Proto-RunTime for DSL Implementations

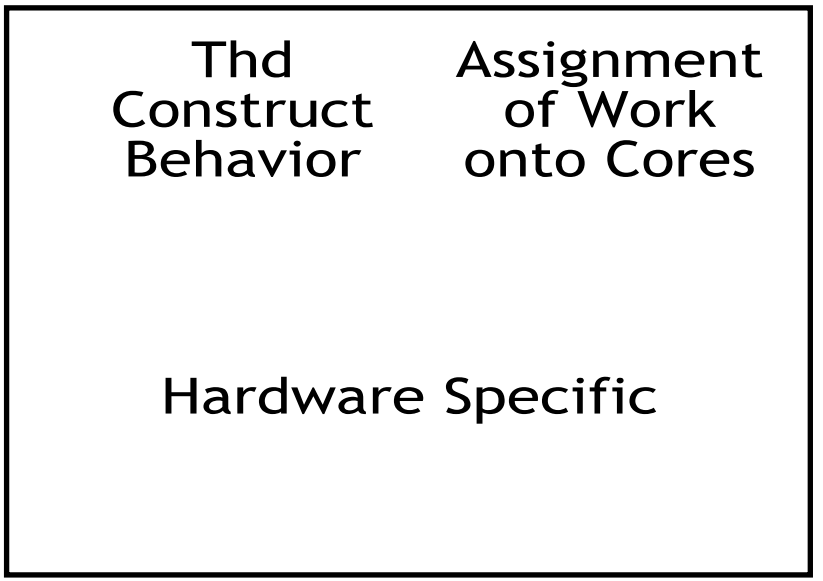
Sean Halle
CWI and OpenSourceResearchInstitute.org

Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

Runtime Systems

- A language defines the behavior of constructs
  - Behavior of constructs is implemented in runtime
  - Assignment of work onto cores is in runtime
  - Performance of runtime depends on HW details
    - Comm btwn cores for runtime internals: “constraint update”



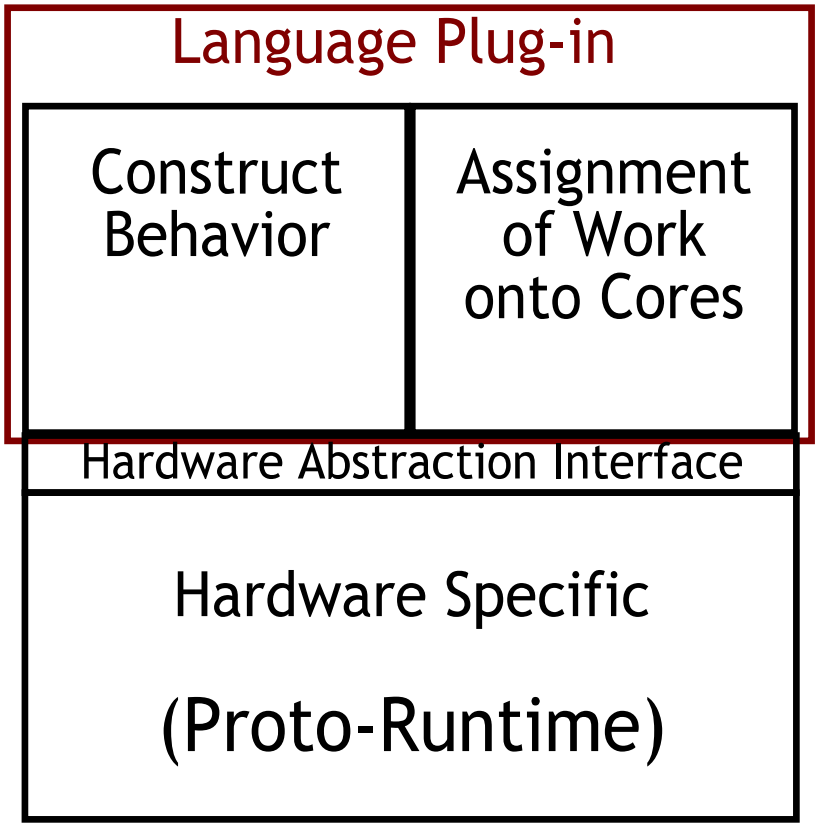
Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

Modularization of Runtime

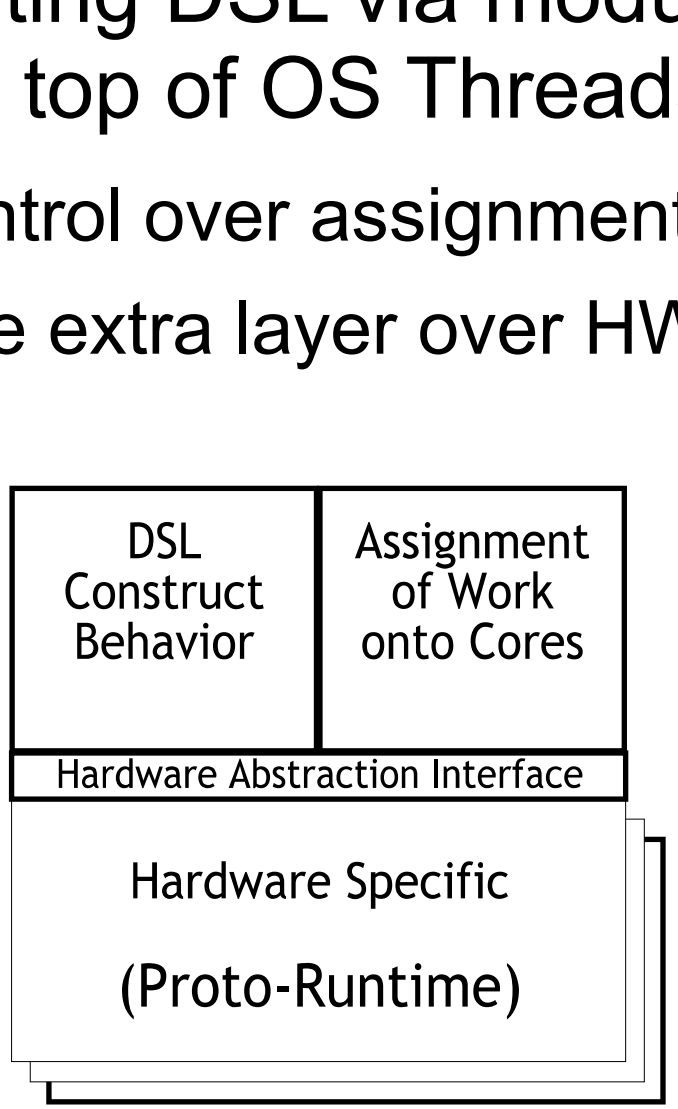
- Three modules:
  - HW specific encapsulates low level details
  - Construct Behavior Module
  - Assignment Module
- Interface between modules abstracts hardware
  - HW specific module enforces interface

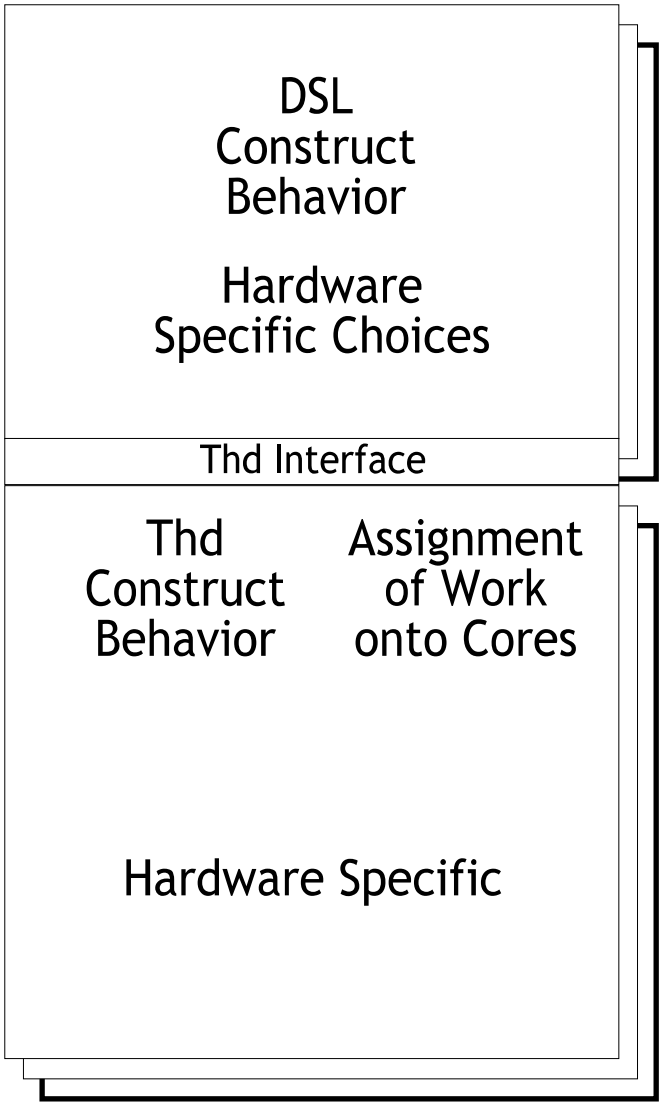
runtime system



Modularization of Runtime

- Implementing DSL via modules versus on top of OS Threads
  - Gain control over assignment
  - Eliminate extra layer over HW



