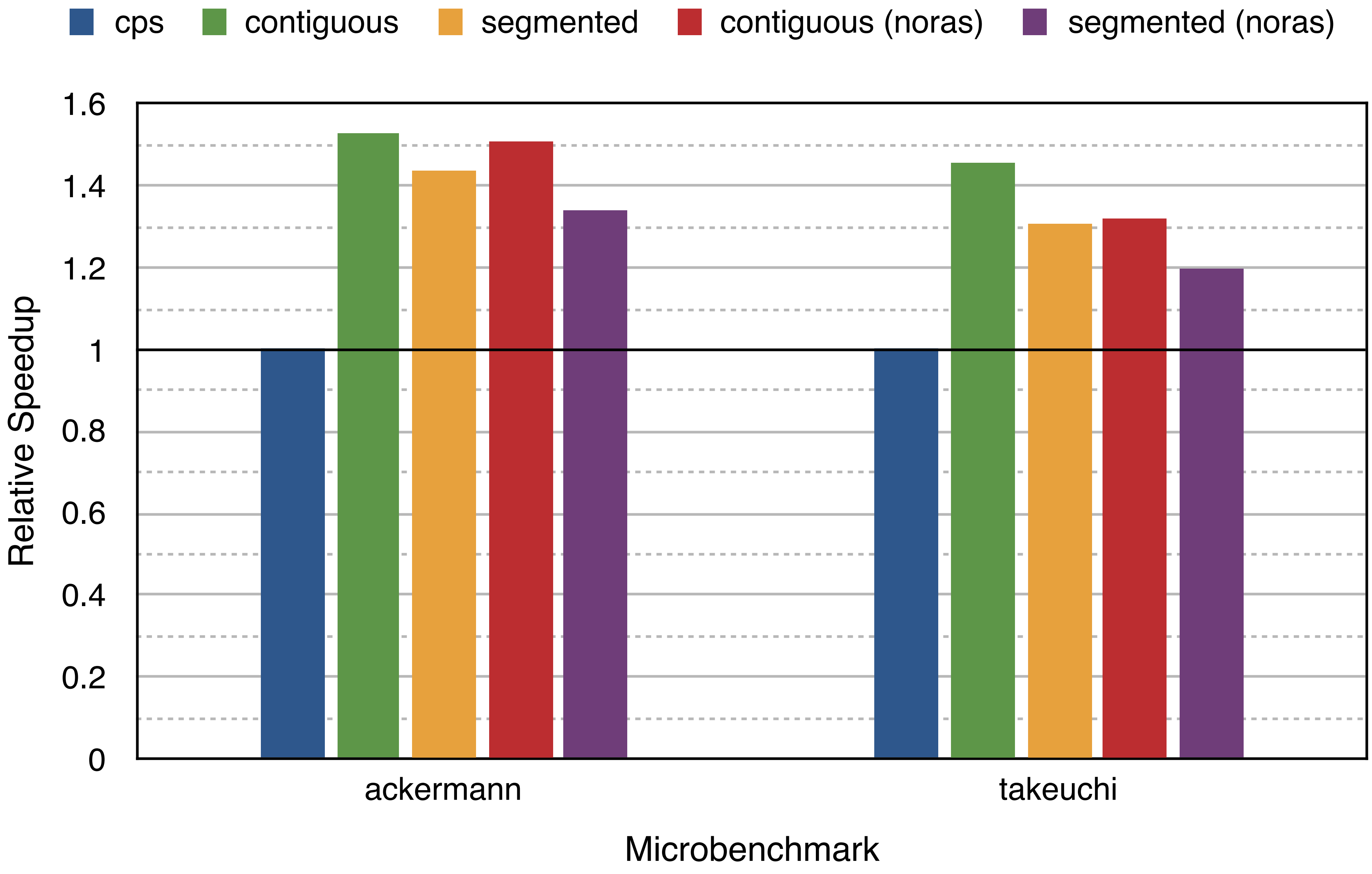
The Important Bits

- Sequential performance
- Concurrency overhead
- Friendly implementation

