Peredvizhnikov Engine: Design and Implementation of a Completely Lock-Free Scheduler

by Eduard Permyakov
<edward.permyakov@gmail.com>

August 2023
Version 1.0.0

1.0 Background and Motivation

As Moore's Low continues to hold up less and less and gains in single-core performance continue diminishing year-over-year, chip manufacturers are instead turning to adding more cores to continue delivering more computing power. The Apple M2 comes with up to 12 CPU cores and the 12th generation Intel processors sport up to 16 cores. As such, it becomes increasingly more important to write highly parallel programs in order to efficiently utilize the available computing power.

In addition, the performance advances of GPUs have greatly outpaced those of CPUs in recent years. Applications are having a progressively harder time saturating the GPU with work, as the cost of deriving the GPU commands and sending them to the graphics card becomes an ever more prominent bottleneck. This has driven the development of new graphics programming APIs such as Direct3D 12 and Vulkan, which are highly asynchronous in their nature and, among other benefits, aim to allow the graphics programmer to utilize the available parallelism found in modern CPUs.

Game engines, due to their need to deliver high-fidelity graphics and accurate simulations of large worlds in real-time, are some of the most performance-hungry software available. It is oftentimes the case that game engines are at the forefront of bringing new ideas about high-performance programming into the mainstream. In the days of single-core CPUs, games would run in a single thread, typically with a top-level event loop that would poll for events from the OS, perform one tick of simulation and render the frame. As multicore processors became more commonplace, it became fashionable to split rendering and simulation into separate threads, thereby pipelining the work of a single frame. In addition, certain tasks such as file IO could be put into background threads, to be scheduled in and out by the OS and thus not delay the critical processing of a particular frame. In more recent years, the trend has evolved to job and fiber systems which use a userspace scheduler to execute a directed acyclic graph of tasks on a pool of OS threads for each frame.

Peredvizhnikov Engine aims to be a performant, highly-parallel framework for games and, more generally, any applications which require extracting the maximum benefit from the system's CPU and GPU resources. Compelled by the trend of growing parallelism, the engine adopts the *actor model* of concurrent computation, using userspace tasks, or *actors*, as the building blocks for high-level simulation logic. Instead of using error-prone monolithic logic, complex algorithms can be realized with a set of tasks which encapsulate their private state and can interact with other tasks via *messages* and *events*, which are, in essence, asynchronous multicast messages. For example, a rendering engine can have a large number of small tasks such as *Engine*, *BufferUploader*, *FileReader*, *CameraController*, *ShaderCompilationServer*, *PhysicsSimulator*, or *UIWindow*, all of which can be thought of as running independently of one another. The logic of simulating and rendering a frame will be realized as a chain reaction of messages and events sent from one actor to another. The advantages of such a programming model are manifold. Firstly, the actors can be transparently scheduled on an arbitrary number of cores, ensuring optimal saturation of CPU resources on any hardware. Moreover, such a design fits well with modern graphics APIs which have asynchronous submission and completion of GPU jobs. In addition, this model allows a high degree of flexibility in extending the engine's features, which can be as simple as adding additional actors to interact with already existing messaging protocols. The actor model is also highly fault-tolerant, as an error in a tightly encapsulated task doesn't need to halt the progress of the entire engine.

Besides all the aforementioned benefits, due to the fact that all state is private to some actor and changes or queries to it are made only via messages, the actor model removes the need for lock-based synchronization. Without tasks being blocked on acquiring locks, the engine is wholly immune to deadlock or priority inversion. It offers guaranteed progress even when a thread is arbitrarily terminated, or when a thread is preempted and halted for an arbitrary amount of time. This is especially critical for real-time applications such as games, which require the production of a new frame within a specified time budget.

