The PPC64 Port of VirtualThreads

Richard Reingruber, SAP
October, 2022

THE BEST RUN SAP

Agenda

VirtualThreads Refresher

VM Continuations
- Abstract Operations
- Implementation based on StackChunks

”Freezing” / Reallocating Frames from Stack into StackChunks on Heap

“Thawing” vice versa

GC Integration

Potential Performance Cliffs

Integration with established internal APIs related to Threads and Stacks

Virtual Threads (aka Project Loom)

Lightweight user-mode threads hiding non-blocking IO

Allow for more concurrency for higher throughput

Comparatively Simple Programming Model

- facilitates Development/Maintenance

Compatible

- With existing thread APIs
- With existing tools
- Beware limitations, most notably: active synchronized blocks/methods prevent lightweight context switching

Virtual threads can significantly improve application throughput

- If the number of concurrent tasks is high (more than a few thousands)
- If the workload is not CPU-bound

https://openjdk.org/jeps/425

VM Continuations for Switching Virtual Threads (VT)

VM Support required (`-XX:+VMContinuations`)

A (VM) Continuation is a Java Object C

- Instantiated with a lambda expression λ or Runnable (just like a thread)
- λ can be executed by calling the `run()` method of C; `run()` blocks while C executes
- C can self-suspend execution by calling its `yield()` method.
  `run()` will then unblock and return to its caller while `yield()` blocks
- C can be resumed by calling the `run()` method of C again.
  The `yield()` from the suspend above will then return and execution continues.

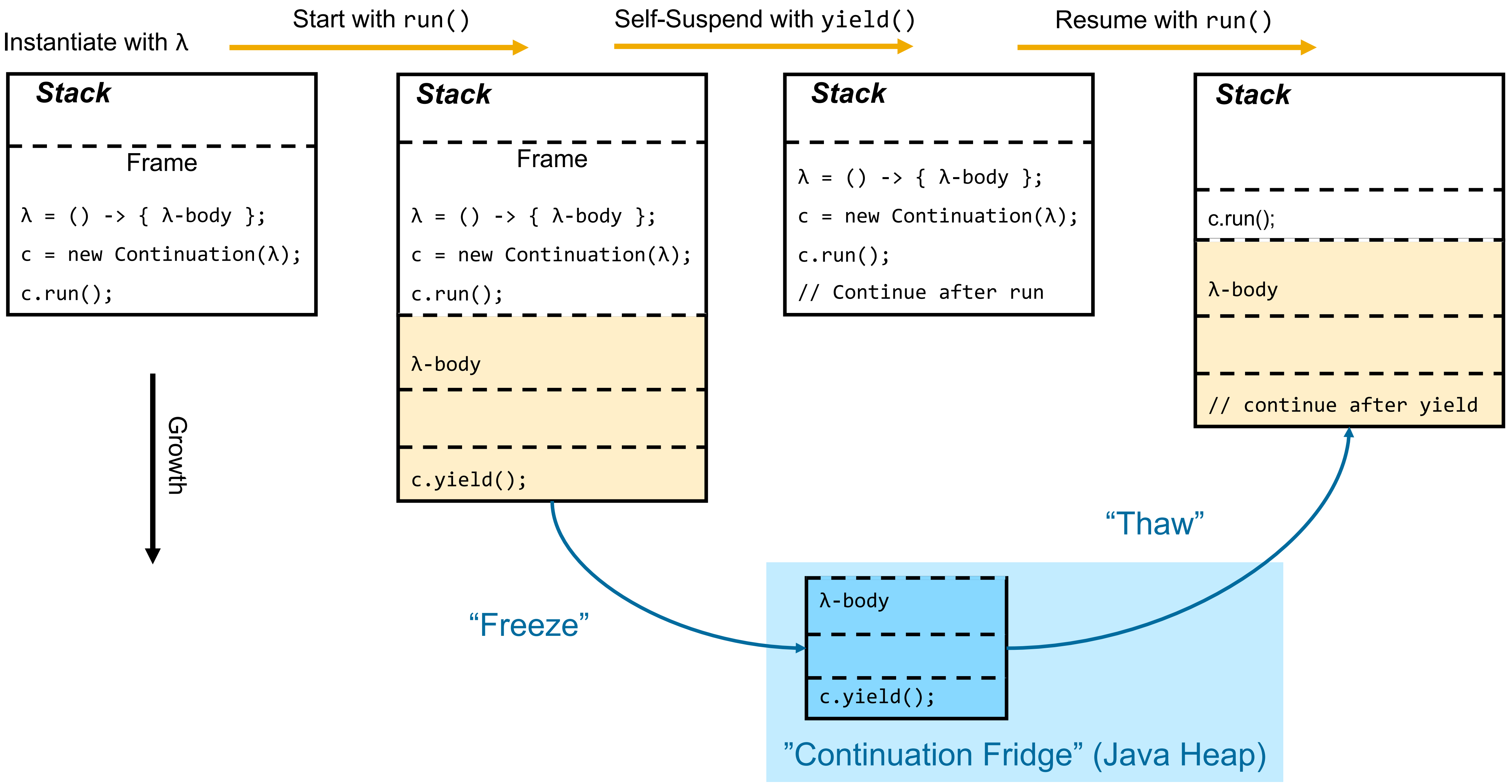
Starting a VT means creating a Continuation and running it

The VT Scheduler makes use of the resume and suspend operations to switch VTs

Threads are not preempted depending on time
but only in classlib calls that block (synchronization, IO)

Continuations are private jdk internal objects.
Without VM support (`-XX:-VMContinuations`) VT are implemented as std. heavy-weight threads