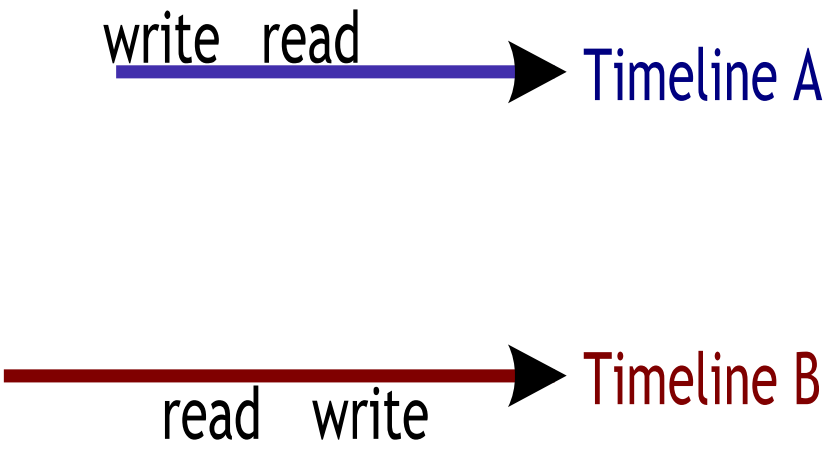


Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

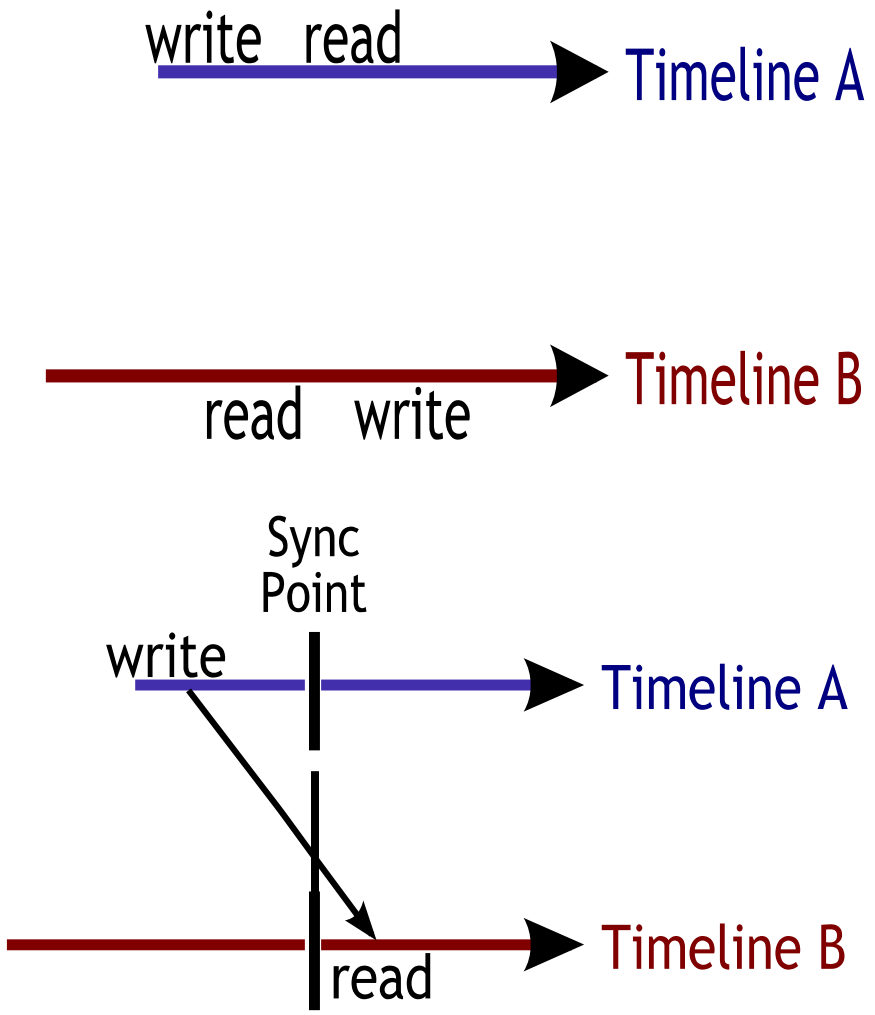
Parallelism

- About multiple timelines
- Independent, can slide relative to each other



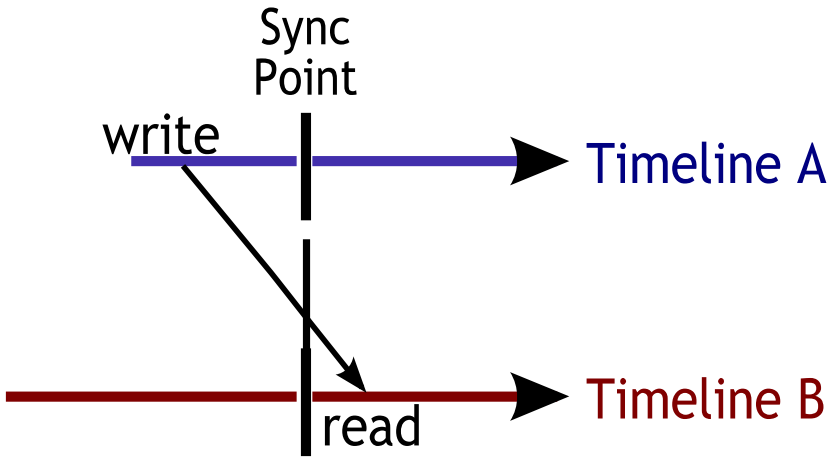
Parallelism

- About multiple timelines
- Independent, can slide relative to each other
- When communicating, have to constrain slide
  - Sync construct does that



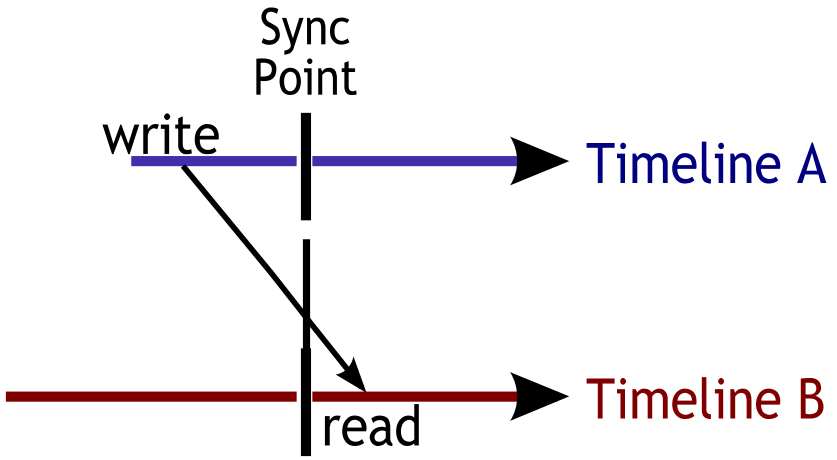
Sync Constructs

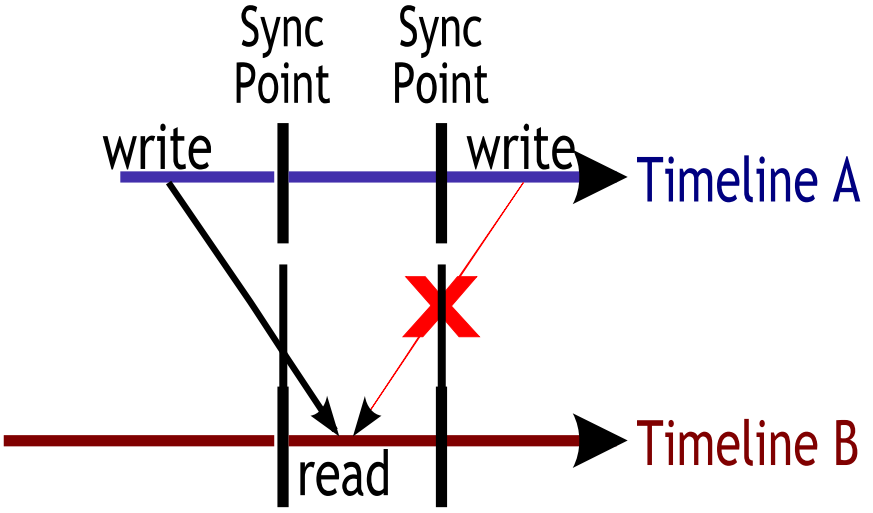
- Sync controls slide
- Controls side-effect based communication between timelines



Sync Constructs

- Sync controls slide
- Controls side-effect based communication between timelines
- Ensures wrong comm does not happen