2.0 Atomic Work

Atomic Serial Work is an abstraction used in realizing the higher-level lock-free algorithms used in Peredvizhnikov Engine's scheduler. It allows lock-free, atomic invocation of a group of instructions from a restricted set, ultimately a form of *lock-free serialization.*

It is based on the idea that it is possible to atomically write a function pointer to some memory location, execute the function, and overwrite the pointer location with a *null* pointer, thereby informing other threads that this operation has completed and letting them take their turn. By all means, this first draft of the algorithm is very much flawed. What if the owning thread is terminated mid-execution? The remaining threads would be left deadlocked. This basic idea must be developed further.

Inspiration can be drawn from classical lock-free data structures and algorithms. Even the most foundational structures such as Michael and Scott's lock-free queue or Harris' lock-free linked list don't modify the entire state of the structure with a single atomic instruction.

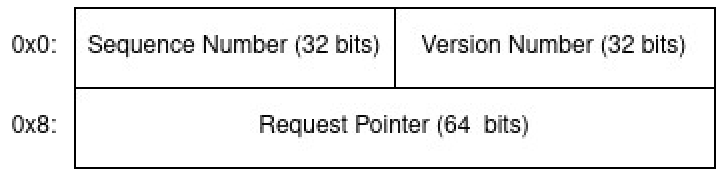
They employ the idea of *helping*, whereby a single atomic instruction executed by one thread irrevocably determines the future state of the entire structure and any number of threads can *help* in advancing the entire data structure to the pre-determined state. For example, in the Michael and Scott queue, two pointers stored in disjoint memory locations are employed, one for the head and one for the tail. While the head pointer is always pointing to the first node, the tail pointer can lag behind. Due to the linked-list structure of the queue, any thread performing an *enqueue* or *dequeue* operation can detect that the tail pointer has become desynchronized and swing it to the correct location. This fundamental idea can be applied to address the aforementioned deadlock problem. A single thread's atomic write of the function pointer will determine that this function will be completed by any number of threads. If any thread detects that there is an existing installed function pointer, it can *help* complete its' execution.

It is clear that the algorithm is still not fully-developed. How can a thread just "continue" the execution of some function that has already been partially completed by another thread? Moreover, the thread which wrote the function pointer can still be running on a different core, causing two different invocations of the function running in parallel. For the sake of developing the idea, it is helpful to step back temporarily and restrict the problem set to just *pure* functions (i.e. functions which just compute a value and don't have any observable side effects). In that case, the thread which detected the function pointer can simply invoke its' local copy of the function pointer and obtain the same result as the thread which installed the function pointer. Furthermore, it is possible to allow specifying an argument list for the function by adding an extra level of indirection. Instead of atomically installing a function pointer directly, rather install a pointer to a *request* structure, which will store the function pointer and the associated arguments. While, in principle, this algorithm would work, it suffers from very limited applicability.

There is one final leap that must be made in order to make the algorithm exponentially more useful. That is, the set of supported invocables must be broadened from that of just pure functions. Here the concept of *restartable functions* is defined as those functions which can be safely restarted any number of times on different threads and produce the same set of side effects. Restartable functions are a superset of pure functions and a subset of reentrant functions. Practically speaking, all side-effects produced by restartable functions are based on the *compare-and-swap* primitive, such that they modify a memory location only on the first pass, and leave it unchanged on subsequent passes. Furtheremore, restartable functions can make use of certain higher-level primitives based on the compare-and-swap instruction, namely inserting identical data into a lock-free set with the same key on each invocation, or publishing some data through an atomic shared pointer. In order to prevent duplicate side-effects from different invocations of a request on different threads, a *sequence number* associated with a request pointer is atomically derived and is passed to all invocations of that request. Inside the body of the installed function, which must be restartable, the sequence number can be used to prevent duplicate updates of the same state.

