A Mutable Hardware Abstraction to Replace Threads

Sean Halle, Albert Cohen

A Mutable Hardware Abstraction to Replace Threads

Sean Halle[1,2,3] and Albert Cohen[1]

[1] INRIA and École Normale Supérieure, France
[2] University of California at Santa Cruz, USA
[3] Technical University Berlin

Abstract. We propose an abstraction to alleviate the difficulty of programming with threads. This abstraction is not directly usable by application programmers. Instead, application-visible behavior is defined through a semantical plugin, and invoked via a language or library that uses the plugin. The main benefit is that parallel language runtimes become simpler to implement, because they use sequential algorithms for the parallel semantics. This is possible because the abstraction makes available a virtual time in which events in different program time-lines are sequentialized. The parallel semantics relate events in different time-lines via relating the sequentialized versions within the virtual time-line.
We have implemented the abstraction in user-space and demonstrate its low overhead and quickness to implement a runtime on three sets of parallelism constructs: rendez-vous style `send` and `receive`; Cilk style `spawn` and `sync`, which have similar performance to Cilk 5.4; and `mutex` and `condition variable` constructs from pthreads, which have 80x lower overhead than Linux thread operations. Development time averaged around two days per set, versus an expected duration of weeks to modify a thread-based runtime system.

1 Motivation

Thread parallelism constructs have been well documented to be difficult to program with. They directly expose low-level concurrency to the programmer. Arbitrary non-deterministic behavior and deadlocks can arise from improperly synchronized code. Efficient execution requires non-blocking algorithms whose correctness require deep understanding of weakly consistent memory models. In addition, the operating system abstraction for threads comes with a very high context-switching and synchronization overhead.

A partial solution. To deal with the last problem, a parallel language's runtime turns off operating system threads by pinning one to each physical core. This way, the custom runtime is assured that the software thread is one-to-one with a physical core. It then implements a user-level thread package that lets it control which OS thread a computational task is assigned to. Finally, the runtime then implements the language's parallel semantics in terms of those user threads.

The user-level threading approach addresses the system overhead issue, but it still hides important events such as input-output or node-to-node communications in a cluster. Hence, the more scalable runtimes need to coordinate task assignment to cores with application access of input and output, memory allocation over non-uniform cache and memory hierarchies, offloading to hardware accelerators, power management, and inter-node communication in a cluster. The user-level threading approach also makes the parallel runtime implementation cumbersome, error-prone and complex, because it is still written in terms of threads.

Overall, parallel language implementations must deal with a number of challenges normally deferred to the operating system, and they still suffer from the complexity of non-blocking shared memory concurrency.

Ideally, the OS would be in terms of a mutable hardware abstraction, and export mutations as new behavior. We define a mutable hardware abstraction to be an interface to hardware-level behaviors that are normally inside the OS or below it. Examples include communication between cores, allotting time-slots to applications, and establishing ordering of events among cores (which is what atomic memory operations and equivalent patterns of instructions do). The kernel itself would be implemented in terms of such an abstraction, and would accept mutations the same way it accepts device-drivers. It would then export the mutated behaviors for the language to trigger.

A language would implement its runtime behaviors in the form of a mutation. Being inside the OS, such a runtime has secure access to kernel-only hardware mechanisms as well as the hardware abstraction. It could negotiate with the kernel to manage physical resources, in a low-overhead way. One benefit of this arrangement is that the language runtime gains the ability to control which task is assigned to which processing element at what time. This enables high performance and low-energy data affinity techniques. For example, the runtime could track data within the memory hierarchy and assign tasks to locations close to their consumed data.

Contribution. We show in this paper the definition and implementation of such a mutable hardware abstraction, albeit at user-level rather than in the kernel. The abstraction lets a language's runtime be implemented as a mutation, which we call a plugin. The plugin implements parallelism constructs and assignment of tasks to cores.

We focus in this introductory paper on the definition of the abstraction and its support for parallelism constructs, postponing exploration of assignment of tasks onto cores and other performance optimizations to following papers. This paper defines multiple time-lines in a program, and a virtual timeline that globally orders events from them. It demonstrates three sets of parallelism constructs: synchronous `send-receive` motivated by process calculi; `spawn` and `sync` from Cilk [7, 10]; and `mutex` and `condition variable` from pthreads. The assignment policy we implemented with them is simply first-come first-served.

We call the abstraction Virtualized Master-Slave, or VMS. It exports facilities to create virtual processors and control how their timelines relate to each other, and relate to physical time. It also exports facilities to suspend a virtual processor and for an executable to interact with the plugin. The plugin embodies most of a language's runtime. A wrapper-library or keyword is what appears in application code, and is what triggers the runtime.

Organization of paper. Section 3 provide the original concepts and definitions of VMS. Section 4 focuses on the implementation, describing the elements and how they interact, then relating them back to the theoretical definition. Section 5 takes the point of view of the application code, studying the usage and implementation of parallel language constructs as a VMS plugin. To wrap up, measurements of effectiveness appear in Section 6 and conclusions in Section 7.