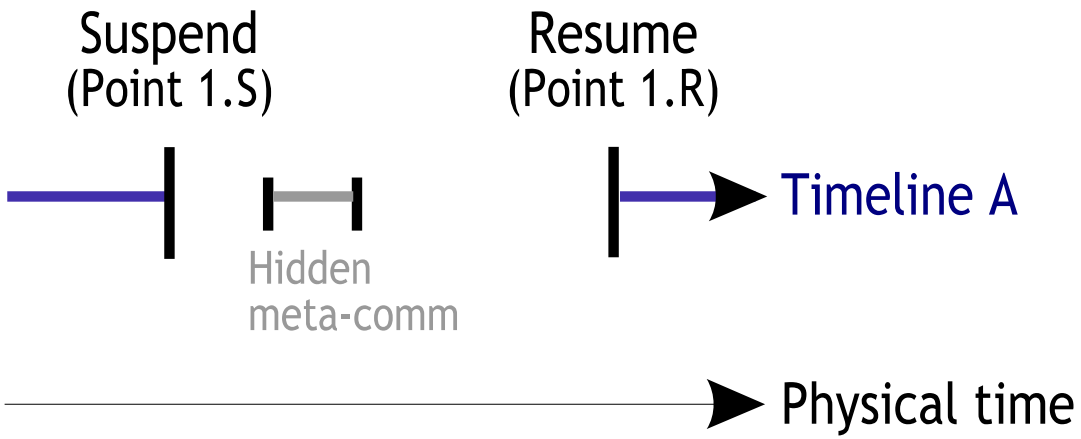


Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

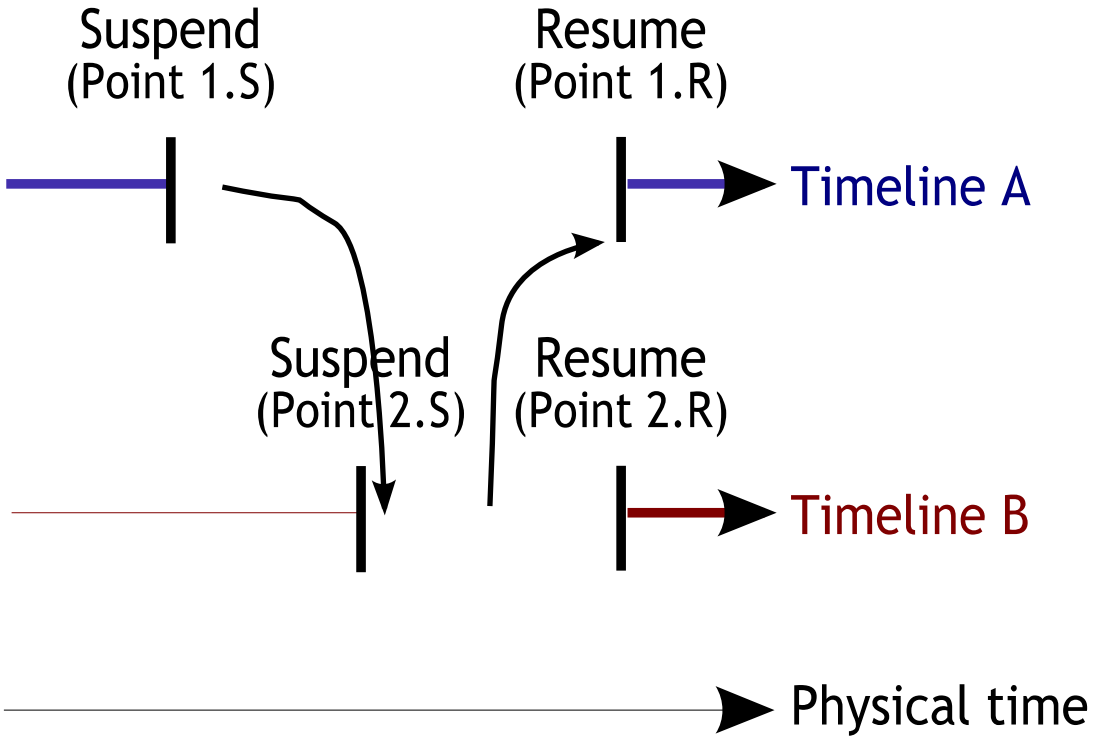
Tie-Point

- Below level of Threads: HW abstraction
- Controls suspend and resume of timelines
- Implies “hidden” meta-comm about timeline state



Tie-Point

- Sync construct == variable delay, causally ended
  - Things happen in system, cause timeline to resume
- Chain of meta-comm “causes” implements a sync construct



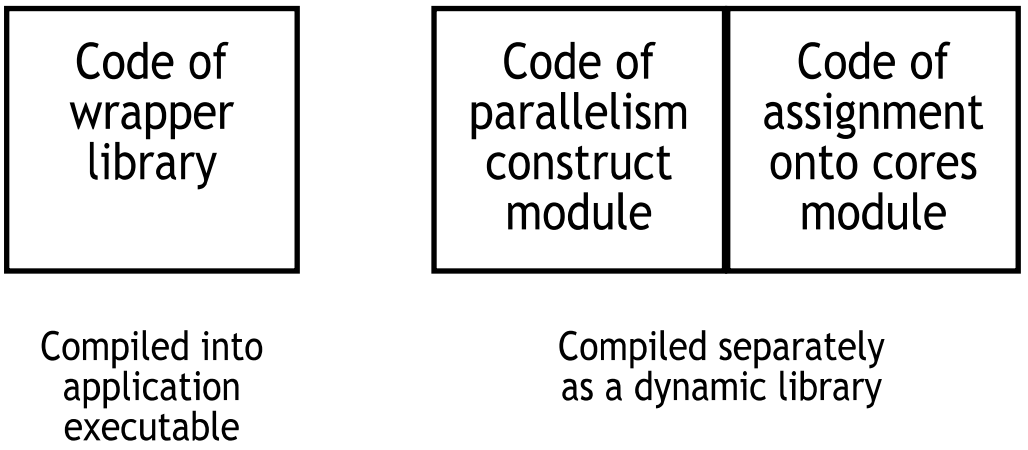
Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

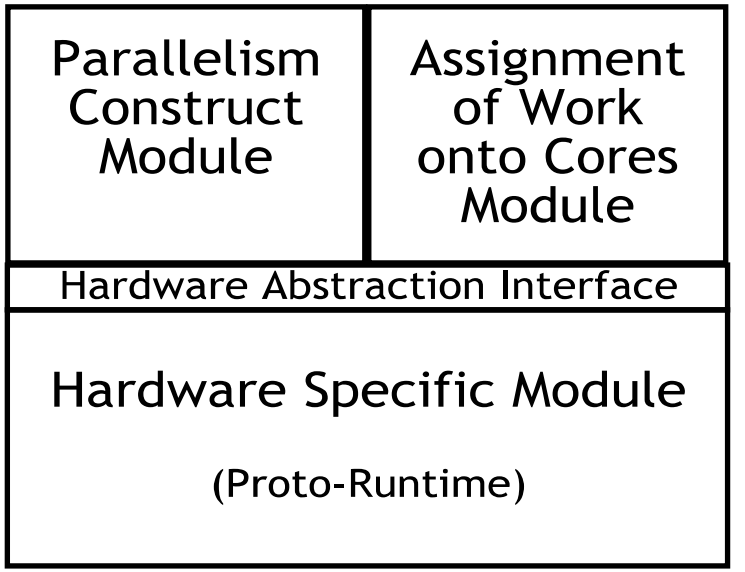
Code Stack of the Language

- Lang has 2 parts: wrapper lib and plugin
- Wrapper lib compiled into App executable
  - calls proto-runtime primitives (provided by PRT lib)
- Plugin compiled in dynamic lib, called by PRT

Code Breakdown of a Language Implementation



Code Stack for Runtime System



Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

Implementing a Language

- Implement synchronization behavior
  - Plugin code does meta-comm to create tie-points
    - Tie-point created when meta-computation inside suspend call on one timeline causes resume of other timelines
  - Meta-comp is request handler code, inside plugin
  - Requestion handler:
    - Called when a timeline suspends
    - Manages which timelines remain suspended
    - Chooses which get resumed
    - Request handler interactions make the causality chain
      - That chain is the “semantics” of a sync construct