The final inter-thread protocol is as follows: Threads will use a CAS loop in order to attempt to install their request and, at the same time, atomically update the sequence number of the request. Furthermore, a version number associated with the request will be set in the same atomic operation, to address the *ABA problem*, essentially not being able to tell apart between two distinct request pointers when the memory associated with one request has already been recycled by the memory allocation system. The thread will signal completion by overwriting the request pointer with a *null* pointer. If a thread is unable to install its' request due to a competing request from a different thread already present, it will complete that request and retry installing its' own. If the request function is restartable, then any number of parallel invocations of the request will produce the same side-effects as a single invocation of the request on a single thread. On x86-64 processors which support the *CMPXCHG16B* instruction, 16 bytes can freely be used for the atomic control block. However, the same algorithm can be realized on other platforms such as AArch64 with frugal bit-packing. It must also be mentioned that it is necessary to guarantee *safe memory reclamation* when different threads access shared memory locations belonging to a request, for example, utilizing *hazard pointers*. Ultimately, using the aforementioned algorithm, if all installed functions are *restartable*, then the completion of any request is atomic and fully lock-free.

Figure 1: 128-bit structure of the restartable request atomic control block



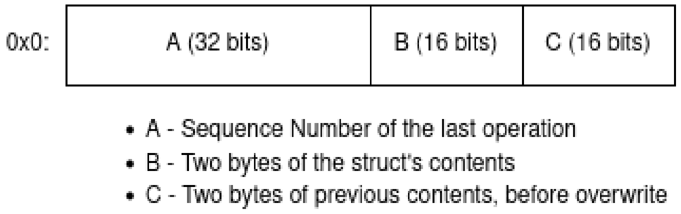
In addition, two related atomic primitives used for realizing more complex lock-free algorithms are introduced. *Atomic Parallel Work* allows any number of threads to concurrently process an array of values. The processing result for each value is published by inserting the result into a lock-free set. Again, competing threads can *steal* an outstanding request and *help* complete it, thereby making the entire operation atomic and lock-free. *Atomic Work Pipeline* is a set of *Atomic Parallel Work* invocations that are sequenced using *Atomic Serial Work*, in other words, a multi-stage vector version of *Atomic Serial Work*.

3.0 Atomic Struct

Atomic Struct allows atomic lock-free access to an arbitrary-sized block of memory, supporting the operations of *Load*, *Store*, *Exchange,* and *CompareExchange.* It is founded on the *Atomic Serial Work* primitive.

Each operation (*Load, Store, Exchange, CompareExchange)* is a type of request, the execution of which is serialized with an *Atomic Serial Work* instance. This means that any number of threads can concurrently be executing any number of requests, some of which may have already *completed.* However, a completion of a request on any one thread guarantees that all side-effects produced by that request are already visible. An executing request that has already been completed by a different thread is said to be *lagging*, and it will not produce any visible side-effects. The result of a request (ex. the read contents of the structure) is published by setting an atomic shared pointer, with a CAS loop to prevent duplicate overwrites.

In order to prevent overwrite of the structure's data by *lagging* requests, the contents of the structure are scattered into a set of *Sequenced Half Words*, each of which is updated only with an atomic compare-and-swap instruction.


0x0:
A (32 bits)
B (16 bits)
C (16 bits)
• A - Sequence Number of the last operation
• B - Two bytes of the struct's contents
• C - Two bytes of previous contents, before overwrite
Figure 2: Atomic Sequenced Half Word