2 Background and Related Work

User-level thread packages and most parallel language runtimes have to side-step OS threads, by pinning one to each core, which effectively gives the user-level package control over the core. Our VMS implementation also does this. We are not claiming in this paper to have the OS level implementation of VMS that is possible – but just the user-space version.

Related work. The most primitive methods for establishing ordering among cores or distributed processors are atomic instructions and clock-synchronization techniques that time-stamp events [16, 4].

Meanwhile, the most closely related work is Scheduler Activations [2], which also allows modifying concurrency constructs and controlling assignment of virtual processors onto cores. However it has no virtual time to guarantee globally consistent sequentialization, and no interface for plugins.

BOM [6], which is used in Manticore to express scheduling policies and synchronization, also bears resemblances to VMS, but at a higher level of abstraction. BOM is a functional language, rather than a primitive abstraction meant to sit at the hardware-software boundary as VMS is.

Coroutines is a high-performance means of switching between tasks. Coroutine scheduling and stack handling techniques were well suited to the user-space implementation of VMS.

Other related work either provides an abstraction of the thread model, or is a full language with specific parallelism constructs. As a protypic example of user-level threads, Cilk [7, 10] provides a simplified abstraction with an efficient scheduling and load balancing algorithm, but limited to fork-join concurrency. OpenMP [18] is a typical example of a parallel extension of sequential languages; it allows creating tasks and controlling their execution order. We claim that both Cilk and OpenMP, as well as most thread abstractions or parallel languages may be implemented via plugins to VMS, with similar performance.

VMS is unique in that it doesn't impose its own concurrency semantics as a programming model, but rather takes preferred ones as plugins. This makes it only a *support* mechanism to implement language runtimes – VMS is hidden from the application, underneath the language. Parallelism constructs may be implemented as VMS plugins, easily, quickly, and with high performance as indicated in Section 6.

This work presents a first incarnation and evaluation of VMS. We plan to explore the embedding into VMS of a variety of parallel languages, with a special interest for coordination languages [8, 15, 5]. We will also explore VMS's compatibility with different concurrent semantics [14, 13, 17, 12, 1]. One particularly important application would be to use VMS to facilitate the design and implementation of the emerging hybrid programming models, such as MPI+OpenMP, or OpenMP+OpenCL [3, 9].

3 Abstract Definition of VMS

We start with an intuitive overview, then precise the definitions and properties in the following sub-sections.

Definitions: 1) We want to avoid the confusion associated with the various interpretations for the terms "thread" and "task" so will use the term *Virtual Processor* (VP), which we define as state in combination with the ability to animate code or an additional level of virtual processors. The state consists of a program counter, a stack with its contents, a pointer to top of stack, and a pointer to the current stack frame. 2) A *physical processor* executes a sequential stream of instructions. 3) A program-timeline is the sequence of instructions animated by a Slave VP, which is in turn animated by a physical processor.

Intuitive Overview. VMS can be understood via an analogy with atomic instructions, such as Compare and Swap (CAS). These are used to establish an ordering among the timelines of cores. They consist of two parts: 1) the semantics of what is done to the memory location, 2) a mechanism

that establishes an ordering among the cores. For CAS, the semantics are: "compare value in this register to value at the address, and if same, then put value in second register into the address." Multiple kinds of atomic instructions share the same order-establishing mechanism, they simply provide different semantics as a front-end.

VMS can be viewed as virtualizing the order-establishing mechanism. It allows the semantics to be plugged-in to it. This breaks concurrency constructs into two parts: the VMS mechanism, which establishes an ordering between events in different timelines; and the plugin, which supplies the semantics.

Below the interface, hardware mechanisms are employed to order specific points in one physical processor's timeline relative to specific points in another's timeline. Above the interface, a plugin provides the semantics that an application uses to invoke creation of the ordering.

Together, VMS plus the plugin form a parallelism construct, by which an application controls how the time-lines of its virtual processors relate. Such constructs also guarantee relations of VP time-lines to hardware events.

As an example, consider a program where one VP writes into a data structure then calls a `send` construct. Meanwhile, a different VP calls the `receive` construct then reads the data structure. The semantics of the `send` and `receive` constructs are that all data written before the `send` is readable in the other time-line after the `receive`. To implement these constructs, VMS provides the mechanism to enforce the ordering, and to include the writes and reads in that ordering. The plugin directs that mechanism to order the `send` event before the `receive` event.

What the VMS interface provides: The interface provides primitive operations to create and suspend VPs; a way for plugins to control when and where each VP is (re)started; a way for application code to send requests to the plugin; and a way to order a specific point in one VP time-line relative to a specific point in another VP time-line. All implementations of the VMS interface provide these, whether it is on shared memory or distributed, with strong memory consistency or weak.

Specification in three parts. We specify the observable behavior of a VMS system *with plugins present*. Hence, the specified behaviors remain valid with any parallelism construct implementable with VMS. First we give the specification of a computation system that VMS is compatible with; then specify a notion of time and the key VMS guarantee; and lastly specify virtual processor scheduling states and transitions between them.