Visualization of Continuation Operations



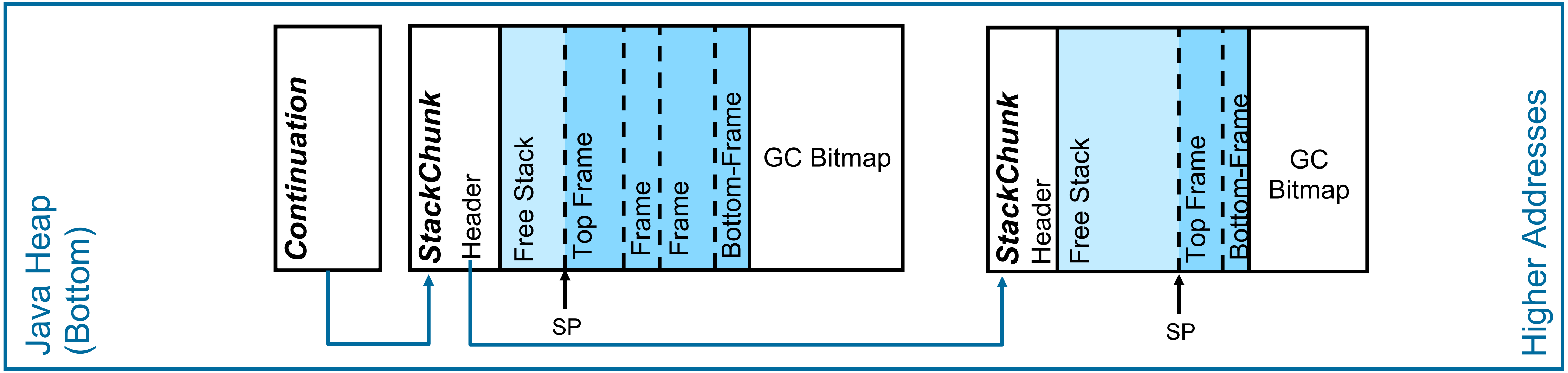
Continuations and StackChunks

`Continuation` and `StackChunk` are classes in the package `jdk.internal.vm`

A Continuation has a list of StackChunks

The StackChunks store the “frozen” frames of a Continuation on heap

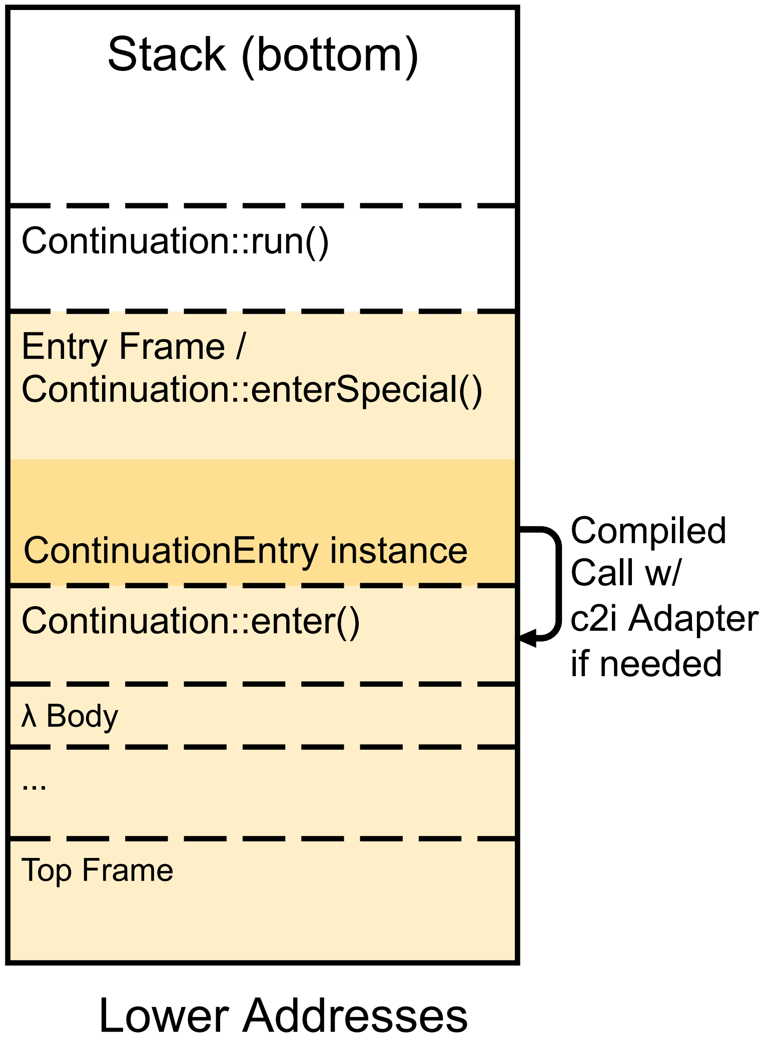
- Top frame has lowest address (just like on the stack)
- StackChunks are of variable length and require special handling during GC
- New Klass type to manage this: `InstanceStackChunkKlass`



enterSpecial Intrinsic, Entry Frame, ContinuationEntry

`j.i.vm.Continuation::enterSpecial()`

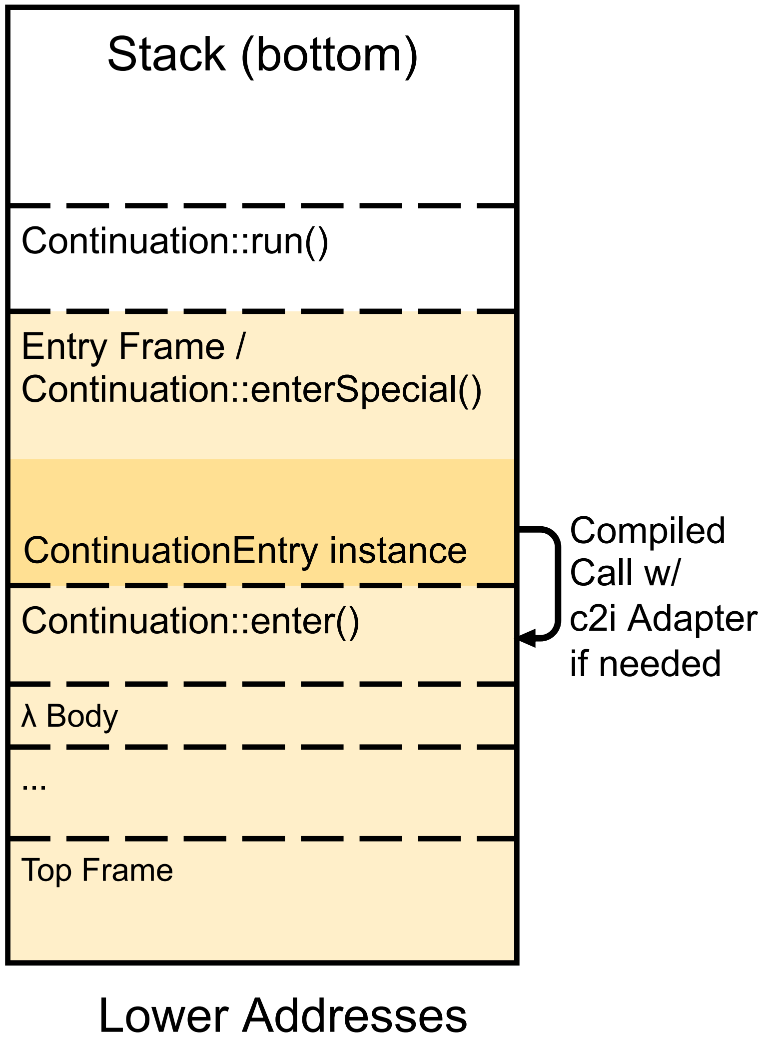
- declared as native method, implemented as a vm intrinsic
- Special native wrapper
  - nmethod that is called with the compiled (jit) calling convention
  - handwritten generated assembler
- Pushes the `Entry Frame` with the `ContinuationEntry` instance, initializes it, and prepends it to the `JavaThread`'s list of CEs
- Depending on parameter `isContinue` either
  - `false`: a static compiled call is executed. There is special handling in the runtime that binds the call to Continuation::enter() which calls the continuations λ
  - `true`: stack frames are copied ("thawed") from the heap to the stack and control is transferred to the top frame
- See [`gen_continuation_enter()`]



enterSpecial Intrinsic, Entry Frame, ContinuationEntry

`ContinuationEntry` instance

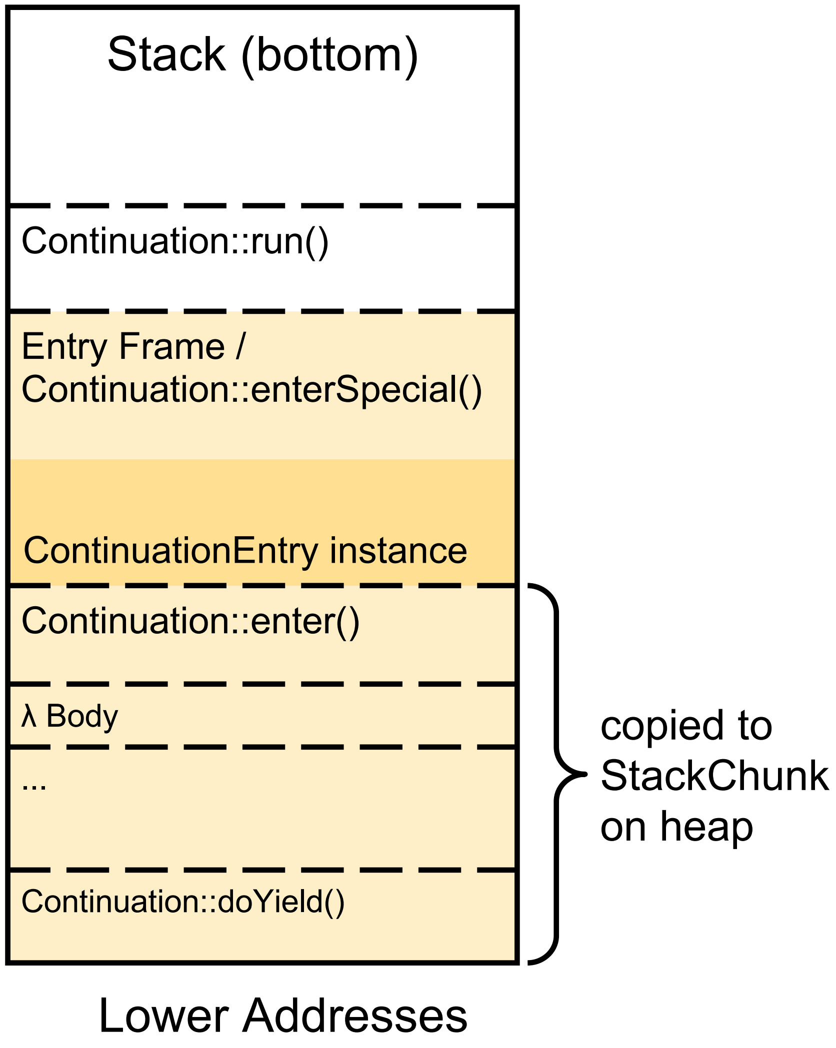
- located in the Entry Frame
- `JavaThread` has a list of its `ContinuationEntry` instances
- has a Reference to `j.i.vm.Continuation` instance
- Other continuation related data copied from the JavaThread for the parent continuation (e.g. locked monitor count)



doYield Intrinsic

`j.i.vm.Continuation::doYield()`

- declared as native method, implemented as a vm intrinsic
- Special native wrapper
  - nmethod that is called with the compiled (jit) calling convention
  - handwritten generated assembler
- Calls the runtime where the frames atop the entry frame are copied ("frozen") to a `StackChunk` instance on the heap
- Pops frozen frames by restoring the `ContinuationEntry*` from the `JavaThread`.
- Copies the parent's data in the `ContinuationEntry` back to the `JavaThrad` and removes this `CE` as list head
- Pops the entry frame and returns control to its caller
- See [gen_continuation_yield()]