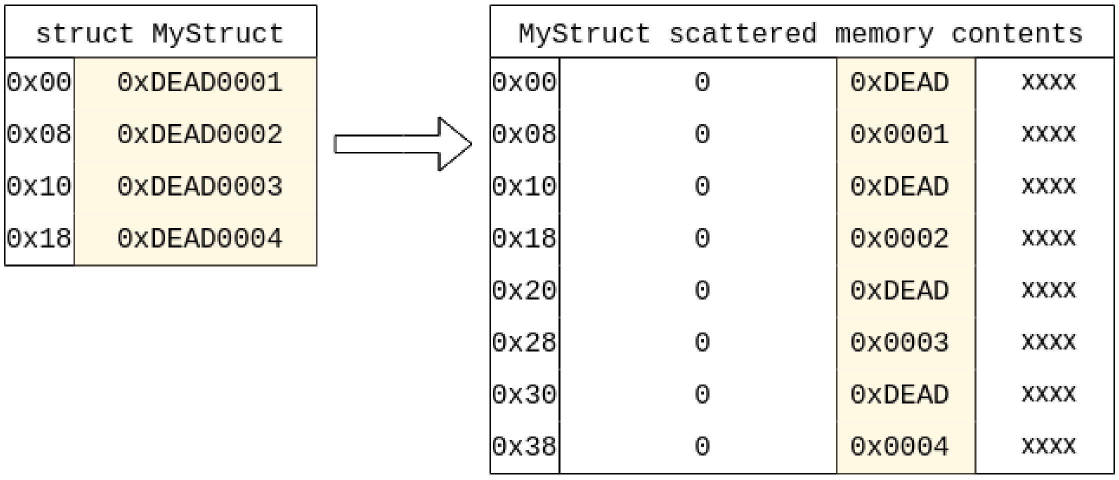


Figure 3: Layout of scattered structure contents in memory

struct MyStruct	
0x00	0xDEAD0001
0x08	0xDEAD0002
0x10	0xDEAD0003
0x18	0xDEAD0004

MyStruct scattered memory contents			
0x00	0	0xDEAD	xxxx
0x08	0	0x0001	xxxx
0x10	0	0xDEAD	xxxx
0x18	0	0x0002	xxxx
0x20	0	0xDEAD	xxxx
0x28	0	0x0003	xxxx
0x30	0	0xDEAD	xxxx
0x38	0	0x0004	xxxx

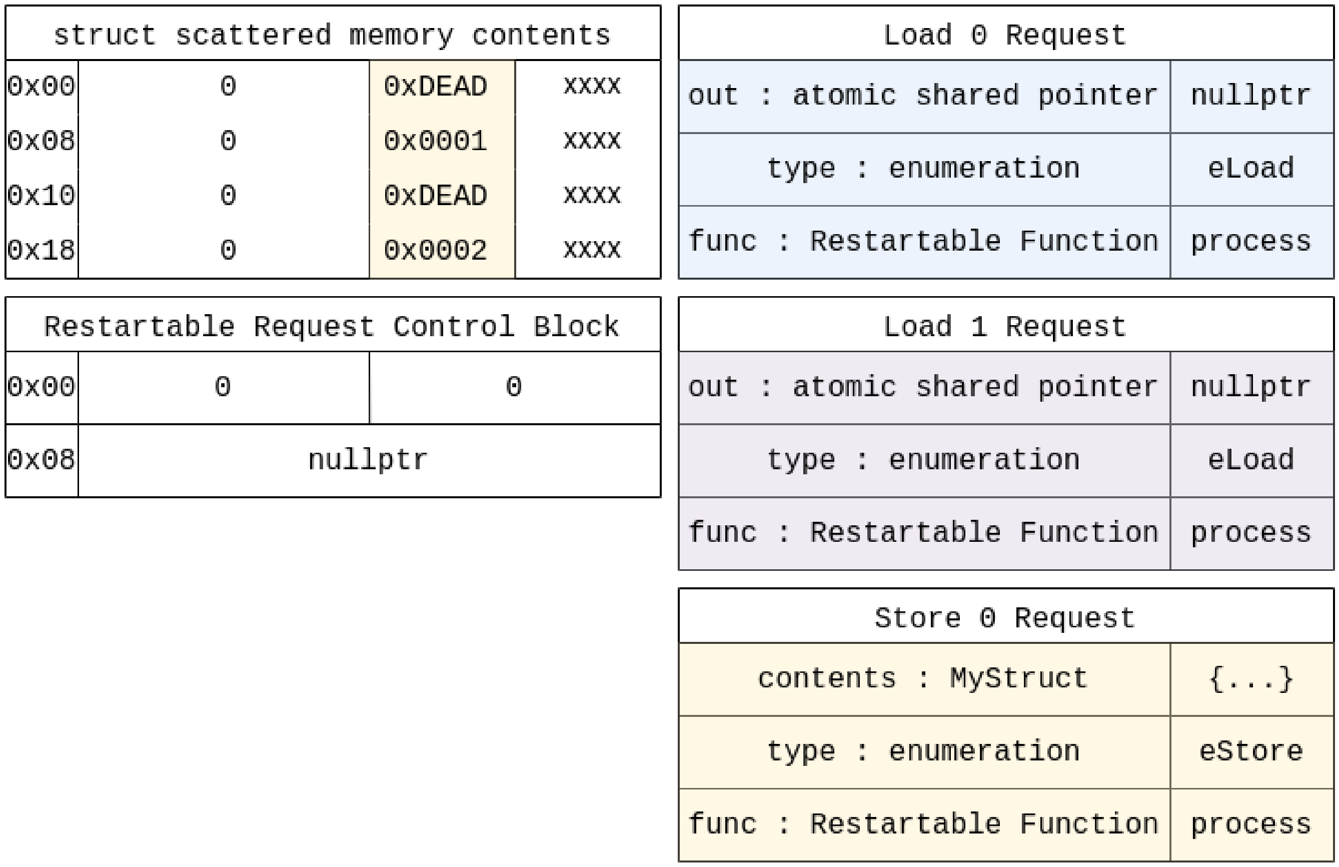
A *Load* request gathers the structure's data from the Sequenced Half Words. It will do so by atomically bumping the sequence number of each half-word to that of the current request with a compare-and-swap operation and extracting the previous half-word. If the sequence number has already been advanced by a concurrent thread executing the same request, the contents of the half-word can simply be read. If the sequence number has already passed the current request's sequence number, then it can be deduced that this request has already been completed by a different thread, and thus can be aborted. The gathered structure contents are published through an atomic shared pointer, to be consumed by the thread which generated the request.