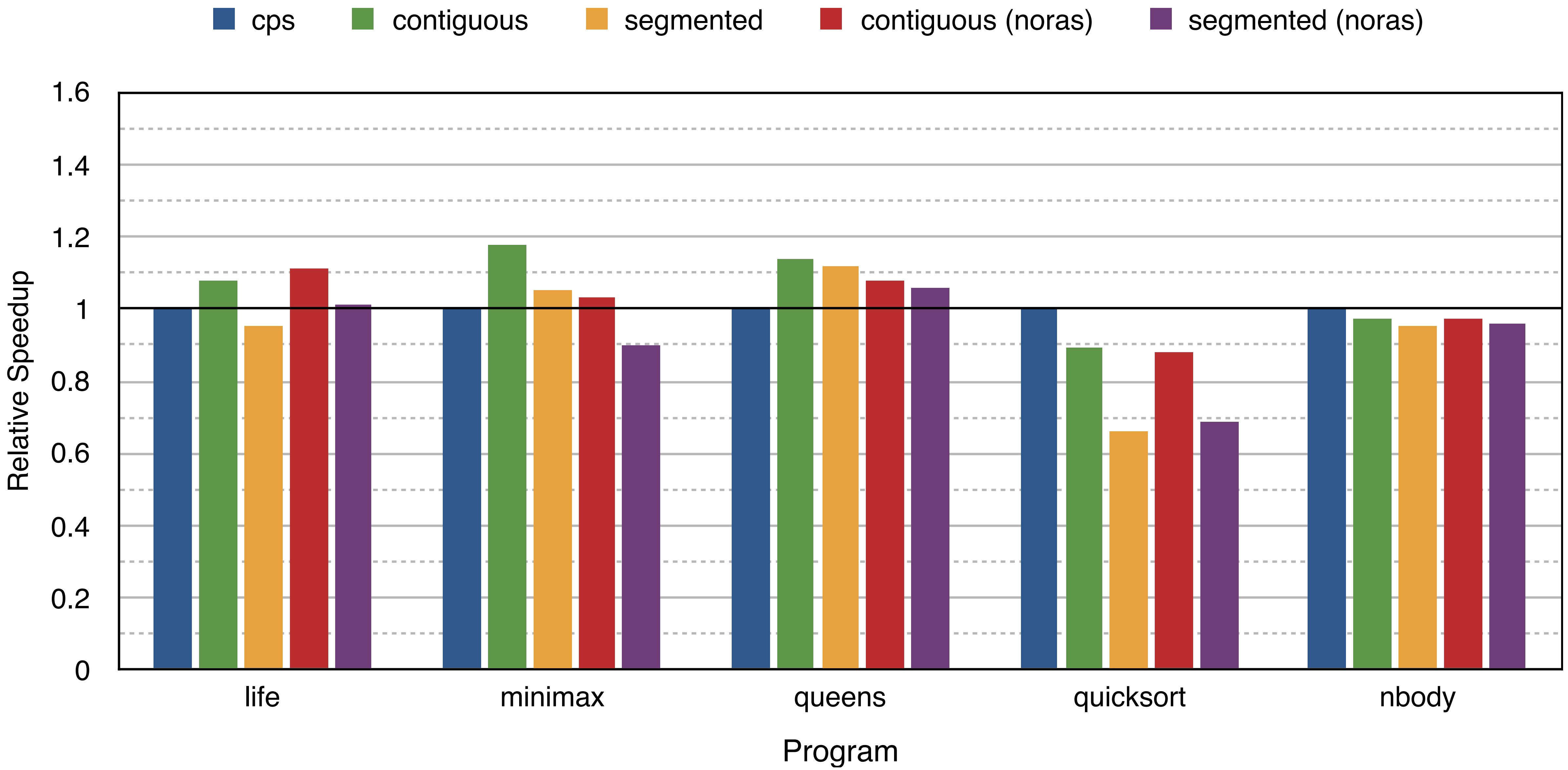
CPU Support for Stack Allocation



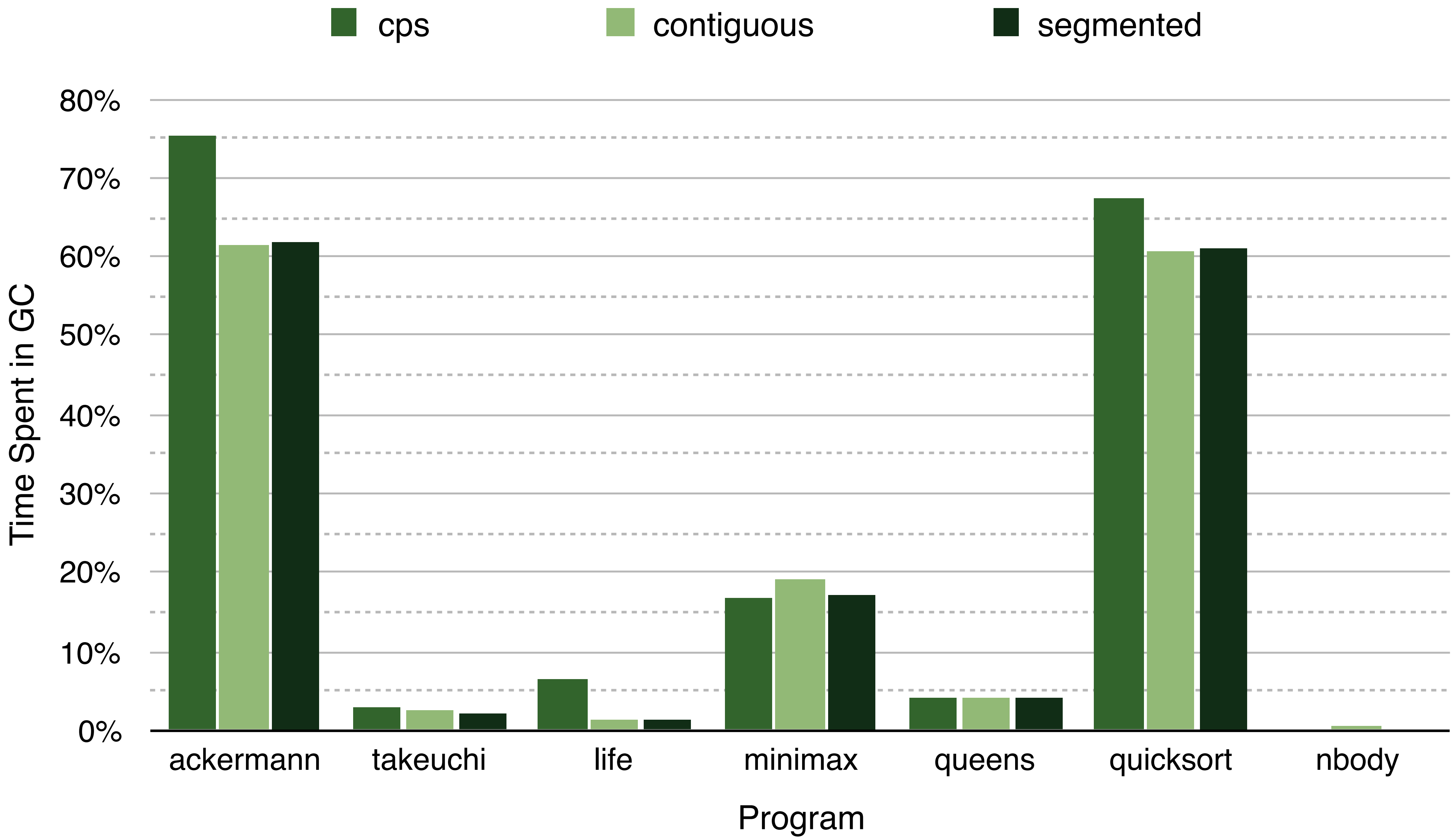