Freezing Frames – Slow Path

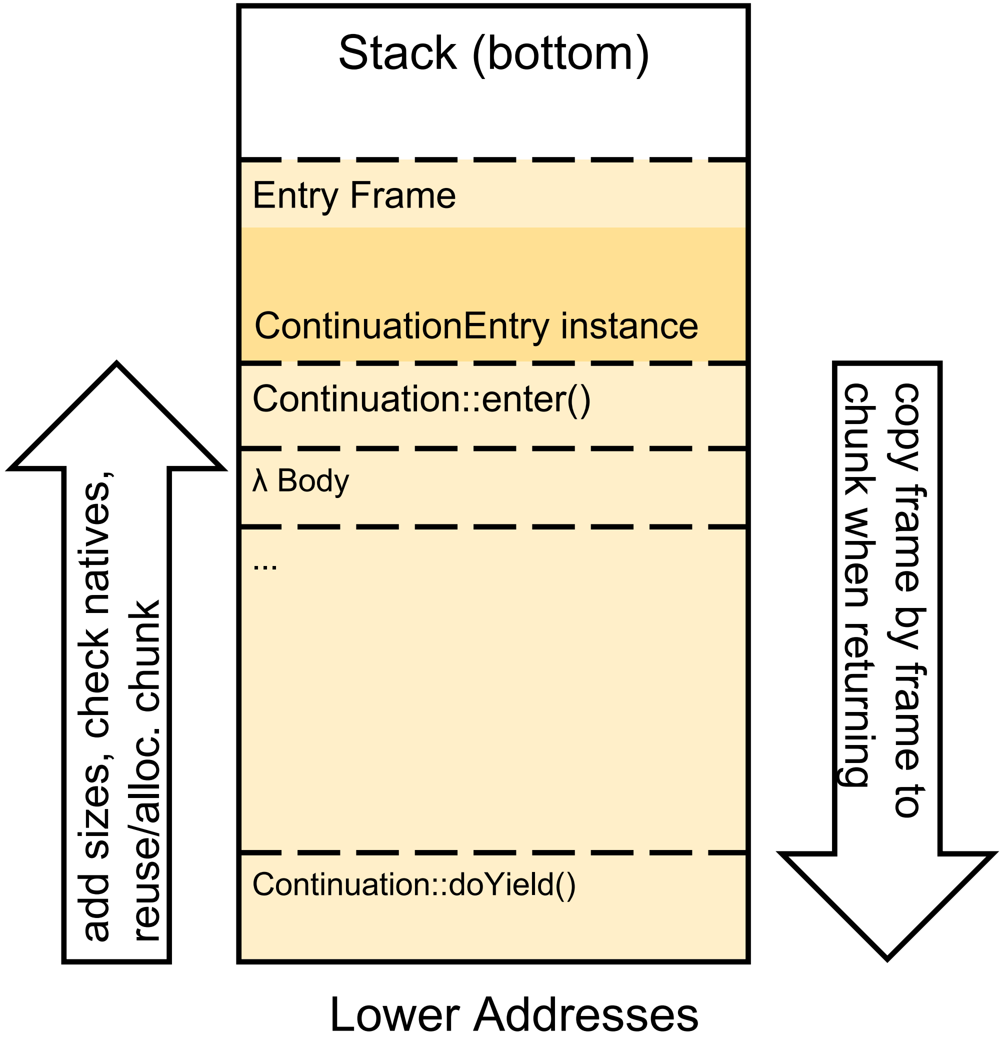
Fail if `JavaThread::_held_monitor_count > 0`

Iterate frames recursively top to bottom
- adding the sizes
- fail if native frame found

At the bottom frame
- required size S is known
- existing StackChunk can be reused
  – if S <= free in
  – if never traversed by GC
  – if it does not need special GC barriers (Shenandoah, ZGC)
- Otherwise a new StackChunk is allocated

Then Copy, frame by frame, bottom to top to StackChunk

Code: `FreezeBase::new_heap_frame()` and callers



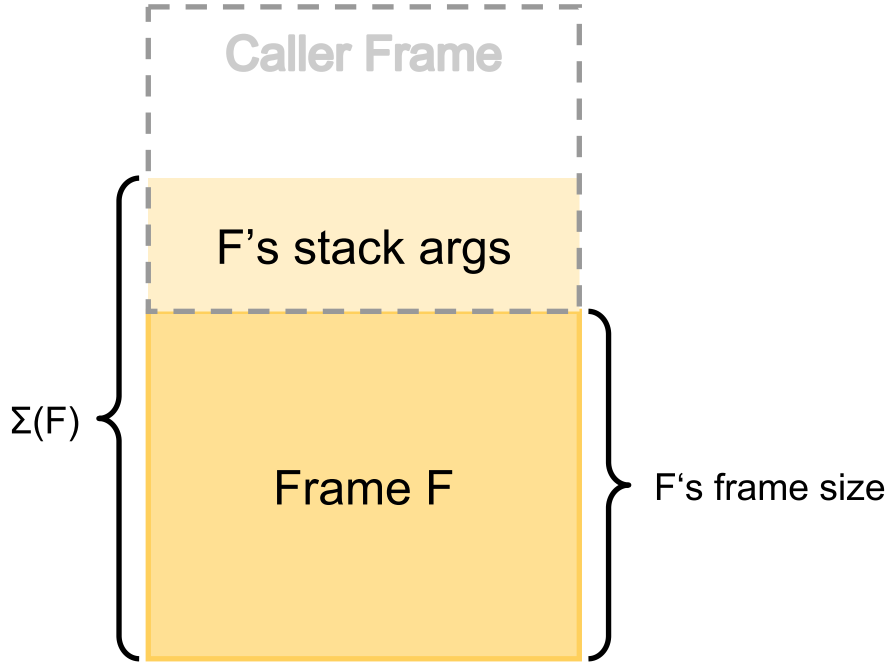
Freezing Frames – Slow Path – Sizing

Frame Size

- straight forward: decrement of the stack pointer (SP) before the body of a function is executed
- Beware details, e.g. interpreter frames can grow when stack locks are allocated

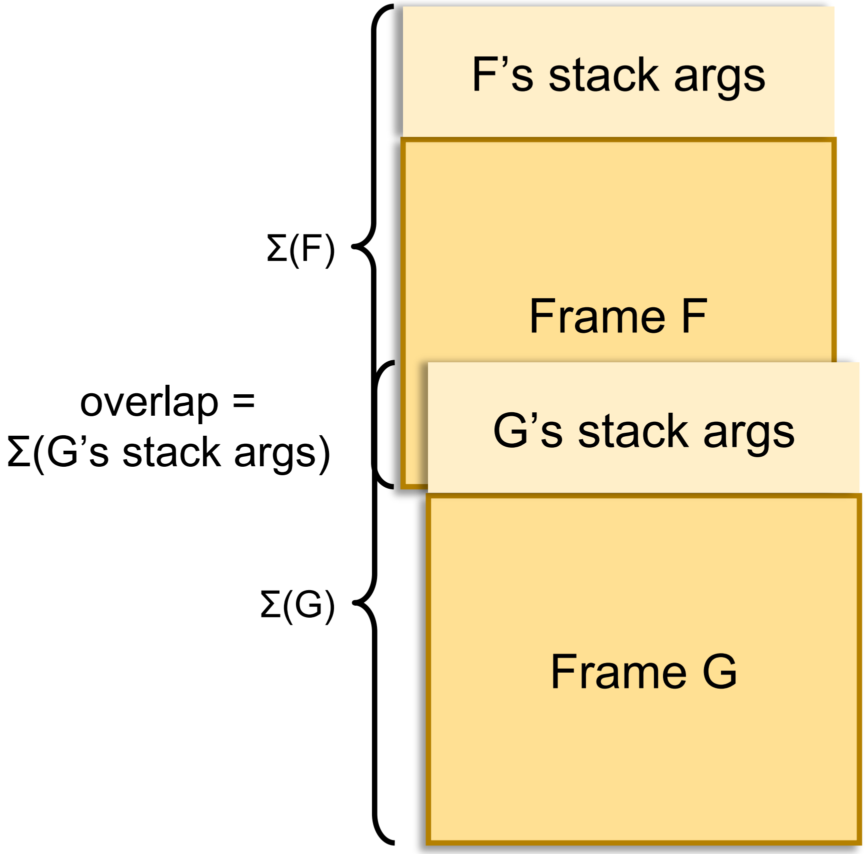
Size function Σ

- size occupied by a frame (part) in the StackChunk
- Frame: size of complete state!
  – This includes args passed on stack
  – Even though stack args reside in the caller