Impl of Mutex Acquire and Release

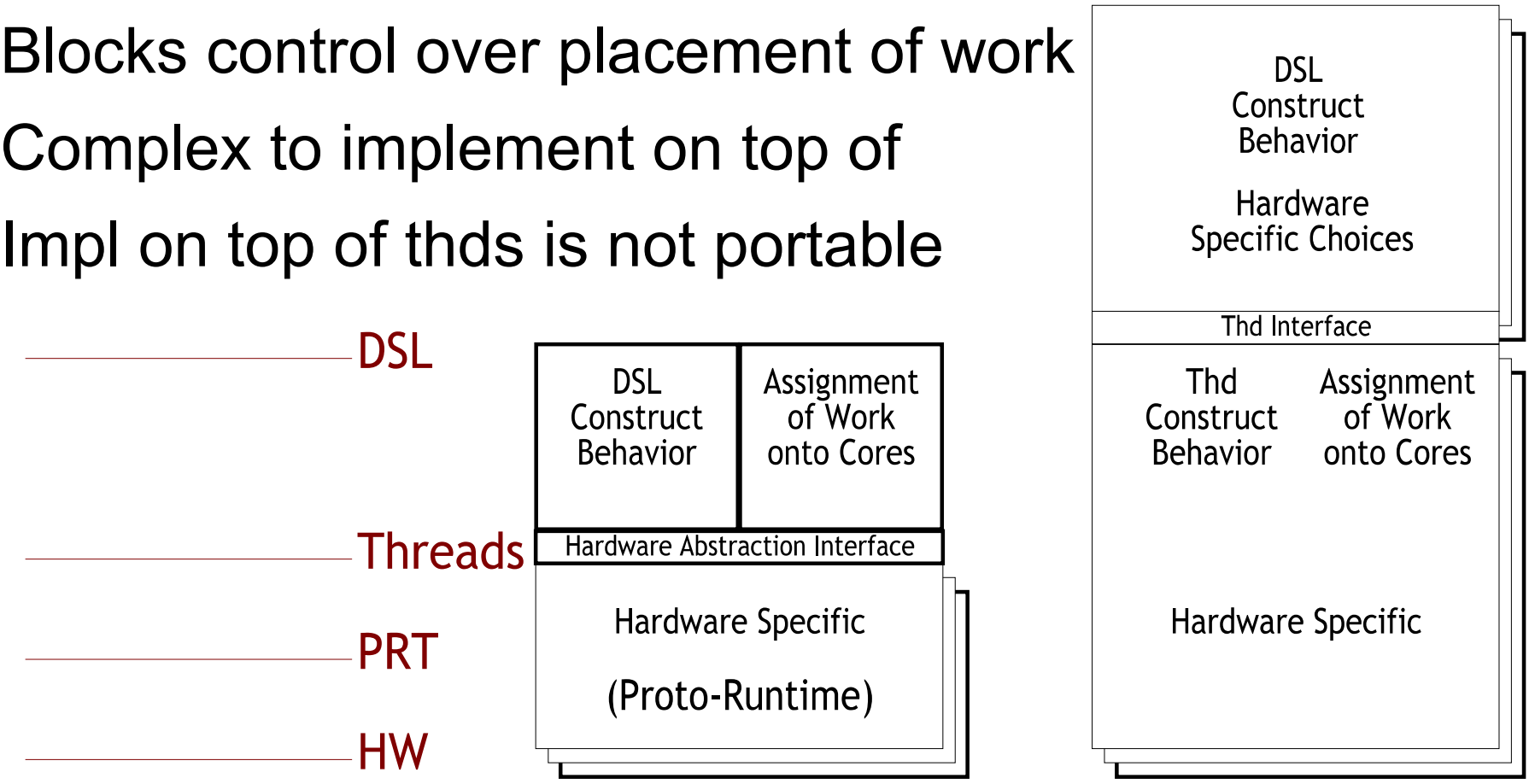
- Mutex data struct:
  - “current owner” field
  - “queue of waiting timelines” field
- Acquire:
  - Check current owner, if full, place calling timeline into queue, else place calling timeline into owner
- Release:
  - Set owner to NULL, then get next from queue, put that into owner, and resume it, as well as resume callling timeline
    - The resume of both timelines ties them together

Outline

- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

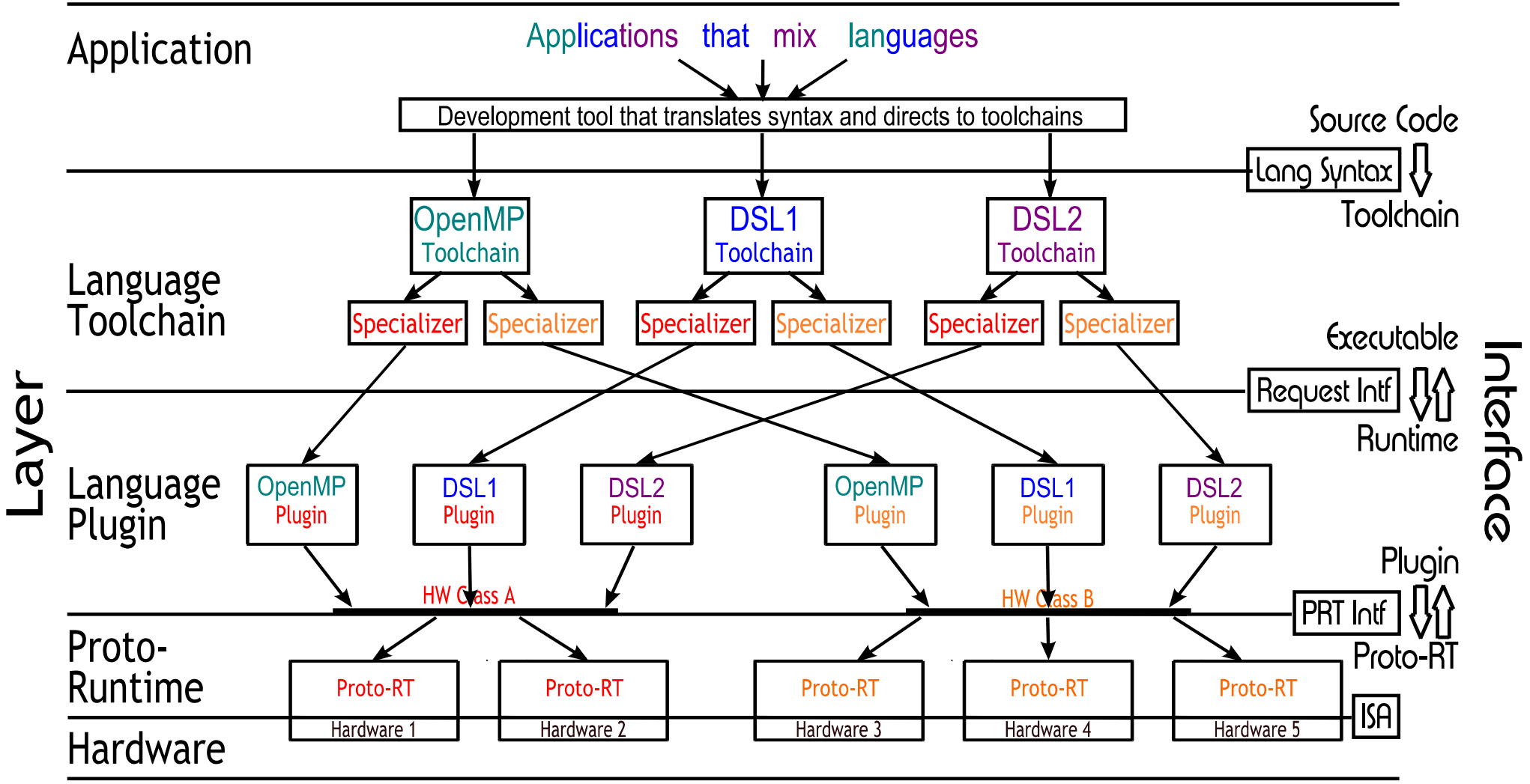
Why PRT?

- Threads is too low level for application code
- But it's too high level for implementing DSL
  - Blocks control over placement of work
  - Complex to implement on top of
  - Impl on top of thds is not portable



Why PRT?

- Eco system when consider industry-wide



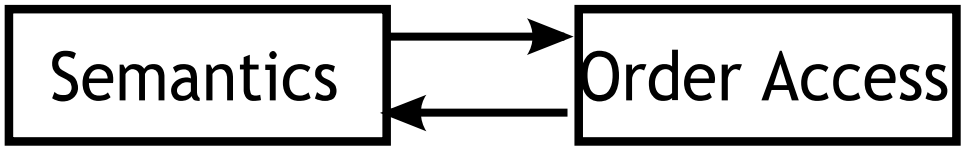
Why PRT?

- Less overhead, simpler, more control
  - Higher performance, when work size small
  - Simple, straight-forward implementation
  - Control over execution model and work placement
    - Higher performance
  - Services for debugging, performance tuning
  - Inherit runtime performance tuning
  - Single implementation ports (unlike Thd impl when performance matters and machine array large)

Conclusion

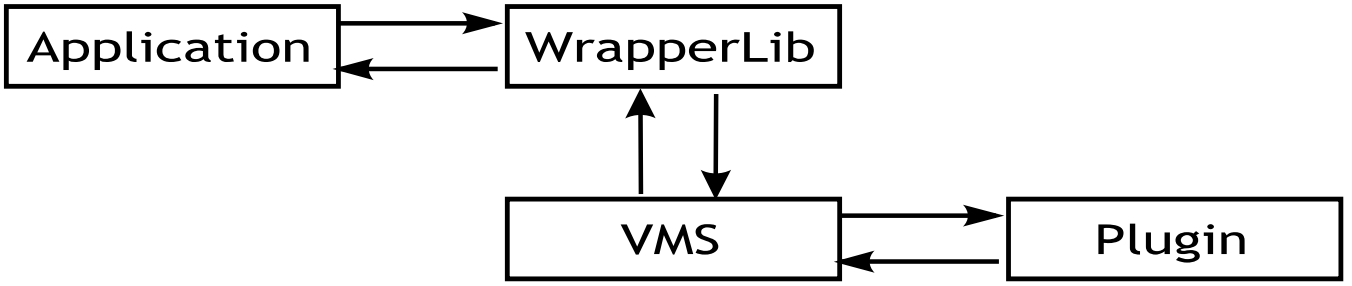
- Runtime Systems: “big picture, the problem”
- Modularization: “top level solution”
- Parallelism & Sync: “requirement of solution”
- Tie-point: “concept enables soln”
- Language Code Stack: “top level how to use”
- Language Impl Details: “details how to use”
- Why ProtoRunTime

Sync-Constructs Have 2 Parts



- 1) establish ordering
- 2) semantics
- Ex: decomposition of CAS instruction
  - 1) HW in Mem: only one core can access address
    - creates sequential ordering across timelines (cores)
  - 2) semantics in pipe: “compare, swap other if same”
    - can reuse order mech. with different sem. (Ex: x86)
- VMS == an analog (sync impl. in 2 parts)
  - VMS virtualizes ordering mechanism (virt time)
  - Interface connects custom semantics to order mech

How Use VMS



- App calls wrapperLib “mutex_lock(mutex, VP);”
- Inside call, switch to VMS
- VMS calls plugin (custom sync sem)
- Plugin chooses when to switch back => completes call == timing behavior == sync construct

- Key Feature: plugin written in sequential style
  - (NOT a sequential program, but no concerns about shared data nor concurrency in alg – easy impl.)