Sequential Microbenchmarks



Sequential Programs



GC time as percentage of overall running time

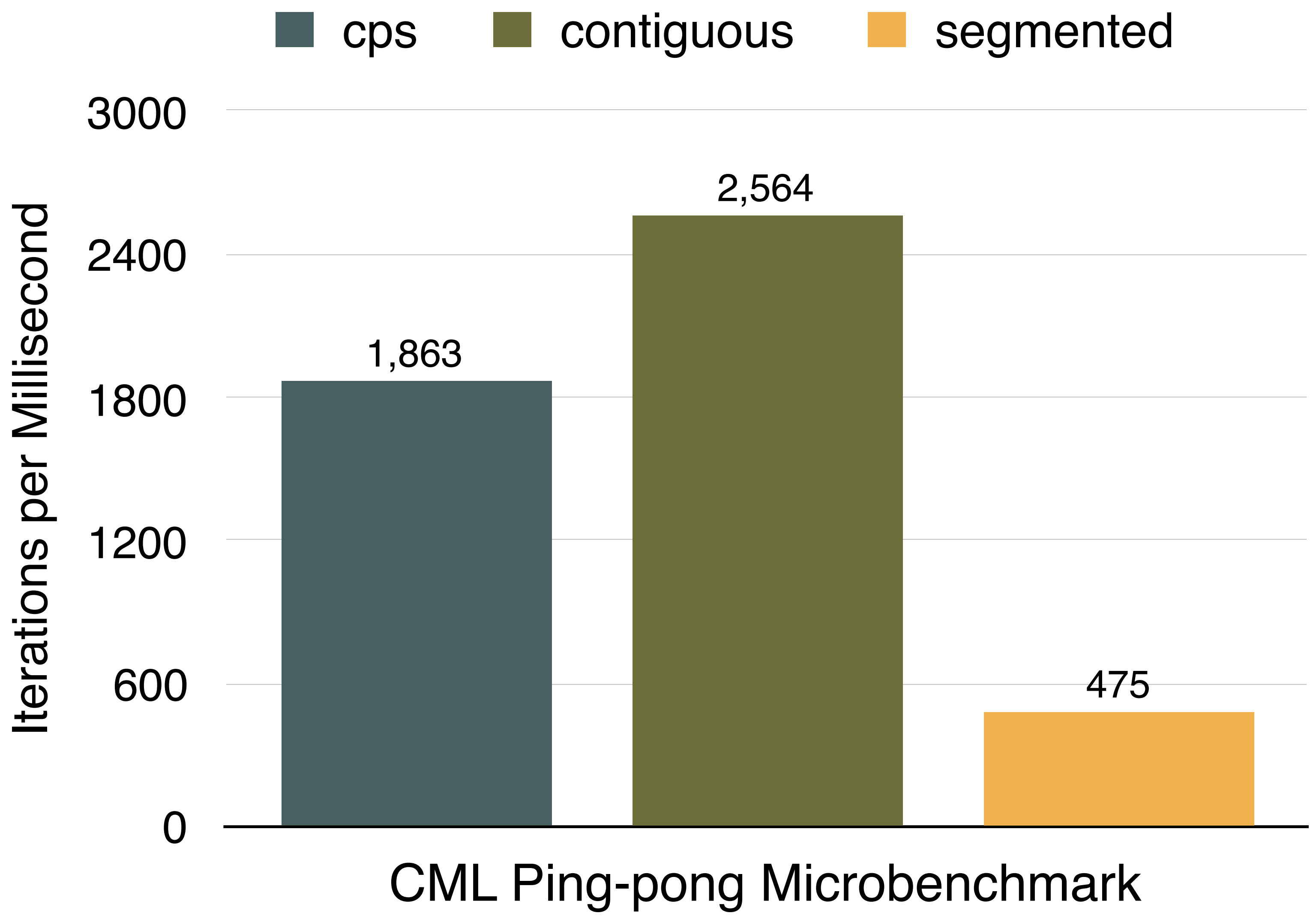


GC time as percentage of overall running time

CPS does place more load
on the garbage collector
when stacks are extremely deep.