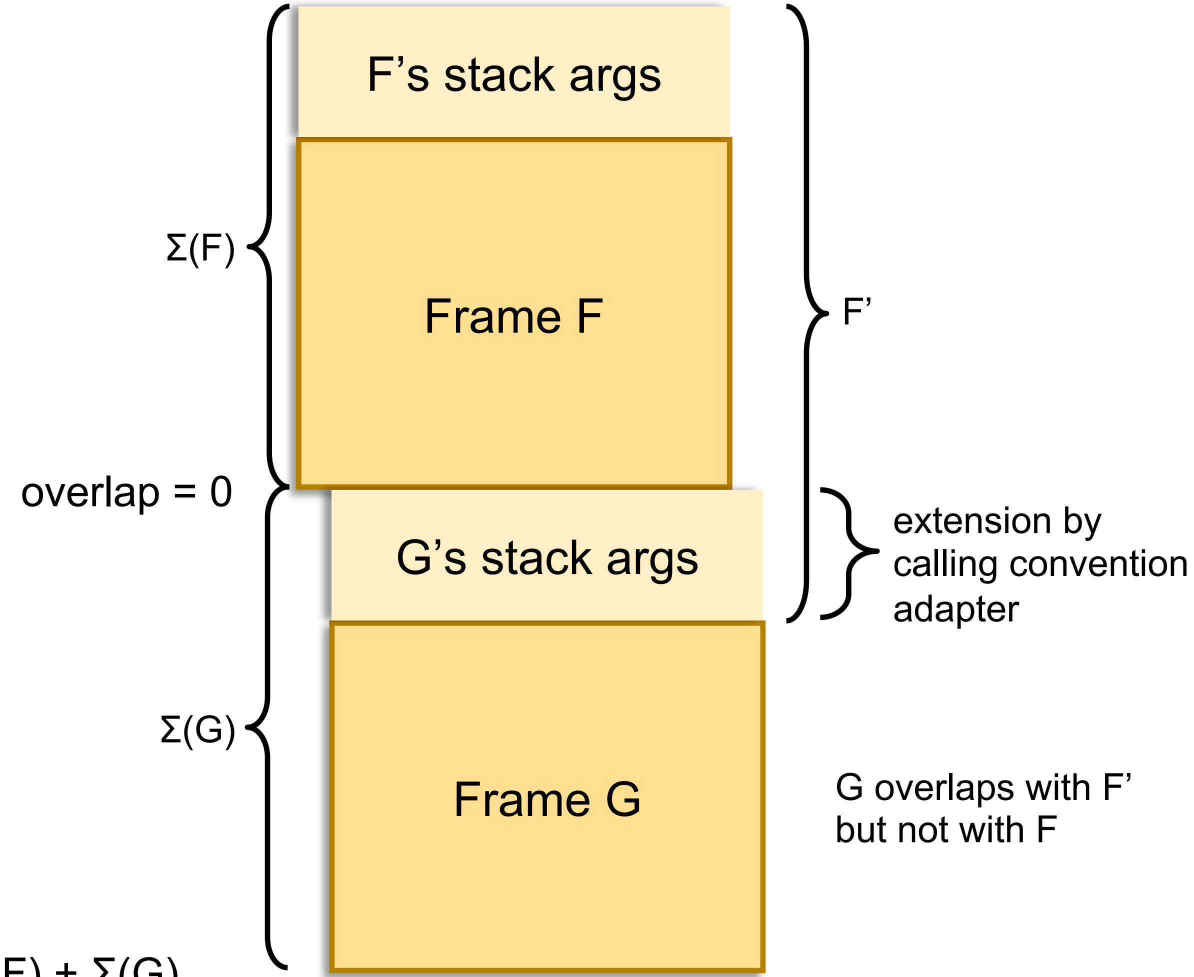


Freezing Frames – Slow Path – Sizing Caller/Callee

Same Frame Type: stackargs in Caller Frame



Different Frame Type: stackargs in ext. Caller Frame



$\Sigma(F, G) = \Sigma(F) + \Sigma(G)$
$- \text{overlap}$

Freezing Frames – Slow Path – Sizing with Metadata

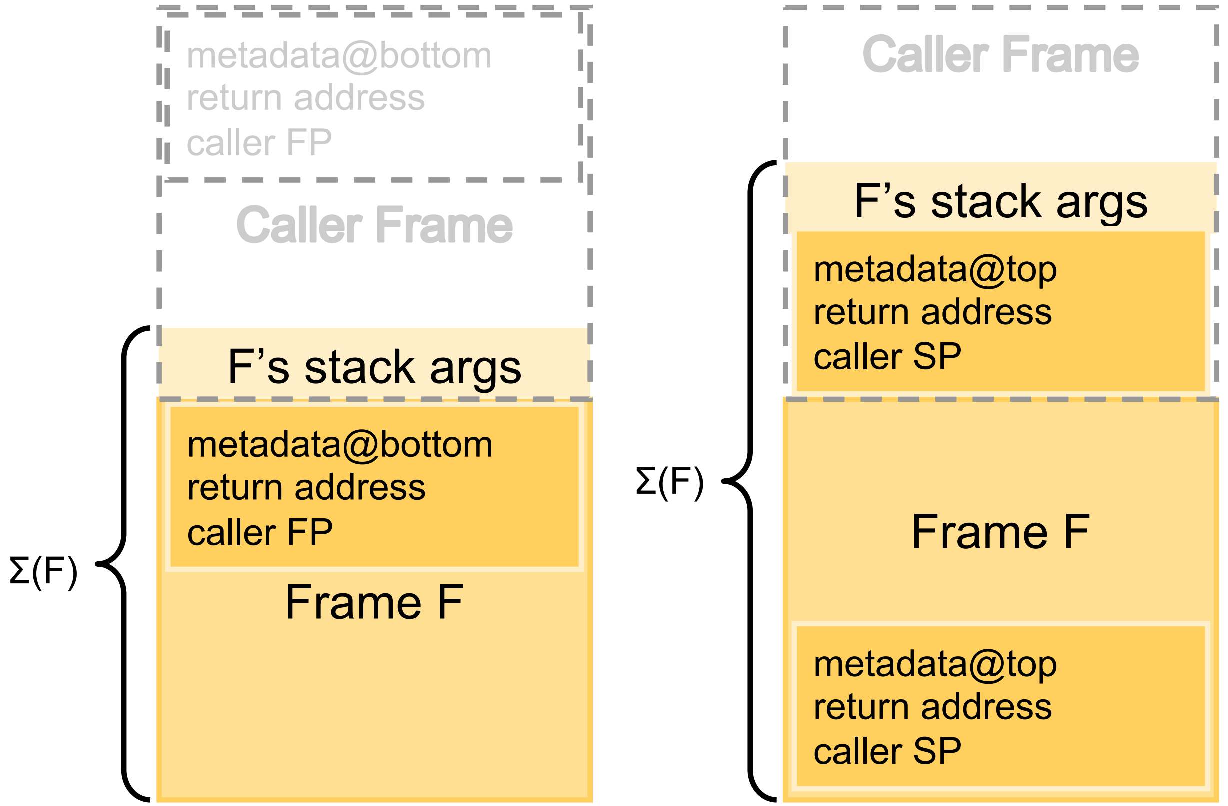
Metadata: return address, prev. FP, prev. SP, etc.

x86, aarch64

- metadata is stored at frame bottom (metadata@bottom)
- all metadata received from the caller is stored in own metadata

ppc64

- metadata at frame top
- not all metadata received from the caller is stored in own metadata
- return address is stored in caller's metadata
- Back chain: caller's SP is stored in own metadata, such that `callers_SP = *(calles_SP)`