Step Back: What Are We After?

- Triple goals:
  - High productivity (similar to sequential dev.)
  - High portability (performant on all)
  - High Adoptability (this is real-world research!)
- Productivity requires rapid cycle
  - Give constructs to coders, see results, modify
- Portability to be practical requires eco-system
  - reduce work of porting *language,* organize players
- Adoptability requires fitting existing practices
  - Lib-based → same tools, 99% of code sequential

Portability Motivation

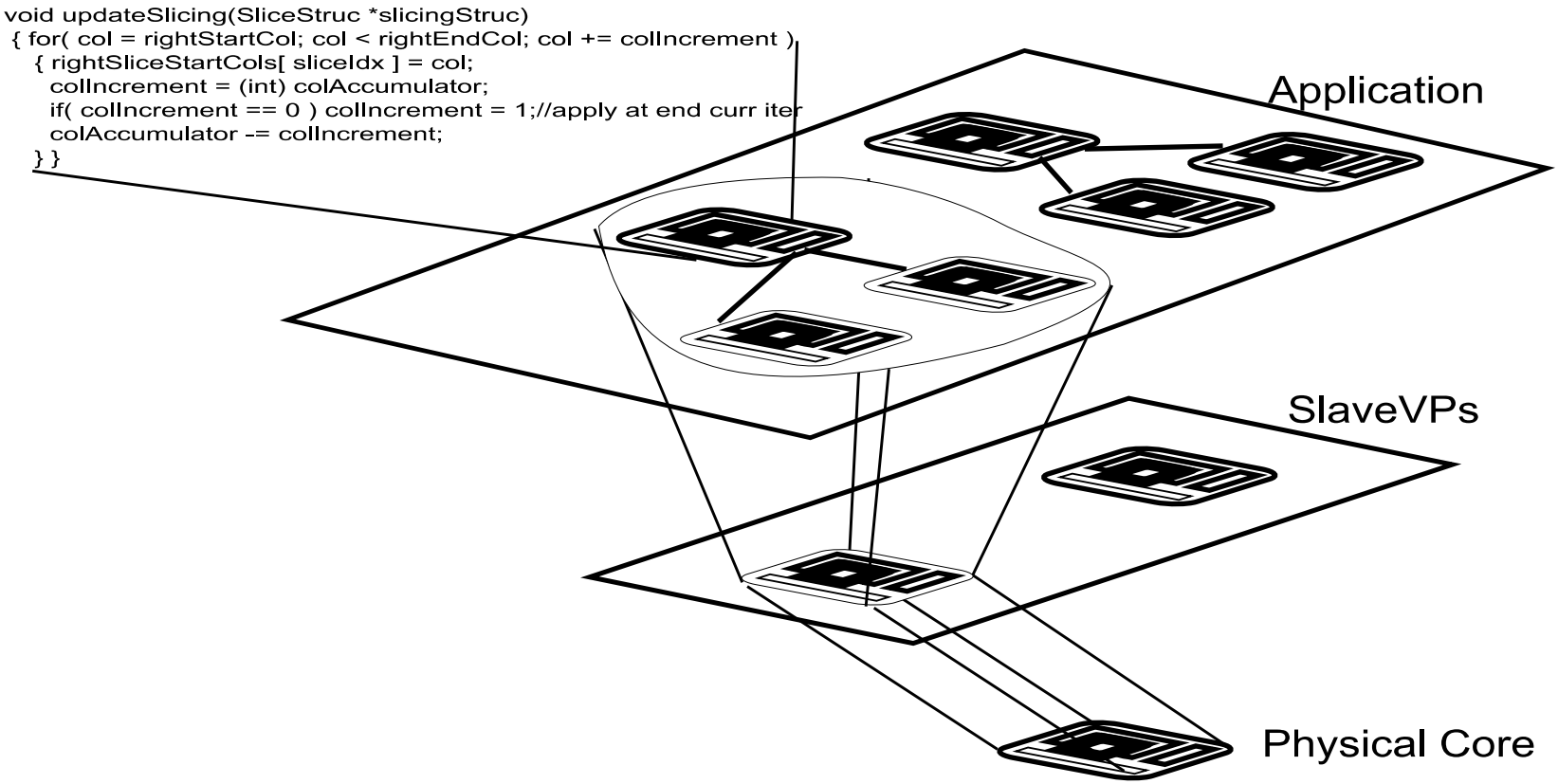
- Portability is an eco-system problem
  - Need “middleware” to reuse runtime work
    - Many languages reuse scheduler work
    - Funnel many HW up to one interface, use one runtime
  - Several goals must be met *simultaneously*
  - VMS provides what's necessary, but *not* sufficient
- Targeting source-level portability
  - Possibly recompiled many times, but *source* same
  - Requires proper portability constructs in lang
    - Dependencies, task resizing → match hierarchy of HW

Benefits

- Research on languages – increase exploration
  - Ex: Library-based lang embedded into base lang
  - quick and easy (vs normal runtime on top of Thds)
- Domain-specific languages – more practical
  - Less work to port lang to many targets
- Application-specific (lib-based) languages
  - parallelism encoded inside custom constructs
  - splits development: application-coding vs construct coding -- splits expertise cleanly
  - Ex: Implement parallelism in H.264 as language