3.1 The Specifications for a VMS-compatible Computation System

- An application creates multiple VPs, which are Slaves, each with an independent time-line.
- A schedule of Slaves is generated by a Master entity, from within a hidden time-line(s).
- A schedule is the set of physical locations and time-points at which Slaves are (re)animated.
- All semantic parallelism behavior is invoked by Slaves communicating with the Master.
- A Slave communicates with the Master by using a VMS primitive, which suspends the Slave.

Where we define: Semantic Parallelism Behavior is the actions taken by a parallelism construct, which establishes an ordering among events in different Slave timelines.

Discussion: The key point is that *scheduling is separated from the application code.* This is enforced by the schedule being generated in a time-line hidden from the application. The rest of the requirements are consequences of that separation.

The Master entity appears as a single entity, to the slaves. However it may be implemented with multiple (hidden) timelines. This is the approach taken in our initial implementation, which has several Master VPs hidden inside the VMS implementation.

3.2 The Time-Related Specifications of VMS

To prepare for the time-related specifications, we give an advance peek of the following section, 3.3. There, Slave VPs are specified to have three scheduling states: Animated, Blocked, and Ready. When a parallelism construct starts execution, the Slave transitions from Animated to Blocked. When it ends execution, the Slave transitions from Blocked to Ready. VMS provides a way to control the order of these state-transitions, which is equivalent to controlling the order of the parallelism-constructs. Controlling the state transitions is how the ordering among constructs in different timelines is established.

With that background, here are time-related specifications for VMS:

- VMS provides a Virtual timeline that globally orders changes of scheduling state of Slave VPs.
- Ordering is created among construct-invocations by controlling the order of Blocked to Ready transitions in the Virtual timeline.
- Causally tied construct-invocations are tied-together inside the Master.
- VMS enforces ordering of *observations* of physical events in Slave timelines to be consistent with the Virtual time ordering.
- Virtual time defines only ordering, but not spans, nor widths.

Discussion: Most importantly here, Virtual time defines a global ordering among Slave state-transitions. To make this useful for parallelism, VMS must be implemented so that observations of physical events, like reads and writes to shared memory, are consistent with that ordering.

The Virtual timeline plays the same role as the mechanism added to memory systems to support atomic instructions. All atomic instructions require hardware that establishes an ordering among the timelines of physical cores. That hardware sequentializes execution of atomic memory accesses to the same address. VMS virtualizes this mechanism. It provides the same ordering function.

An important point is that the Virtual timeline is generated inside the Master. When a Slave uses the VMS primitive to send a parallelism-construct request, it suspends. However, that Slave doesn't actually transition state from Animated to Blocked until the Master *acknowledges* the suspension. It is the acknowledgement that adds the Slave transition into the Virtual timeline.

The essential value of VMS is using it to control the order of observing events. It has to be able to causally link the execution of a parallelism construct in one timeline to the execution of a construct in a different timeline. Establishing such a causal link is called *tying together* two construct executions. It is specific executions from different timelines that are causally linked with such a tie.

The key VMS guarantee: the order of observing physical events is consistent with the order of tied together parallelism constructs.

To explain this, take as given: two Slaves both execute parallelism constructs, those are tied together by the Master, establishing a causal ordering from one to the other. So, one construct is the *before*-construct, the other is the *after*-construct. Now, the guarantee means that any events

triggered before the before-construct, in its timeline, are guaranteed to be detected in the other timeline as also preceding the after-construct. In addition, events triggered after the after-construct are guaranteed not visible before the before-construct in its timeline. This two-part guarantee is the result of the above specifications of VMS's time-related behavior.

Definitions: Some more definitions, to prepare for the next explanation. 1) An *ordering-point* exists in a Slave VP's timeline as a zero-width event that can be tied to ordering points in other timelines. It is initiated by a Slave VP executing the suspend primitive, and ended by the Master transitioning the Slave back to Animated. 2) A trace-segment is a portion of a Slave VP's stream of instructions bounded by ordering-points.

Hence, the timeline of a Slave is a sequence of trace-segments. Each trace-segment is animated by a single physical processor, but not necessarily the same as animated the Slave's other trace segments.

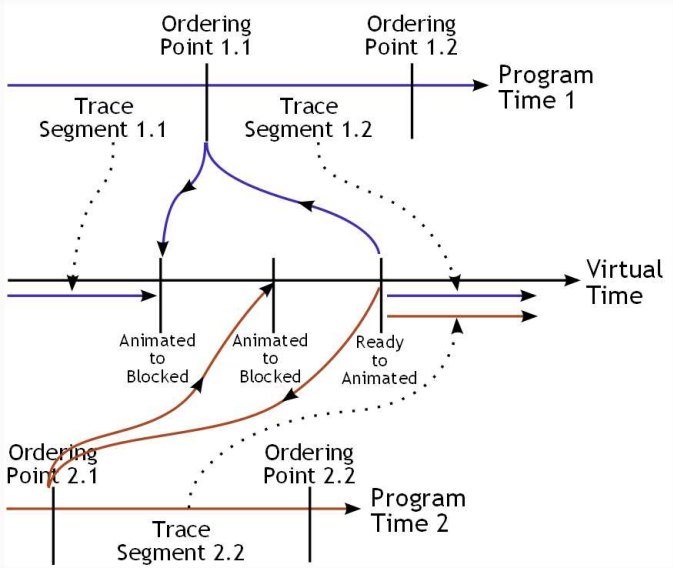

Fig. 1. Time Behaviors: Shows Ordering Point 1.1 being tied to Ordering Point 2.1. As a result, VMS guarantees that events triggered in Trace Segment 1.1 are seen as having taken place in the past in Trace Segment 2.2. Also shows that there is no common tied ordering point between segments 1.2 and 2.2, so VMS provides no guarantees about what order segment 2.2 sees events triggered in segment 1.2.