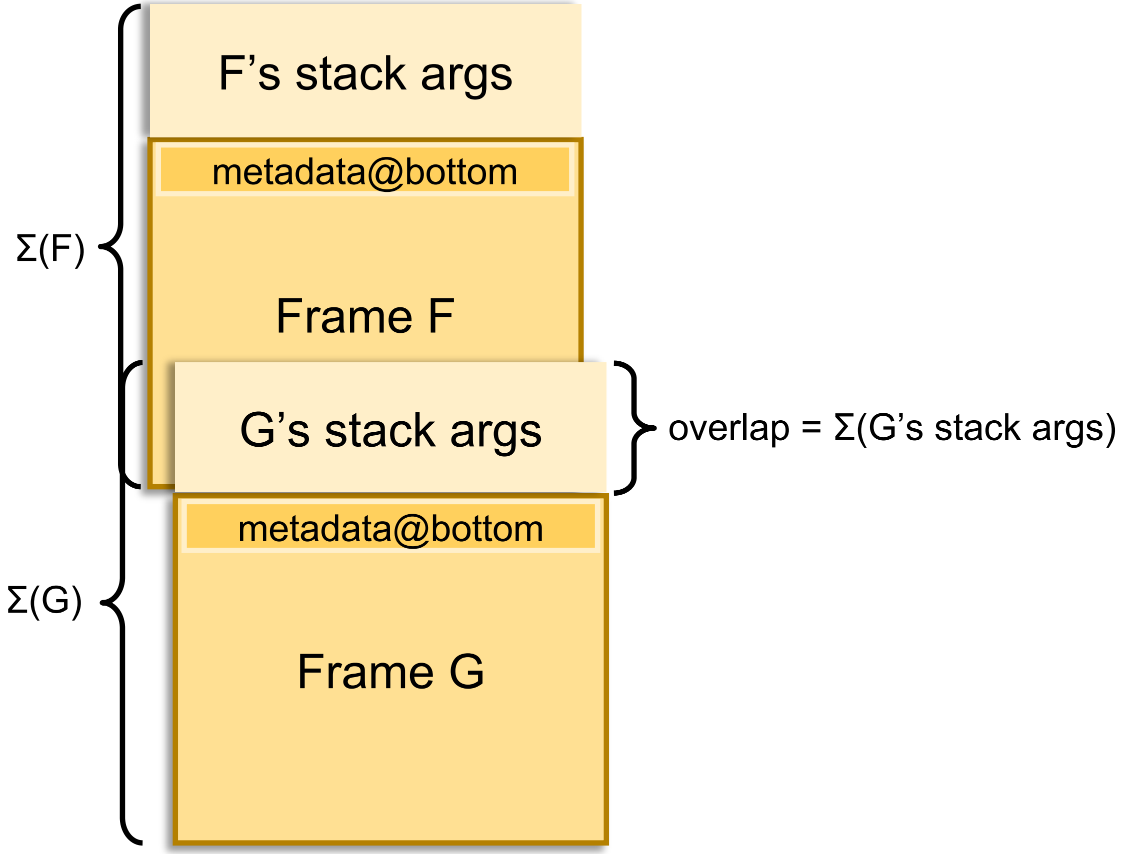


Σ(F) = frame_size(F) + Σ(F's stack args) + Σ(metadata@top)

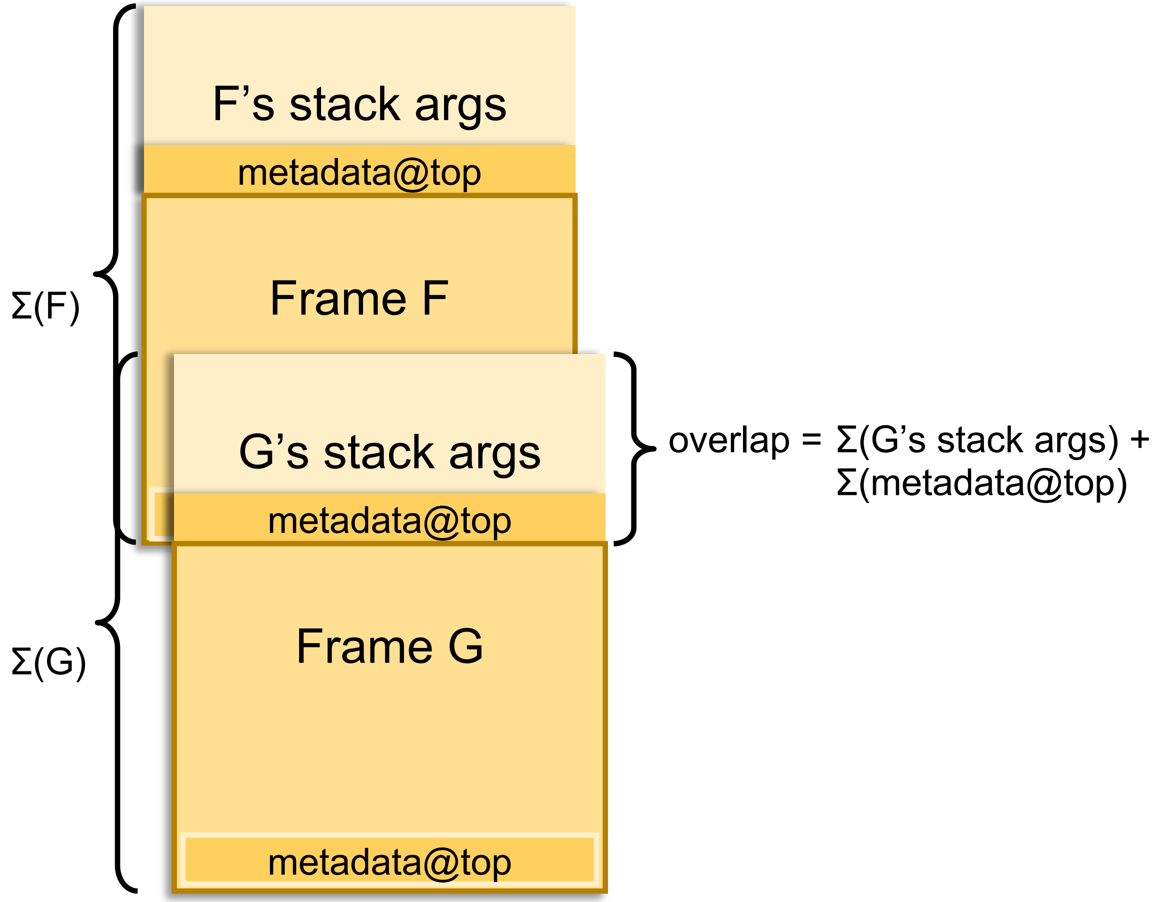
Σ(metadata@top) is a platform dependent constant