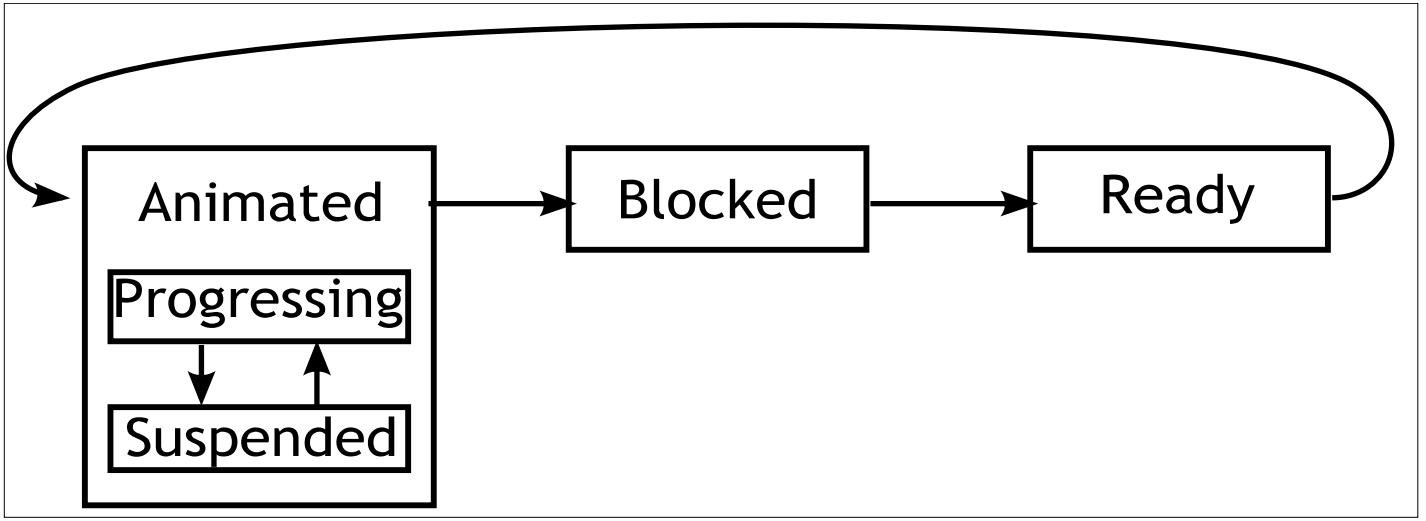
Specification of VMS

View of VMS System



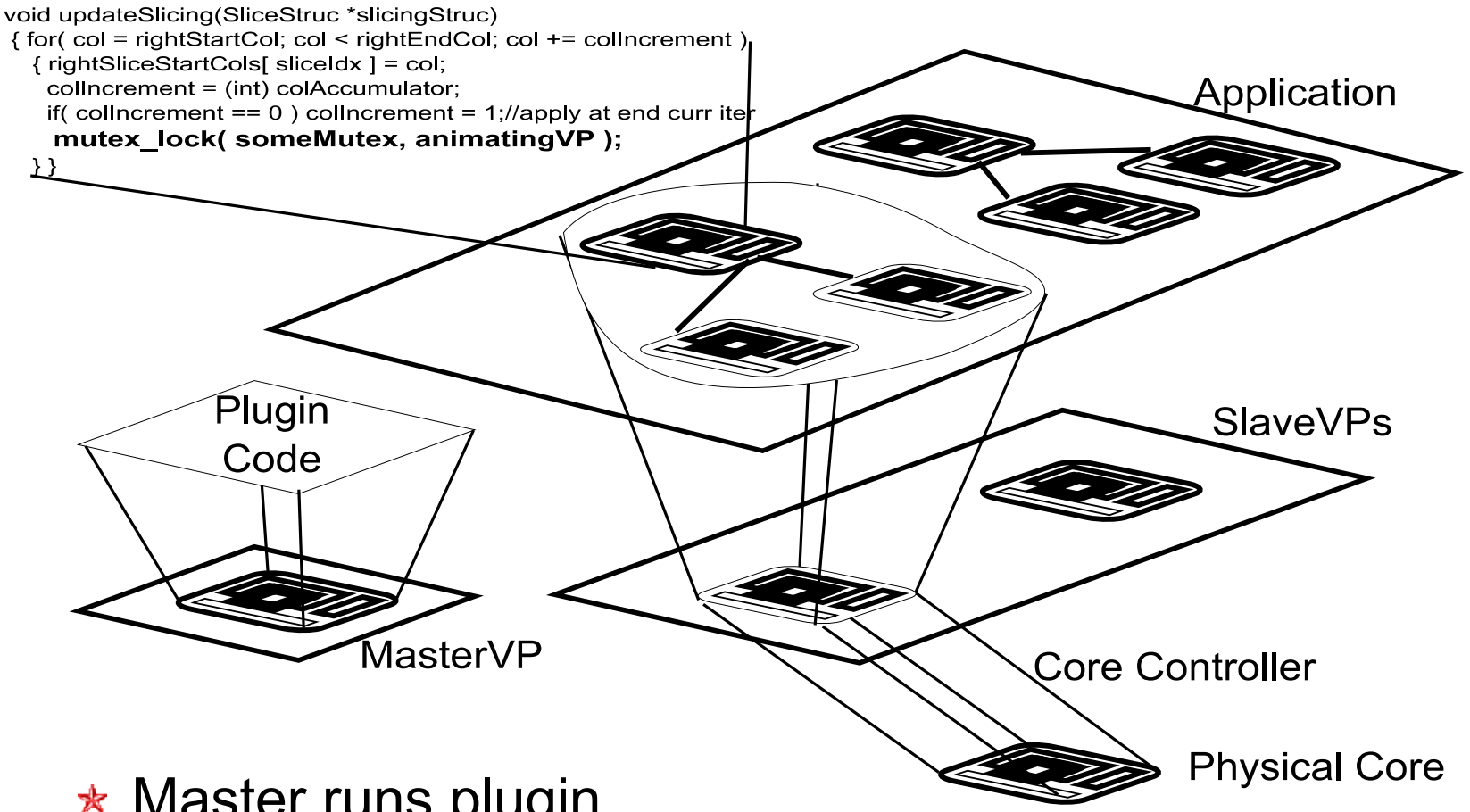
- See dissertation for theory of animators

Slave State



- Virtual-timeline globally orders state changes
  - analog of cross-timeline hardware in mem system
- Slave initiates Animated->Blocked Transition
  - By suspending (inside wrapper library)
- Plugin: Blocked->Ready and Ready->Animated

View of VMS System



- Master runs plugin
- Plugin decides timing of (re)animating SlaveVPs
- Plugin maps Slave VP to physical core
- Core Controller sequences the phys core among the VPs

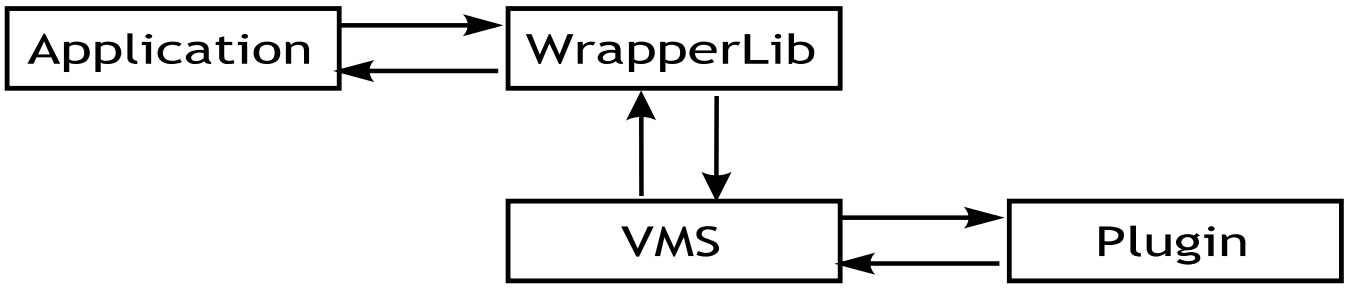
Slave State 2

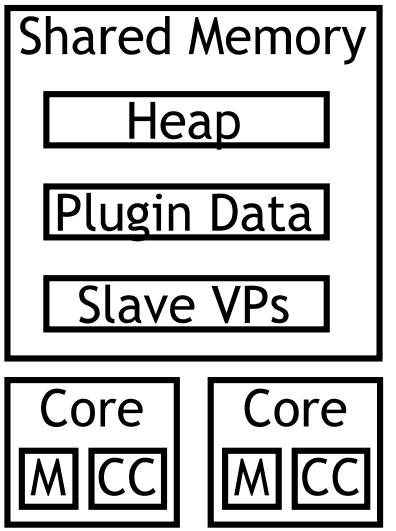
- Suspended and Progressing are local to Core
  - Slave suspending doesn't change its State!
- Master notices suspension later, does state chg
  - Inserts Animated → Blocked into Virtual-timeline
- The slack is critical to Performance Advantage
  - At suspend, other Slave transitions to progressing
  - Hides synchronization wait-time (for all languages)
  - Exploits parallelism to keep processor busy
  - vs atomic-instr, lock, or barrier, which idle processor

Implementation

user-level proof of concept

Elements of *One* Implementation

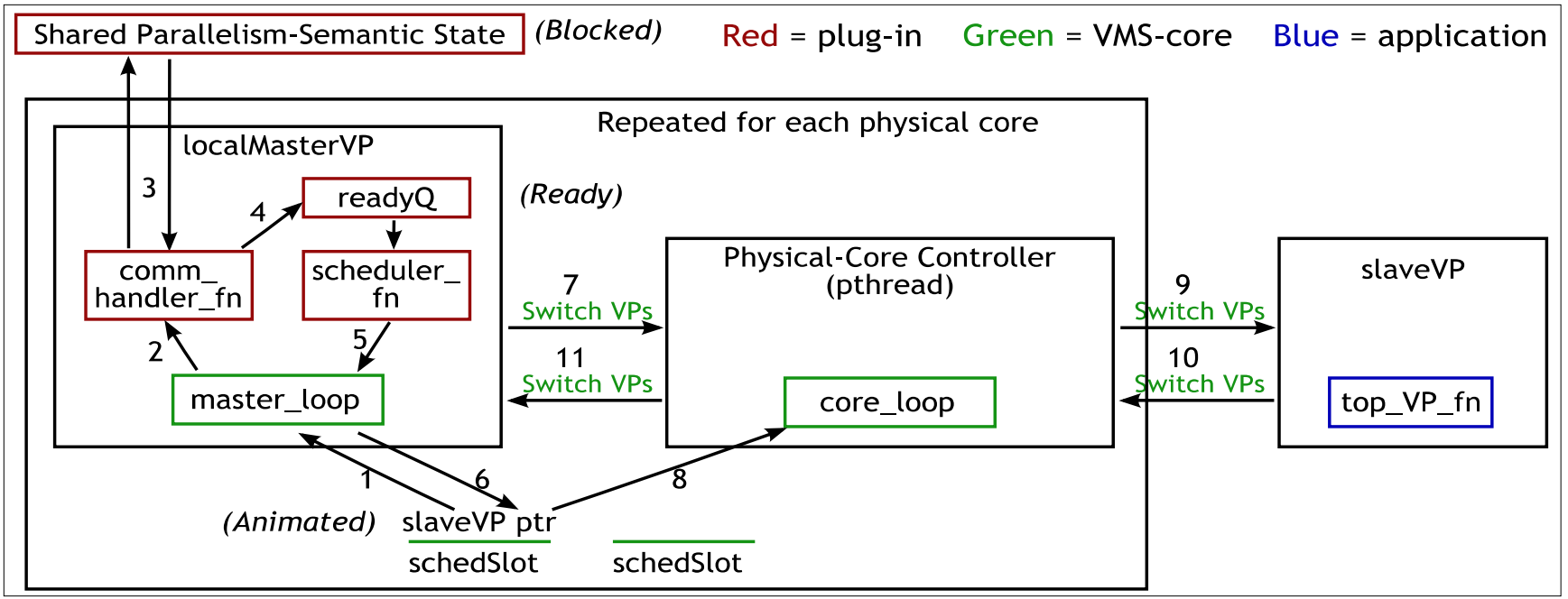




- App-code runs in a *Slave VP*
- SlaveVP animated by a core
- Switch to VMS suspends Slave, ret. reanimates