Relating time-lines to each other. Figure 1 shows two ordering points being tied together. A trace segment starts when an ordering-point is ended by its Slave transitioning to Animated. Because the transition to Animated exists as a point in Virtual time, the start of a trace-segment has a well-defined position within Virtual time. Likewise, a trace-segment is ended by its Slave executing the suspend primitive of VMS. Although this does not have a well-defined point in Virtual time, every execution of suspend is acknowledged by the Master, which transitions the Slave to Blocked. That

transition does have a well-defined position in Virtual time. Hence, the end of every trace-segment is associated with a well-defined position in Virtual time.

As a result, trace segments can be ordered relative to each other, by checking their start and end points in Virtual time. If they have no overlap in virtual time then they have a total ordering. However, if any portion of them overlaps in Virtual time, then they are considered concurrent trace-segments, and their Slaves are considered to be executing in parallel between those points of Virtual time.

Note that this is conservative because it doesn't take into account the physical wait time between a Slave suspending and the Master acknowledging. The Slave may stop executing at a physical time-point that would map onto an earlier point in Virtual time. In some cases, ending the Slave's trace-segment at the earlier point would eliminate the overlap with a particular other trace-segment. But VMS's set of specifications doesn't allow such mapping of suspend-execution onto Virtual time (for implementation-related reasons, which require downloading the code and gaining experience with it, to establish a common language, for an explanation to be understood).

A subtlety is that events triggered before one tied ordering-point, *might* be visible in the other timeline before the other tied ordering-point. In the figure, segment 2.1 might be able to see events from segment 1.1 if it looked. The VMS guarantee doesn't cover overlapped trace-segments. Physical events triggered before are only guaranteed visible *after* the tie point, and events after are only guaranteed *not* visible *before* the tie point.

We call this bounded non-determinism, because events within overlapped trace-segments have non-deterministic ordering, but the region of non-determinism can be bounded by tied ordering-points. This allows a program to specify non-determinism, but control the region of non-deterministic behavior. For example, a reduction construct could be created that non-deterministically assigns portions of the reduction work to overlapped Slave segments. It would tie together ordering points from all the Slaves that mark the end of reduction. Hence, the outcome is deterministic, but the path to get there is not.

Sequential algorithms for parallel constructs. The globally-consistent sequential order in Virtual time enables one of VMS's main benefits: sequential algorithms for parallel constructs. An implementation to tie ordering points together equals an implementation of parallel constructs. A plugin has an ordering of state transitions available, and chooses from those. Sequential algorithms rely on an ordering existing, while concurrent algorithms must include operations that establish an ordering. Plugins have Virtual time ordering available, so they can use sequential algorithms.

3.3 Specification of Scheduling State

Scheduling state is used in VMS to organize internal activity, for enforcing the guarantees.

- VPs have three scheduling states: *Animated*, *Blocked*, *Ready*; see Figure 2.
- VPs in Animated are *allowed* to advance Program time with (core-local) physical time.
- VPs in Blocked and Ready do not advance their Program time.
- Animated has two physical states: *Progressing* and *Stalled*.
- VPs in Progressing advance Program time with (core-local) physical time, those in Stalled do not (allowing non-semantic suspend for hardware interrupts).
- Scheduling states are defined in Virtual time only.
- Progressing and Stalled are defined in (core-local) physical time only; the distinction is invisible in Virtual time.

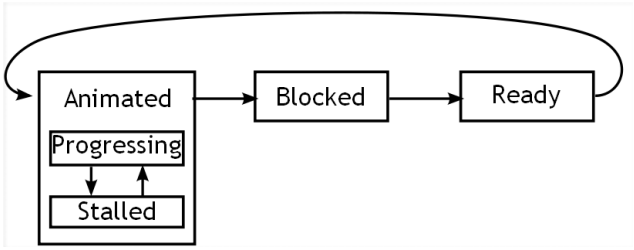

Fig. 2. Scheduling states of a slave VP in the VMS model. Animated, Blocked, and Ready are only defined in Virtual Time and only visible in the Master. Progressing and Stalled are only visible in physical-processor local time, not visible in Virtual time.