Freezing Frames – Slow Path – Sizing Caller/Callee – Metadata

Same Frame Type, metadata@bottom



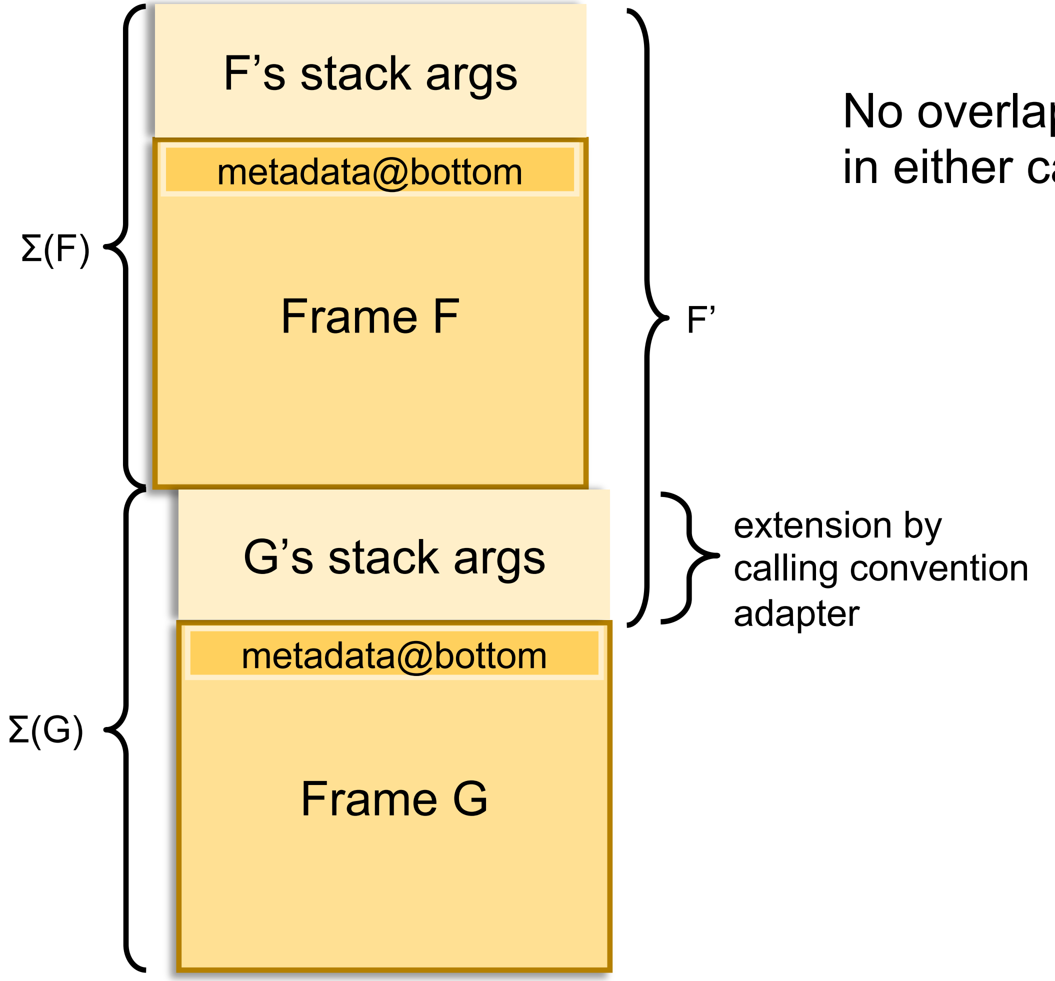
Same Frame Type, metadata@top



$$\Sigma(F, G) = \Sigma(F) + \Sigma(G) - \text{overlap}$$

Freezing Frames – Slow Path – Sizing Caller/Callee – Metadata

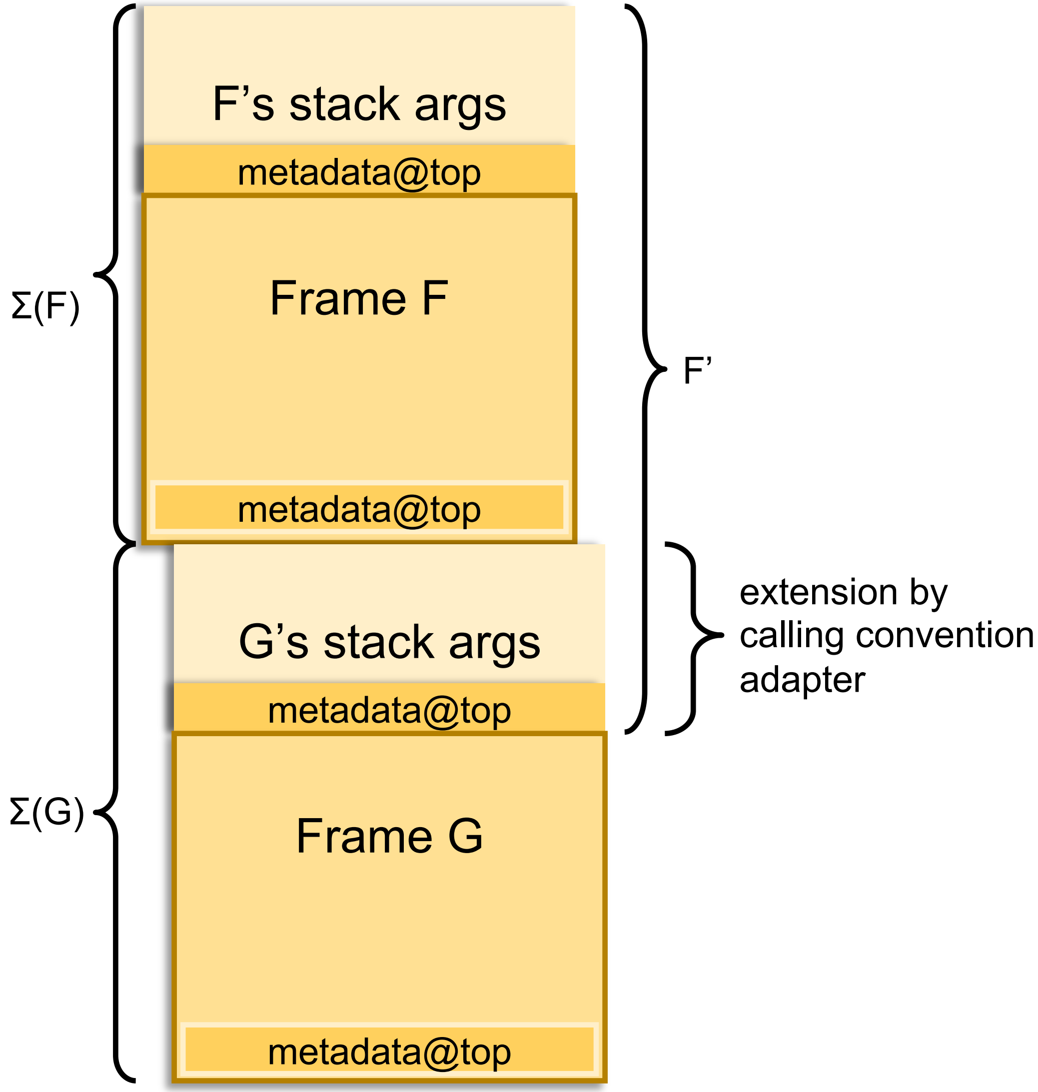
Different Frame Types, metadata@bottom



No overlap
in either case

$\Sigma(F, G) = \Sigma(F) + \Sigma(G)$

Different Frame Type, metadata@top



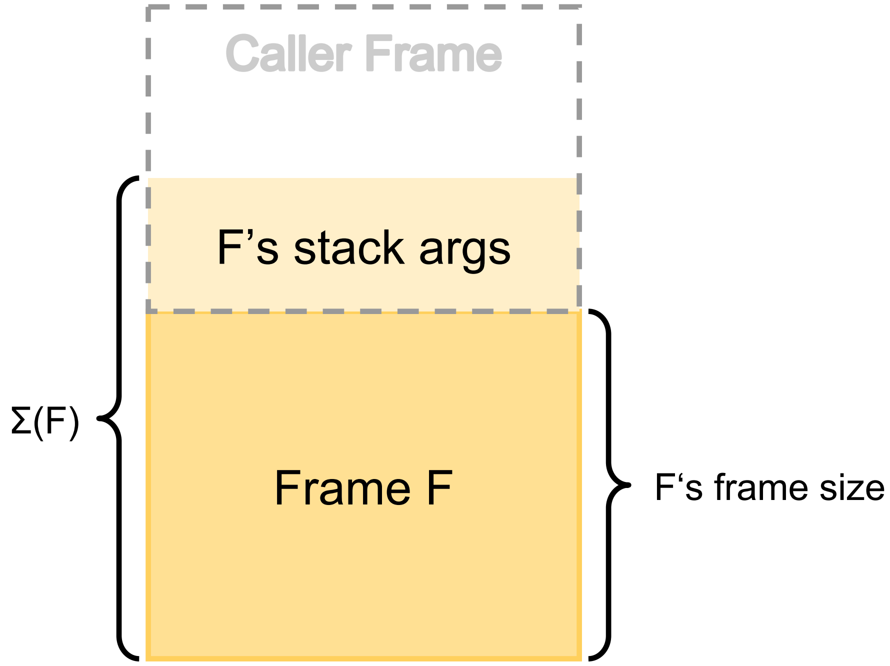
Freezing Frames – Slow Path – Fixups

Relativization

- Interpreter frames have pointers into own frame: `locals, esp, monitors(, ...)`
- each of them converted into an FP offset

Patching

- bottom frame: return barrier (from thaw) gets replaced with real return address, i.e. caller's pc
- interpreter frame: `sender_sp, back link` are relativized
- no fixup of back link of compiled frames



Unextended SP

Frames get extended

- by (Calling Convention) adapters to accommodate stack args as required by the callee's calling convention
- by the interpreter to accommodate locals that are not method parameters