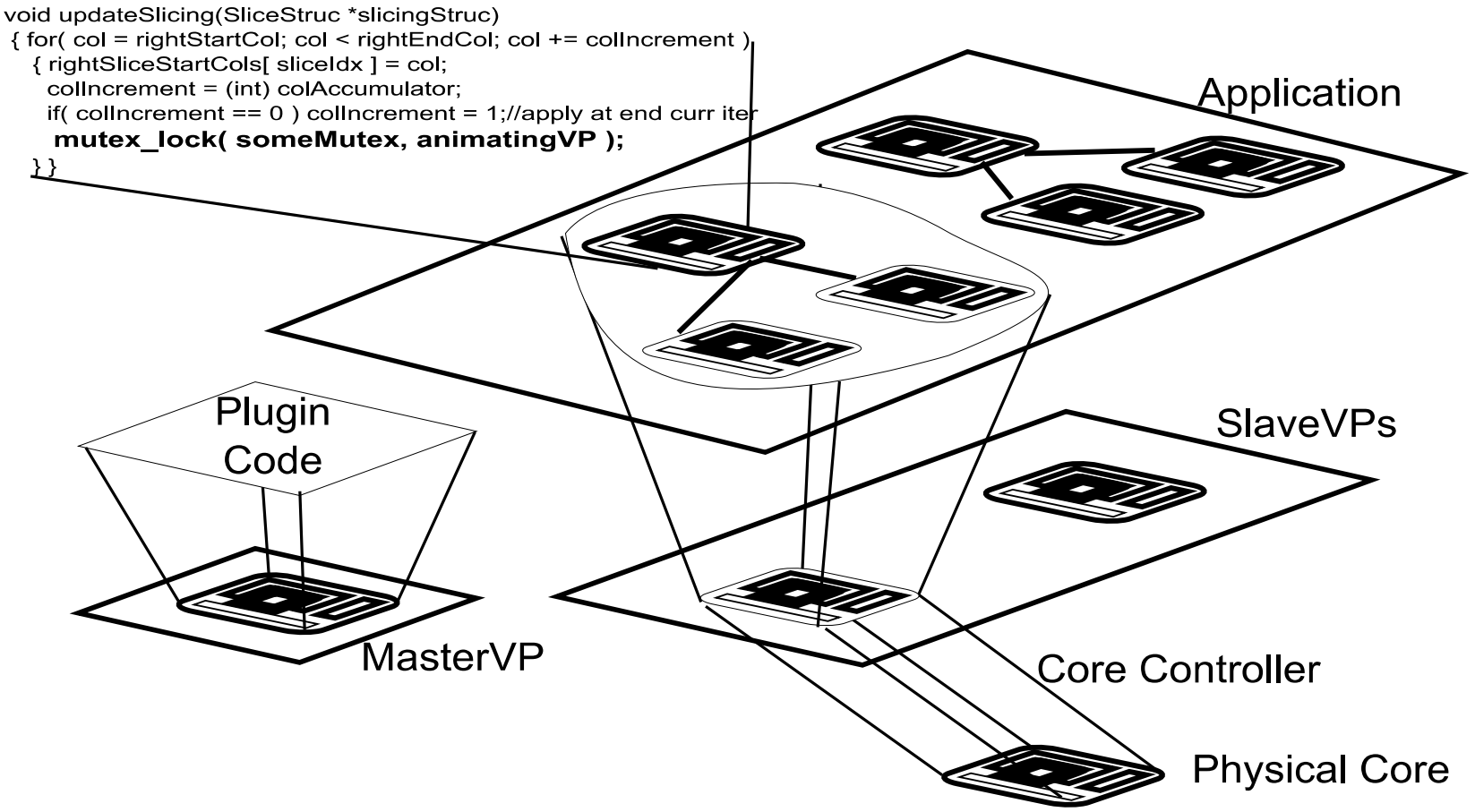
- Inside *this* implementation
  - One masterVP instance and one CoreContlr. per core
  - CoreContlr steers switching: equiv to HW mechanism
  - master_loop in masterVP, “organizes”, calls plugin
  - plugin == function-pointers given to masterVP

Elements in Action



- 1-6 are while localMasterVP animated on core
- Application is while SlaveVP animated on core
- MasterVPs create points in Virtual Time (state chgs)
- CoreCntlr is equivalent to HW support for switching
  - All VPs switch to CoreContlr, which controls switching

View of VMS System



- MasterVP lines up Slaves for the Core Controller
- Core Controller animated briefly while switching between VPs

Code

Usage in App
Wrapper Lib
Plugin comm-handler
(For lib-based lang embedded into C)

Usage in Application

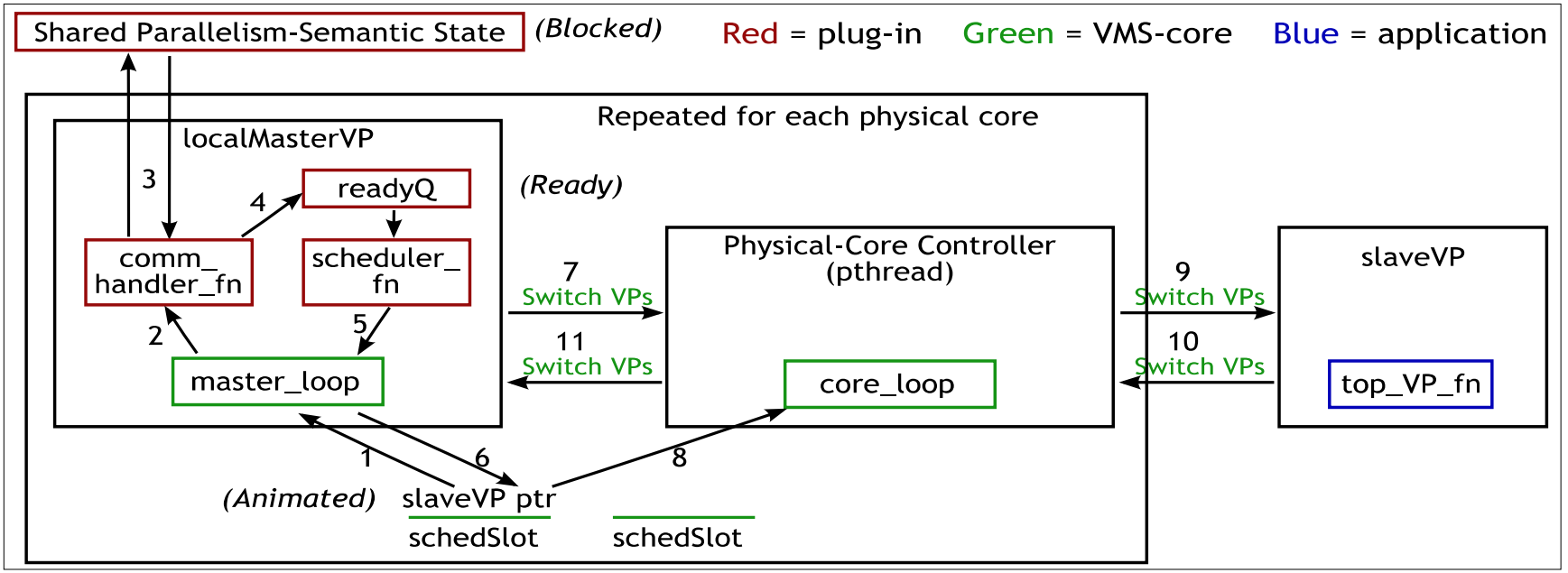
- Creating a new Virtual Processor (after translation from App language):

  top_VP_fn(void *parameterStrucPtr, VirtProcr *animatingVP); //fn to animate

  newProcessor = Vthread__create_thread(&top_VP_fn, paramsPtr, animatingVP);

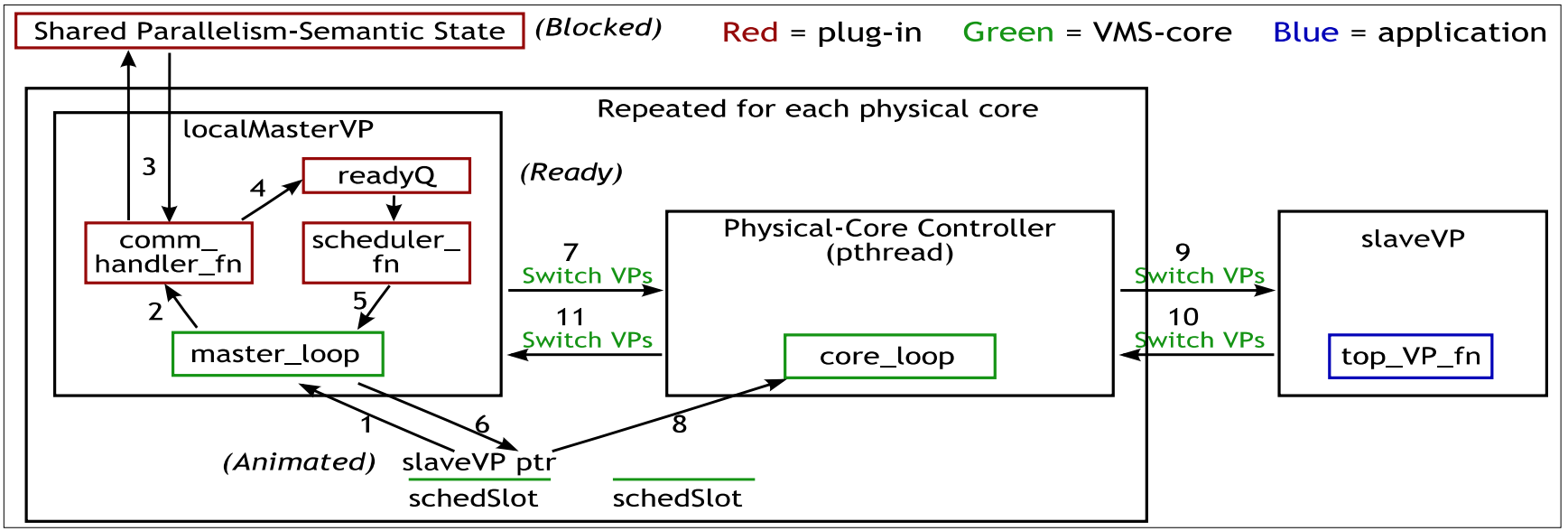
- Invoking parallelism construct (always hand it the animatingVP):