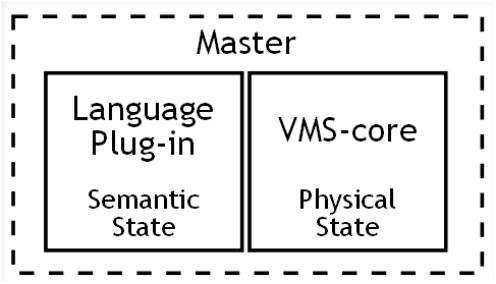

Fig. 3. The Master, split into a generic core and a language-specific plugin. The core encapsulates the hardware and remains the same across applications. The plugin implements the semantics of the parallelism-constructs.

Some important points: (1) only VPs Animated can trigger physical events that are seen in other program time-lines; (2) the distinction between Blocked vs Stalled is that a Slave has to explicitly execute a VMS primitive operation to enter Blocked. In contrast, Stalled happens invisibly, with no effect on semantic behavior. It is due to hardware events hidden inside VMS, such as interrupts.

The Ready state is used to separate the parallelism-construct behavior from the scheduling behavior. It acts as a "staging area" for scheduling. VPs placed into this state are *ready* to be animated, but the scheduler decides when and where.

An interesting point is that in VMS, the causal tie between timelines is created by actions *outside* program timelines. In contrast, memory-based lock algorithms place the concurrency-related behavior *inside* program timelines.

Transition Between Slave Scheduling States.

- VPs transition states as shown in Figure 2.
- Animated→Blocked is requested by a Slave executing suspend, but takes place in Virtual time at the point the Master acknowledges that request.
- Blocked→Ready is determined by the semantics implemented in the plugin.
- Ready→Animated is determined by the scheduler in the plugin.
- Transitions in scheduling state have a globally consistent order in Virtual time.

The parallelism primitives executed by a program do not directly control change in scheduling states. Rather they communicate messages to the Master, via a VMS supplied primitive. If it suspended when sending the request, then the act of the Master acknowledging the request places the Animated→Blocked transition into Virtual time. Inside the Master, the plugin then processes the message. Based on contents, it performs changes in state from Blocked→Ready, creates new VPs, and dissipates existing VPs. Most communication from Slave to Master requires the Slave to suspend when it sends the message. A few messages, like creating new Slave may be sent without suspending.

The suspend primitive decouples local physical time from Virtual time. Execution causes immediate transition to Stalled in physical time, later the Master performs Animated→Blocked, fixing that transition in Virtual time. The only relationship is causality. This weak relation is what allows

suspension-points to be serialized in Virtual time, which in turn is what allows using sequential algorithms to implement parallelism constructs.

3.4 Plugins

The Master entity has two parts, a generic core part and a plugin (Figure 3). The core part of the Master is implemented as part of VMS itself. The plugin supplies two functions: the communication-handler and the scheduler, both having a standard prototype. The communication-handler implements the parallelism constructs, while scheduler assigns VPs to cores.

An *instance* of a plugin is created as part of initializing an application, and the instance holds the semantic and scheduling state for that run of the application. This state, combined with the virtual processor states of the slaves created during that application run, represents progress of the work of the application. For example, multi-tasking is performed simply by the Master switching among plugin instances when it has a resource to offer to a scheduler. The parallelism-semantic state holds all information needed to resume (hardware state, such as TLB and cache-tags is inside VMS).

4 Internal Workings of Our Implementation

We name the elements of our example implementation and describe their logical function, then relate them to the abstract model. We then step through the operation of the elements.

Elements and their logical function. As illustrated in Figure 4, our VMS implementation is organized around physical cores. Each core has its own *master virtual-processor*, `masterVP`, and a *physical-core controller*, which communicate via a set of scheduling slots, `schedSlot`. The Master in the abstract definition is implemented by the multiple `masterVP`s plus a particular plugin instance with its shared parallelism-semantic state (seen at the top).

On a given core, only one of: the core-controller, `masterVP`, or a slave VP, is animated at any point in local physical time. Each `masterVP` animates the same function, called `master_loop`, and each slave VP animates a function from the application, starting with the top-level function the slave is created with, and following its call sequence. The core controller is implemented here as a Linux pthread that runs the `core_loop` function.

Switching between VPs is done by executing a VMS primitive that suspends the VP. This switches the physical core over to the controller, by jumping to the start of the `core_loop` function, which chooses the next VP and switches to that (switching is detailed in Section 5 Figure 7).

Relation to abstract model. We chose to implement the Master entity of the model by a set of `masterVP`s, plus plugin functions and shared parallelism-semantic state. VMS consists of this implementation of the Master, plus the core-controllers, plus the VMS primitive libraries, for creating new VPs and dissipating existing VPs, suspending VPs, and communicating from slave VP to Master. In Figure 4, everything in green is part of VMS, while the plugin is in red, and application code appears as blue, inside the slave VP.

Virtual time in the model is implemented via a combination of four things: a `masterLock` (not shown) that guarantees non-overlap of `masterVP` trace-segments; the `master_loop` which performs transition Animated→Blocked; the `comm_handler_fn` which performs Blocked→Ready and the `scheduler_fn` which performs Ready→Animated. Each state transition is one step of Virtual

time; is guaranteed sequential by the non-overlap of `masterVP` trace segments; and is global due to being in parallelism-semantic state that is shared (top of Figure 4).