A frame's SP before getting extended is the unextended SP

- In adapter and interpreter coding it is called sender SP
- passed to callee (ppc64: R21_sender_SP, x86_64: r13)
  – ignored if compiled
  – stored in interpreter state if interpreted
- Accessible for runtime code through `frame::unextended_sp()`
  Not always accurate, e.g., frame::sender_for_compiled_frame() sets unextended_SP = SP
  (which is correct if the caller is also compiled and irrelevant (not used), if it is interpreted)

Usage

- address values in a compiled frame
- restore original frame (actually `sender_sp` is used for that)

Hardly uses in shared code before Loom

Heap Frame Properties

Many uses of frame::unextended_sp() in continuations shared code

- Expectation: stack arguments are located @ unextended SP
- tweaked unextended SP of interpreted heap frames on PPC64
  - stack arguments @ unextended SP + Σ(metadata@top)
  - with that tweak interpreted and compiled frames on heap are uniform
  - see `ContinuationHelper::InterpretedFrame::patch_sender_sp()`
- Changed shared code to expect stack arguments @ unextended SP + Σ(metadata@top)
  - Introduced platform dependent constant `frame::metadata_words_at_top` for that

No alignment

- Slow path freeze even removes alignment padding

No backlink for compiled frames

- redundant as it can be computed from the SP + frame_size
- PPC64: back chain gets reconstructed when thawing

Freezing Frames – Fast Path

Fast Freeze

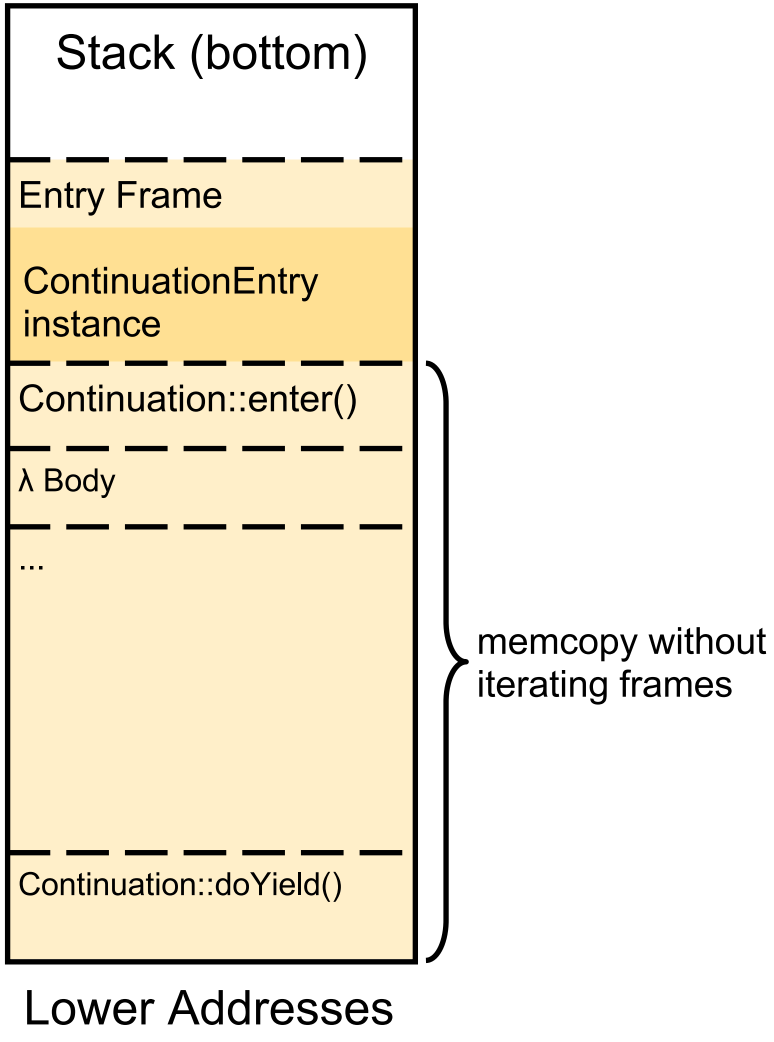
- without iteration of stack frames
- memcopy all between entry and top frame

Requirements

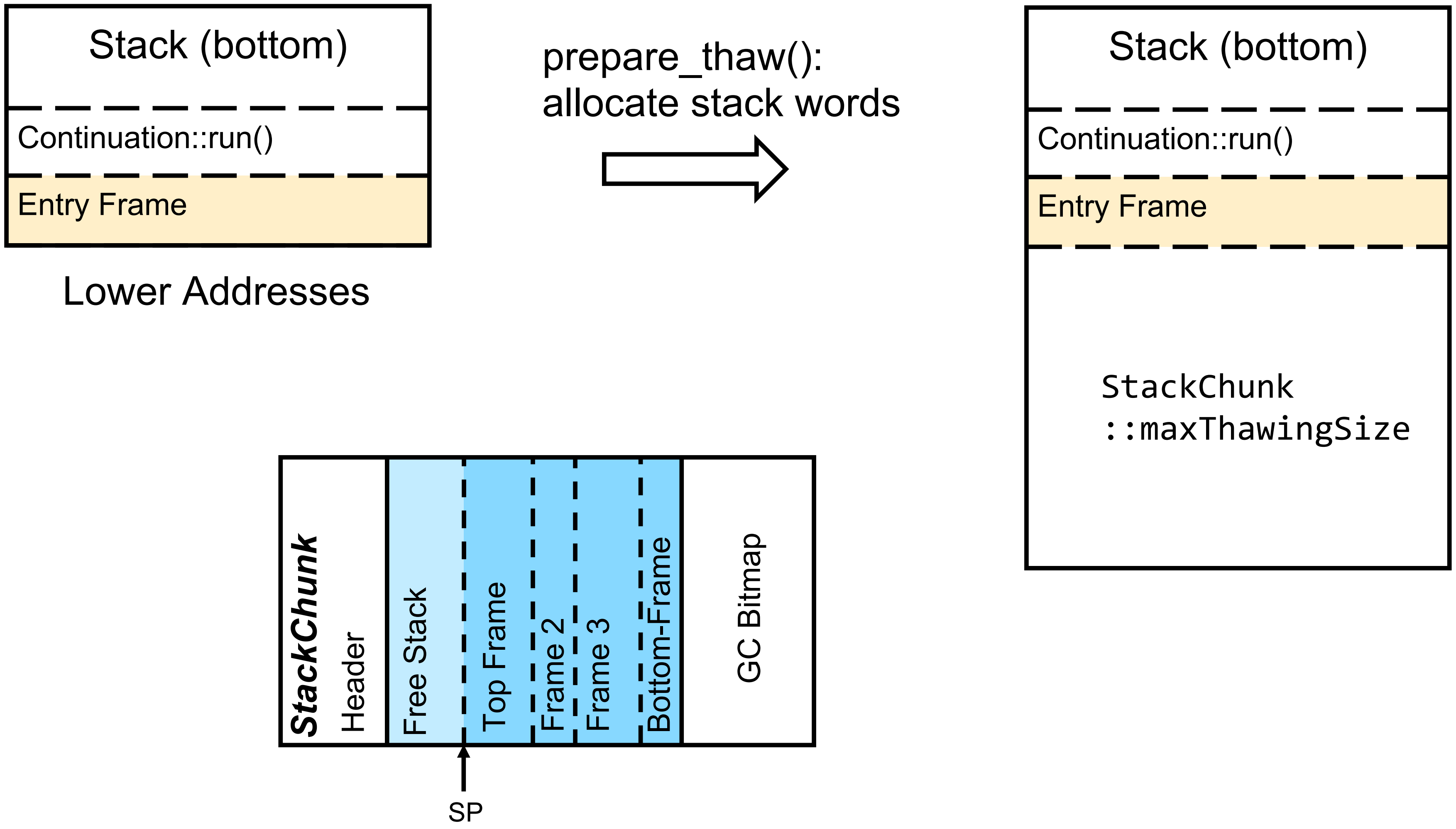
- no locks: check `JavaThread::_held_monitor_count`
- no interpreted / native frames
  - check `JavaThread::_cont_fastpath`: SP of oldest interp./native frame known
  - updated at i2c transitions / deoptimization
  - if interpreted or native frame exists then a i2c transition is needed to freeze
    - `doYield()` uses compiled calling convention
    - JNI to java calls use interpreted calling convention