  Vthread__mutex_lock(mutexID, animatingVP);



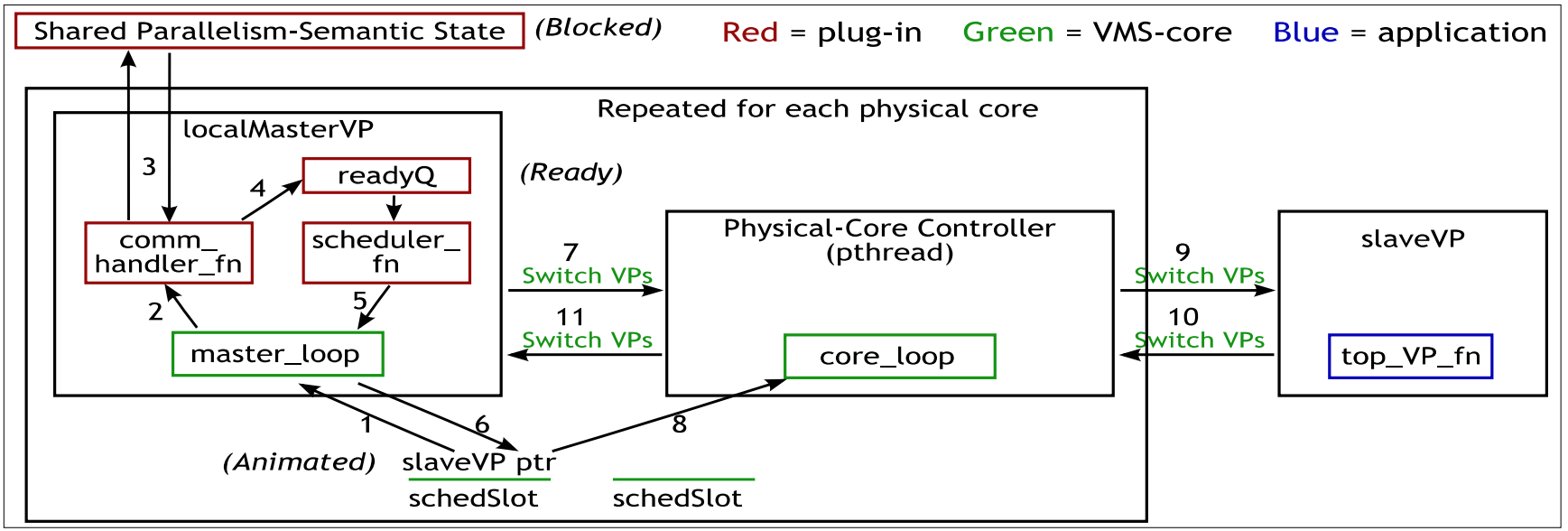
Wrapper Library

```
void * Vthread__mutex_lock( int mutexID, VirtProcr *animatingVP )
 { VthreadSemReq  reqData;
   reqData.reqType   = mutex_lock;
   reqData.mutexID = mutexID;
   VMS__send_sem_request( &reqData, animatingVP );
 }
```



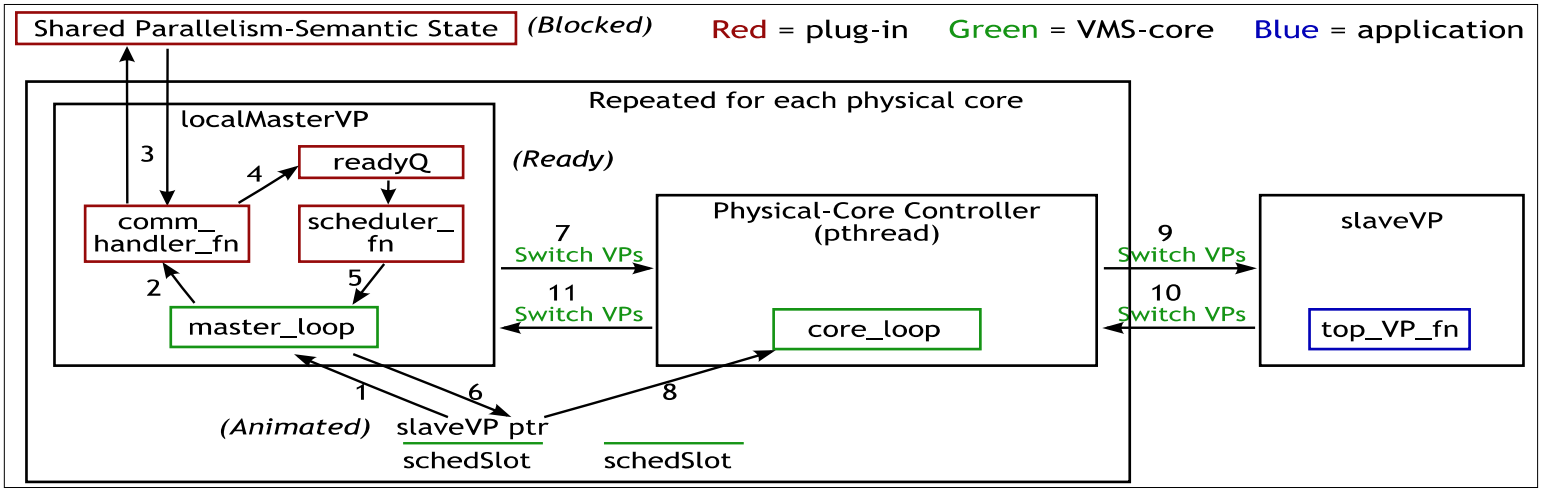
Plugin's Request Dispatcher

```
switch( semReq->reqType )
 { case make_mutex:    handleMakeMutex(  semReq, semEnv);       break;
   case mutex_lock:    handleMutexLock(  semReq, semEnv);       break;
   case mutex_unlock:  handleMutexUnlock(semReq, semEnv);       break;
```



Plugin Handler

```
handleMutexLock( VthreadReq *req, VthreadSemEnv *semEnv)
 { mutex = semEnv->mutexArray[ req->mutexID ];
   if( mutex->holderOfLock == NULL ) //none holding, give lock to requester
    { mutex->holderOfLock = semReq->requestingPr;
      make_ready( req->requestingPr, semEnv ); }
   else     //queue up requester to wait for release of lock
    { writePrivQ( req->requestingPr, mutex->waitingQueue ); }  }
```



Results: Time to Implement

- 2 days to conceive, design, implement, debug
  - Vs est. several weeks to modify traditional runtime

	SSR	Vthread	VCilk
Design	4	1	0.5
Code	2	0.5	0.5
Test	1	0.5	0.5
L.O.C.	470	290	310

Results: Overhead

- 2 orders magnitude less overhead vs pthreads
- local-only is when hit in L1 cache

	Vthread local only	Vthread remote	pthread
Mutex lock	85	1050	50,000
Mutex unlock	85	610	45,000
Cond wait	85	850	60,000
Cond signal	90	650	60,000

Results: Performance

- Comparable to Cilk 5.4
  - Effort to implement: 2 days vs several person-years
  - Point: 2 day rapid prototype has reasonable perf.

Matrix Size	VCilk	Cilk 5.4
81 x 81	0.008	0.017
324 x 324	0.13	0.13
648 x 648	0.85	0.71
1296 x 1296	6.2	4.8

Conclusion

- Virtualizes synchronization hardware
- Very low overhead
- Runtime fast and easy to write
- Fast to prototype parallel language constructs
- Encourages exploration of new lang constructs
- Provides necessary for portability
- Simplifies domain-specific languages
- Makes application-specific languages doable

Portability Motivation (2of2)

- Many HW targets under one interface
  - One interface variant for one class of hardware
  - All in class efficient with *same plugin* (def of class)
  - Reduces work of porting lang
- Runtime porting effort encapsulated inside VMS
  - Plugin encapsulates runtime (sync + scheduling)
    - Isolates scheduling from application-code
    - Reduces work of porting *language* to new HW
  - Clean way to insert auto-tuners, multi-versioned kernels, static-schedules generated by compiler

More on Sync Constructs

- Time-related semantics of sync == choose which Slave Blocked → Ready
  - Assigner (scheduler) chooses Ready → Animated
- Example: mutex_lock
  - Semantics: if lock free,caller Blocked → Ready, else queued, later Blocked → Ready
  - Choice of WHEN is the semantics – plugin provides
- VMS guarantees all physical events before call seen in other timelines as also before their call, and after as after.