Transitions Progressing⇄Stalled within the Animated state are invisible to the parallelism semantics, the Master, and Virtual time, and so have no effect on the elements seen.

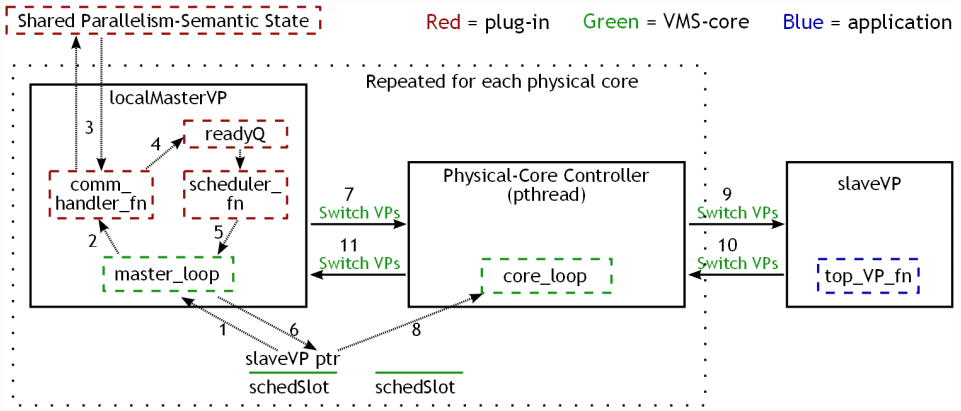

Fig. 4. Internal elements of our example VMS implementation

Steps of operation. The steps of operation are numbered, in Figure 4. Taking them in order:

1. `master_loop` scans the scheduling slots to see which ones' slaves have suspended since the previous scan.
2. It hands these to the `comm_handler_fn` plugged in (which equals transition Animated→Blocked).
3. The VP has a request attached, and data in it causes the `comm_handler_fn` to manipulate data structures in the shared parallelism-semantic state. These structures hold all the slaves in the blocked state (code-level detail and example will come in Figure 8, Section 5).
4. Some requests cause slaves to be moved to a `readyQ` on one of the cores (Blocked→Ready). Which core's `readyQ` receives the slave is under plugin control, determined by a combination of request contents, semantic state and physical machine state.
5. During the scan, the `master_loop` also looks for empty slots, and for each calls the `scheduler_fn` plugged in. It chooses a slave from the `readyQ` on the core animating `master_loop`.
6. The `master_loop` then places the slave VP's pointer into the scheduling slot (Ready→Animated), making it available to the `core_loop`.
7. When done with the scan, `masterVP` suspends, switching animation back to the `core_loop`.
8. `core_loop` takes slave VPs out of the slots.
9. Then `core_loop` switches animation to these slave VPs.
10. When a slave self-suspends, animation returns to the `core_loop` (detail in code in Figure 9), which picks another.
11. Until all slots are empty and the `core_loop` switches animation to the `masterVP`.

Enabling sequential implementation of parallelism semantics. All these steps happen on each core separately, but we use a central `masterLock` to ensure that only one core's `masterVP` can be active at any time. This guarantees non-overlap of trace-segments from different `masterVP`s, allowing the plugins to use sequential algorithms, without a performance penalty, as verified in Section 6.

Relating this to the abstract model: the parallelism-semantic behavior of the Master is implemented by the communication handler, in the plugin. It thus runs in the Master time referred to, in the model, in Section 3. Requests are sent to the Master by self-suspension of the slaves, but sit idle until the other slaves in the scheduling slots have also run. This is the passive behavior of requests that was noted in Section 3, which allows the `masterVP`s to remain suspended until needed. This in turn enables the `masterVP`s from different cores to be non-overlapped. It is the non-overlap that enables the algorithms for the parallelism semantics to be sequential.

5 Code Example

To relate the abstract model and the internal elements to application code and parallelism-library code, we give code snippets that illustrate key features. We start with the application then work down through the sequence of calls, to the plugin, using our SSR [11] parallelism-library as an example.

In general, applications are either written in terms of a parallel language that has its own syntax, or a base language with a parallelism library, which is often called an *embedded language*. Our demonstrators, VCilk [11], Vthread, and SSR, are all parallelism libraries. A parallel language would follow the standard practice of performing source-to-source transform, from custom syntax into C plus parallelism-library calls.

SSR. SSR stands for Synchronous Send-Receive, and details of its calls and internal implementation will be given throughout this section. It has two types of construct. The first, called *from-to* has two calls: `SSR_send_from_to` and `SSR_receive_from_to`, both of which specify the sending VP as well as the receiving VP. The other, called *of-type* also has two calls: `SSR__send_of_type_to` and `SSR__receive_of_type`, which allow a receiver to accept from anonymous senders, but select according to type of message.

Application view. Figure 5 shows snippets of application code, which use the SSR parallelism library. The most important feature is that all calls take a pointer to the VP that is animating the call. This is seen at the top of the figure where slave VP creation takes a pointer to the VP asking for creation. Below that is the standard prototype for top level functions, showing that the function receives a pointer to the VP it is the top level function for.