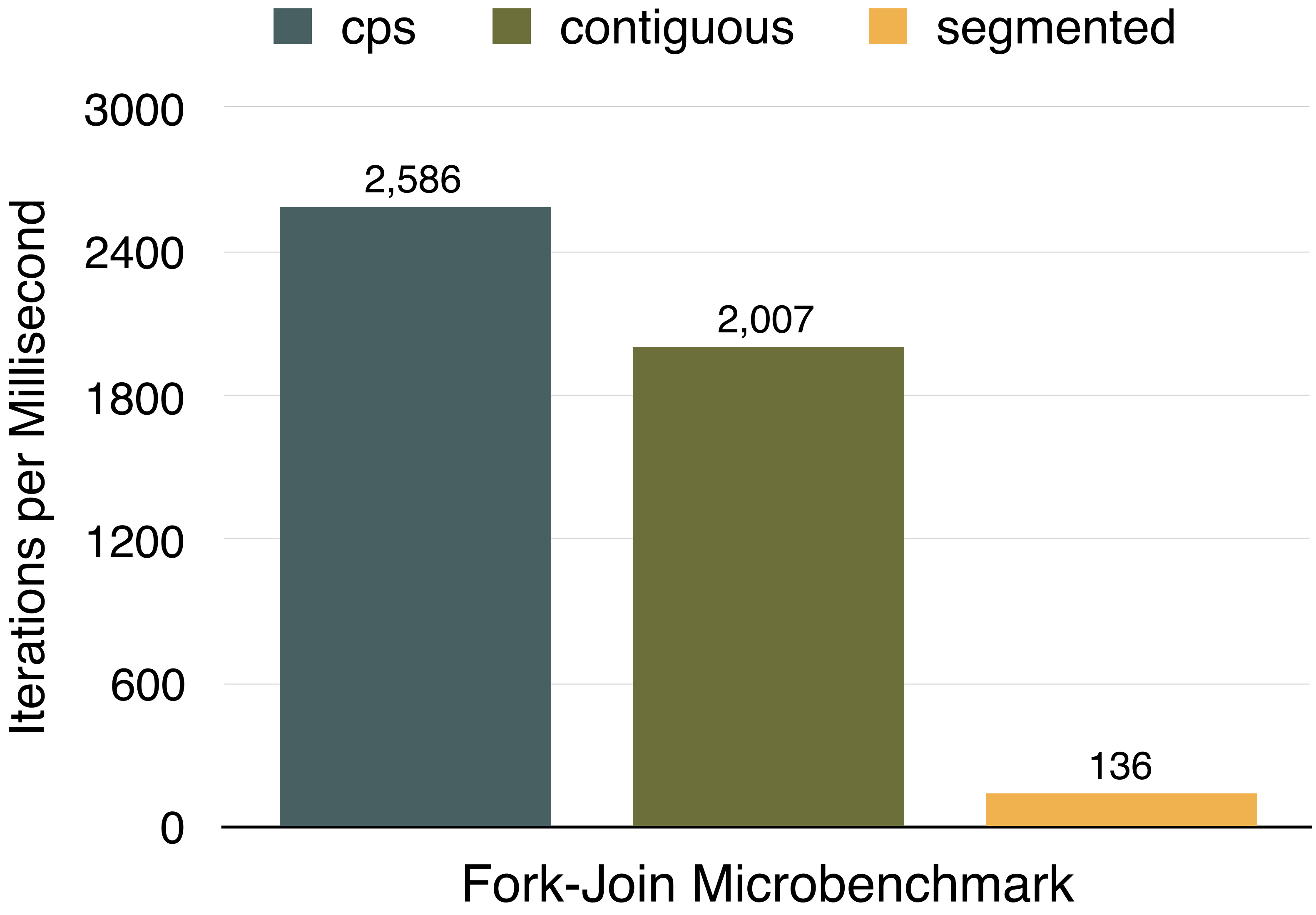
	Nursery Copies
Ack (CPS)	6242M (44%)
Quicksort (CPS)	703M (30%)
Ack (Stack)	1039M (13%)
Quicksort (Stack)	313M (20%)

CML Message Passing Overhead



Higher is better

CML Thread Creation Overhead



Higher is better

Pros and Cons

	Sequential Performance	Concurrency Overhead	Recursion Bound	Implementation Pain
CPS	🙁	😃	😃	😃
Contiguous	😃	😃	🙁	🙁
Segmented	🙂	🐢	😃	😱

Conclusions

- There is no ideal strategy.
- CPS makes sense for easy and fast concurrency.
- Contiguous stacks are faster if you sacrifice friendliness.
- Segmented stacks are hard to implement and tune.

Future Work

- Analyze additional variants of strategies.
- Expand evaluation.
- Submit for publication 😉


Questions?