No Fixups

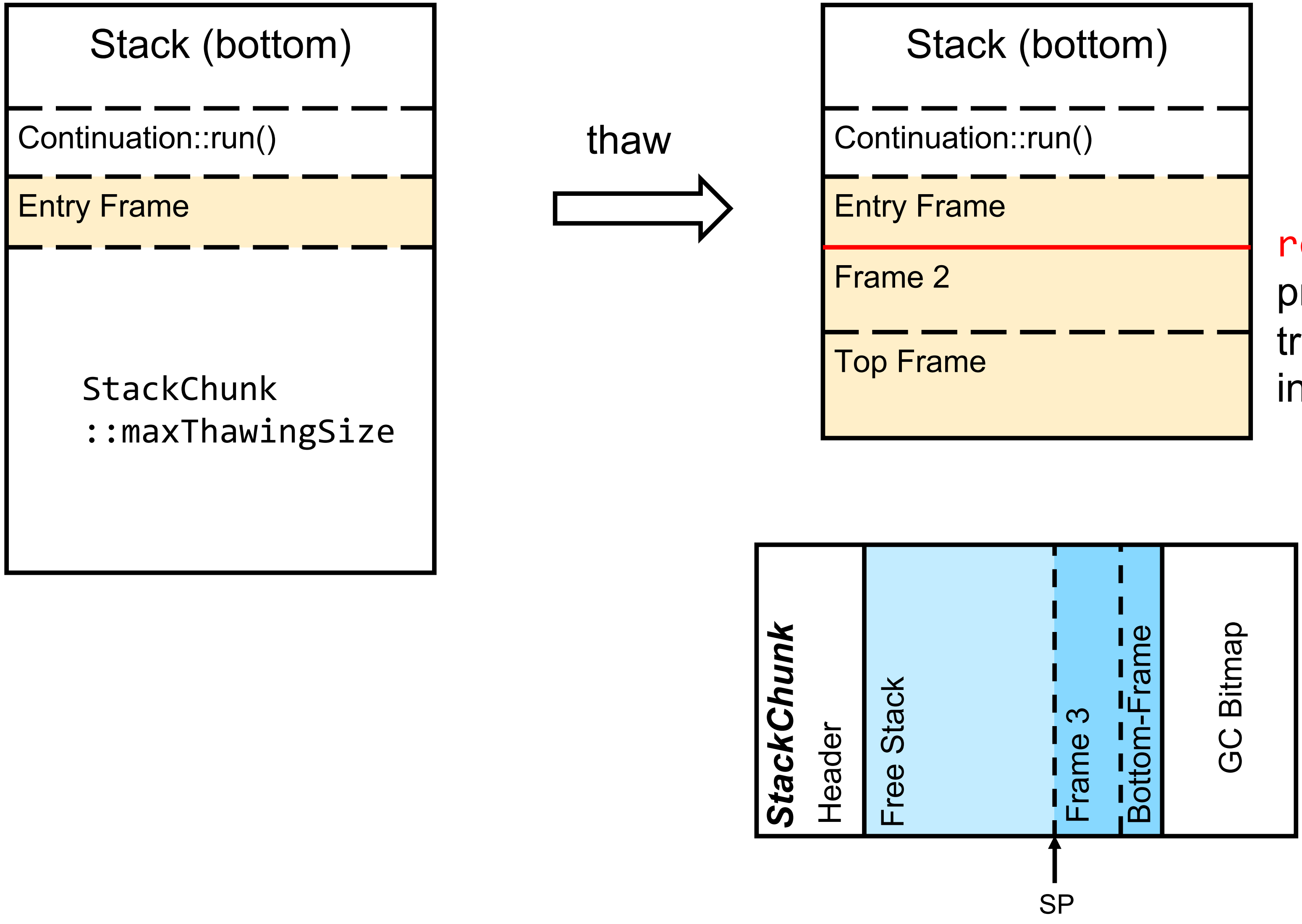
- no stack addresses in compiled frames
- except on ppc: back chain, redundant on heap



Thawing frames – Slow Path – Diagram (1/2)



Thawing frames – Slow Path – Diagram (2/2)



`return barrier`
prevents ret. to run()
triggers thaw of caller
in StackChunk

Thawing frames – Preparation

Injected field `maxThawingSize` of StackChunk C

- size needed to thaw all frames in C with correct alignment
- used to allocate words on stack for frames to be thawed
- actual thaw happens in runtime

Thawing frames – Slow Path

Recursively iterate frames in StackChunk C on heap

- effectively 1 - 3 frames are thawed
  - limit: `int num_frames = (return_barrier ? 1 : 2);`
    !return_barrier: first thaw for run() call; 2 topmost frames are known to return
  - one extra frame can be thawed to avoid situations where the top frame in C is compiled and its callee on stack is interpreted (`”it makes detecting that situation and adjusting unextended_sp tricky”`)
- code: `ThawBase::new_stack_frame()` and its callers

Fixups

- derelativize interpreter frames
- set correct back link of compiled frames
- uncompress oops, derelativize derived pointers
- set return_barrier as return address for bottom frame iff C is not empty after thaw to thaw the bottom frames caller

Thawing frames – Fast Path

Fast thaw

- without iteration of heap frames
- memcopy all in StackChunk

Requirements

- only compiled frames in StackChunk
- not transformed by GC (see GC Integration)
- no special GC barriers needed for StackChunk (Shenandoah, ZGC)
- `!PreserveFramePointer` because otherwise the frames need to be iterated to setup the FramePointers
- Size of frames in StackChunk < 500 words

No Fixups

- Exception: back chain needs to be restored on ppc64
  (see `Thaw<ConfigT>::patch_caller_links()`)

GC Integration

new Klass: `InstanceStackChunkKlass`

- Used to get size of a StackChunk (instances differ in sizes)
- find references in stack using the bitmap stored in each StackChunk instance

StackChunks are lazily transformed for GC just before being traversed

- initialization of oop bitmap
- compressing of oops
- relativization of derived pointers, i.e. converted to offsets
  Derived pointers are pointers into heap objects
- after transformation frames cannot be frozen or thawed on the fast path
  - freeze: difficult to synchronize the necessary transformation with concurrent GCs
  - thaw: would require iterating the frames to uncompress oops and derelativize derived pointers
- see `stackChunkOopDesc::transform()`

Bitmap is cleared when thawing

- only the part that covers the stack args of the bottom frame being thawed
- no need to clear bitmap for all thawed frames because the bits covering free region are ignored

Transformed StackChunks cannot accommodate new frames anymore

Potential Performance Cliffs

If a StackChunk is transformed for GC

- it cannot accommodate new frames
  a new one must be allocated if the continuation yields again
- its frames cannot be fast thawed

Large StackChunks (> 500 words) cannot be fast thawed

Not compilable methods prevent fast path context switch

Integration with established Thread and Stack related (internal) VM APIs

Iterating Physical Frames (`class frame`)

- `RegisterMap`: holds stack walk state, e.g. to find callee saved registers
  - `_chunk`: StalkChunk currently walked
  - `_walk_cont`: where to continue a walk at the callee of the entry frame
    - either with the top frame on heap, used, e.g., in JVMTI
    - or with the entry frame
- `frame::_on_heap` indicates if a frame resides in a StackChunk on heap
- `frame::sender_raw()` uses `stackChunkOopDesc::sender(frame&)` to get the sender of a heap frame
- `StackChunkFrameStream`
  - Iterator for frames in a StackChunk
  - used by `stackChunkOopDesc::sender(frame&)`
- `frame` instances are relativized if representing heap frames
  - heap pointers (_sp, _fp) are offsets relative to the `StackChunk` (wrapped in handle)
  - relativized frames are not invalidated by safepoints
  - derelativize before use, e.g. `stackChunkOopDesc::interpreter_frame_method()`

Integration with established Thread and Stack related (internal) VM APIs

Virtual Frames (`class vframe`)

- abstracts away details like relativized frames

`JavaThread`

- `_vthread`: returned by `j.l.Thread.currentThread()`
- `_cont_entry`: list head of ContinuationEntrys
- `vthread_continuation`(): finds first VirtualThread continuation in `_cont_entry` list.

Testing

make test TEST="hotspot_loom jdk_loom" TEST_VM_OPTS="-XX:+VerifyContinuations”

- x86_64, aarch64, ppc64le
- fastdebug, release

References

JBS Item: https://bugs.openjdk.org/browse/JDK-8286302

Code: https://github.com/reinrich/jdk/tree/ppc_port

Thank you.

Contact information:

Richard Reingruber
richard.reingruber@sap.com

THE BEST RUN

© 2022 SAP SE or an SAP affiliate company. All rights reserved. See Legal Notice on www.sap.com/legal-notice for use terms, disclaimers, disclosures, or restrictions related to SAP Materials for general audiences.