The pointer is placed on the stack by VMS when it creates the VP, and is the means by which the application comes into possession of the pointer. This animating VP is passed to all library calls made from there. For example, the bottom shows a pointer to the animating VP placed in the position of sender in the `send` construct call. Correspondingly, for the `receive` construct, the position of receiving VP is filled by the VP animating the call.

Relating these to the internal elements of our implementation, the `animatingVP` suspends inside each of these calls, passing a request (generated in the library) to one of the `masterVP`s. The `masterVP` then calls the `comm-handler` plugin, and so on, as described in Section 4.

For the `SSR__create_processor` call, the comm-handler in turn calls a VMS primitive to perform the creation. The primitive places a pointer to the newly created VP onto its stack, so that when

Creating a new processor:

```
newProcessor = SSR__create_procr( &top_VP_fn, paramsPtr, animatingVP );
```

prototype for the top level function:

```
top_VP_fn( void *parameterStrucPtr, VirtProcr *animatingVP );
```

handing animating VP to parallelism constructs:

```
SSR__send_from_to( messagePtr, animatingVP, receivingVP );
messagePtr = SSR__receive_from_to( sendingVP, animatingVP );
```

Fig. 5. Application code snippets showing that all calls to the parallelism library take the VP animating that call as a parameter.

`top_VP_fn` is later animated, it sees the VP-pointer as a parameter passed to it. All application code is either such a top-level function, or has one at the root of the call-stack.

The send and receive calls both suspend their animating VP. When both have been called, the communication handler pairs them up and resumes both. This ties time-lines together, invoking the VMS guarantee. Both application-functions know, because of the VMS guarantee (Section 3), that writes to shared variables made before the send call by the sender are visible to the receiver after the receive call. This is the programmer's view of tying together the local time-lines of two different VPs, as defined in Section 3.

Concurrency-library view. A parallelism library function, in general, only creates a request, sends it, and returns, as seen below. To send a request, it uses the combined request-and-suspend VMS primitive that attaches the request then suspends the VP. The primitive requires the pointer to the VP, to attach the request and to suspend it.

In Figure 6, notice that the request's data is on the stack of the virtual processor that's animating the call, which is the `receiveVP`. The `VMS__send_sem_request` suspends this VP, which changes the physical core's stack pointer to a different stack. So the request data is guaranteed to remain undisturbed while the VP is suspended.

Figure 7 shows the implementation of the VMS suspend primitive. As seen in Figure 4, suspending the `receiveVP` involves switching to the `core_loop`. In our implementation, this is done by switching to the stack of the pthread pinned to the physical core and then jumping to the start-point of `core_loop`.

This code uses standard techniques commonly employed in co-routine implementations. Tuning effort spent in `core_loop` is inherited by all applications.

Plugin view. SSR's communication handler dispatches on the `reqType` field of the request data, as set by the `SSR__receive_from_to` code. It calls the handler code in Figure 8. This constructs a hash-key, by concatenating the from-VP's pointer with the to-VP's pointer. Then it looks-up that key in the hash-table that SSR uses to match sends with receives, which is in the shared semantic state seen at the top of Figure 4 in Section 4.

The most important feature in Figure 8 is that both send and receive will construct the same key, so will find the same hash entry. Whichever request is handled first in Virtual time will see the

```
void * SSR__receive_from_to( VirtProcr *sendVP, VirtProcr *receiveVP )
 { SSRSemReq  reqData;
   reqData.receiveVP = receiveVP;
   reqData.sendVP    = sendVP;
   reqData.reqType   = receive_from_to;
   VMS__send_sem_request( &reqData, receiveVP );
   return receiveVP->dataReturnedFromRequest;
 }
```

Fig. 6. Implementation of SSR's receive_from_to library function.

```
VMS__suspend_procr( VirtProcr *animatingVP )
 { animatingVP->resumeInstrAddr = &&ResumePt; //GCC takes addr of label
   animatingVP->schedSlotAssignedTo->isNewlySuspended = TRUE; //for master_loop to see
   <assembly code stores current physical core's stack reg into animatingVP struct>
   <assembly code loads stack reg with core_loop stackPtr, which was saved into animatingVP>
   <assembly code jmps to core_loop start instr addr, which was also saved into animatingVP>
 ResumePt:
   return;
 }
```

Fig. 7. Implementation of VMS suspend processor. Re-animating the virtual processor reverses this sequence. It saves the `core_loop`'s resume instr-addr and stack ptr into the VP structure, then loads the VP's stack ptr and jmps to its `resumeInstrAddr`.

```
handle_receive_from_to( VirtProcr *requestingVP, SSRSemReq *reqData, SSRSemEnv *semEnv )
 { commHashTbl = semEnv->communicatingVPHashTable;
   key[0] = reqData->receiveVP;   key[1] = reqData->sendVP; //send uses same key
   waitingReqData = lookup_and_remove( key, commHashTbl );  //get waiting request
   if( waitingReqData != NULL )
    { resume_virt_procr( waitingReqData->sendVP );
      resume_virt_procr( waitingReqData->receiveVP ); }
   else
      insert( key, reqData, commHashTbl ); //receive is first to arrive, make it wait
 }
```

Fig. 8. Pseudo-code of communication-handler for `receive_from_to` request type. The `semEnv` is a pointer to the shared parallel semantic state seen at the top of Figure 4.

hash entry empty, and save itself in that entry. The second to arrive sees the waiting request and then resumes both VPs, by putting them into their `readyQ`s.

Access to the shared hash table can be considered private, as in a sequential algorithm. This is because our VMS-core implementation allows only one handler on one core to execute at any time.

6 Results

We implemented blocked dense matrix multiplication with right sub-matrices copied to transposed form, and ran it on a 4-core Core2Quad 2.4Ghz processor.

Implementation-time. As shown in Table 1, time to implement the three parallel libraries averages 2 days each. As an example of productivity, adding nested transactions, parallel singleton, and atomic function-execution to SSR required a single afternoon, totaling less than 100 lines of C code.

Execution Performance. Performance of VMS is seen in Table 2. The code is not optimized, but rather written to be easy to understand and modify. The majority of the plugin time is lost to cache misses because the shared parallelism-semantic state moves between cores on a majority of accesses. Acquisition of the master lock is slow due to the hardware implementing the CAS instruction.

Existing techniques may improve performance, such as localizing semantic data to cores, splitting malloc across the cores, pre-allocating slabs that are recycled, and pre-fetching. However, in many cases, several hundred nano-seconds per task is as optimal as the applications can benefit from.

	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

Table 1. Person-days to design, code, and test each parallelism library. L.O.C. is lines of (original) C code, excluding libraries and comments.

		comp only	comp + mem
VMS Only	`master_loop`	91	110
	switch VPs	77	130
	(malloc)	160	2300
	(create VP)	540	3800
Language:			
SSR	plugin – concur	190	540
	plugin – all	530	2200
	lock		250
Vthread	plugin – concur	66	710
	plugin – all	180	1500
	lock		250
VCilk	plugin – concur	65	260
	plugin – all	330	1800
	lock		250

Table 2. Cycles of overhead, per scheduled slave. "comp only" is perfect memory, "comp + mem" is actual cycles. "Plugin-concur" only concurrency requests, "plugin-all" includes create and malloc requests. Two significant digits due to variability.

Matrix size	Lang.	sec.
81 × 81	Cilk	0.017
	VCilk	0.008
324 × 324	Cilk	0.13
	VCilk	0.13
648 × 648	Cilk	0.71
	VCilk	0.85
1296 × 1296	Cilk	4.8
	VCilk	6.2

operation	Vthread		pthread	ratio
	comp only	total		
mutex_lock	85	1050	50,000	48:1
mutex_unlock	85	610	45,000	74:1
cond_wait	85	850	60,000	71:1
cond_signal	90	650	60,000	92:1

Table 3. On left, time in seconds for MM. To the right, overhead for pthread vs. Vthread. First column: cycles for perfect memory. Second: total measured cycles. pthread cycles are from round-trip experiments.

Head to head. We compare our implementation of the `spawn` and `sync` constructs against Cilk 5.4, on the top in Table 3, which shows that the same application code has similar performance. For large matrices, Cilk 5.4's better use of the memory hierarchy (the workstealing algorithm) achieves 23% better performance. However, for small matrices, VCilk is better, with a factor 2 lower overhead. Cilk 5.4 does not allow controlling the number of spawn events it actually executes, and chooses to run smaller matrices sequentially, limiting our comparison.

When comparing to pthreads, our VMS based implementation has more than an order of magnitude better overhead per invocation of mutex or condition variable functionality, as seen on the bottom of Table 3. Applications that inherently have short trace segments will synchronize often and benefit the most from Vthread.

7 Conclusion

We have shown an alternative to the thread model that enables easier-to-use parallelism constructs by splitting the scheduler open, to accept new parallelism constructs in the form of plugins. This gives the language control over assigning virtual processors to physical cores, for performance, debugging, and flexibility benefits. Parallelism constructs of programming languages can be implemented using sequential algorithms, within a matter of days, while maintaining low run-time overhead.

Acknowledgments This work was partly funded by the European FP7 project TERAFLUX.

References

1. G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.
2. Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10:53–79, February 1992.
3. Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In *IWOMP'10*, pages 1–14, 2010.
4. F. Christian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.
5. Intel Corp. CnC homepage. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.
6. Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 37–44, 2007.
7. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation*, pages 212–223, Montreal, Quebec, June 1998.
8. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.
9. Kronos Group. Open Compute Language homepage. http://www.khronos.org/opencl.
10. Cilk group at MIT. CILK homepage. http://supertech.csail.mit.edu/cilk/.
11. Sean Halle. VMS home page. http://vmsexemodel.sourceforge.net.
12. Carl Hewitt. Actor model of computation, 2010. http://arxiv.org/abs/1008.1459.
13. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

14. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland, Amsterdam.
15. Kathleen Knobe. Ease of use with concurrent collections (CnC). In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.
16. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.
17. Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.
18. OpenMP organization. OpenMP home page. http://www.openmp.org.