A *Store* request scatters a new set of contents for the structure into the Sequenced Half Words. It will atomically overwrite the contents and advance the sequence number with a compare-and-swap instruction for each half-word. Again, the sequence number can be used to detect if the half-word has already been overwritten by a concurrent thread executing the same request, or if the current request has already completed.

An *Exchange* request atomically overwrites the structure's contents while returning the previously held contents. To achieve this, the restartable request function will iterate through the sequenced half words, advancing the sequence number, reading the prior contents, and setting the new contents atomically. When overwriting a half-word, it is also necessary to shift the prior contents of the half-word into the *previous* field of the Atomic Sequenced Half Word. The reason for this is to allow a thread to detect and read the prior contents of the half-word when a concurrent thread executing the same request has already overwritten it. A *CompareExchange* request atomically compares the structure's value to some *expected* value, and proceeds to overwrite the structure only if the data is equal to *expected.* Otherwise, it overwrites *expected* with the current value of the structure. The operation is similar to that of the *Exchange* request, except with an additional pass to read and compare the structure's contents before committing the write.

In order to fully appreciate the operation of the algorithm, it is helpful to step through a detailed example. Say there are three threads, *Thread 0*, *Thread 1*, and *Thread 2*, all racing to perform some operation on an *Atomic Struct. Thread 0* and *Thread 2* want to complete *Load* requests and *Thread 1* wants to complete a *Store* request.

Figure 4: Initial contents of relevant memory structures



struct scattered memory contents			
0x00	0	0xDEAD	xxxx
0x08	0	0x0001	xxxx
0x10	0	0xDEAD	xxxx
0x18	0	0x0002	xxxx

Restartable Request Control Block		
0x00	0	0
0x08	nullptr	

Load 0 Request	
out : atomic shared pointer	nullptr
type : enumeration	eLoad
func : Restartable Function	process

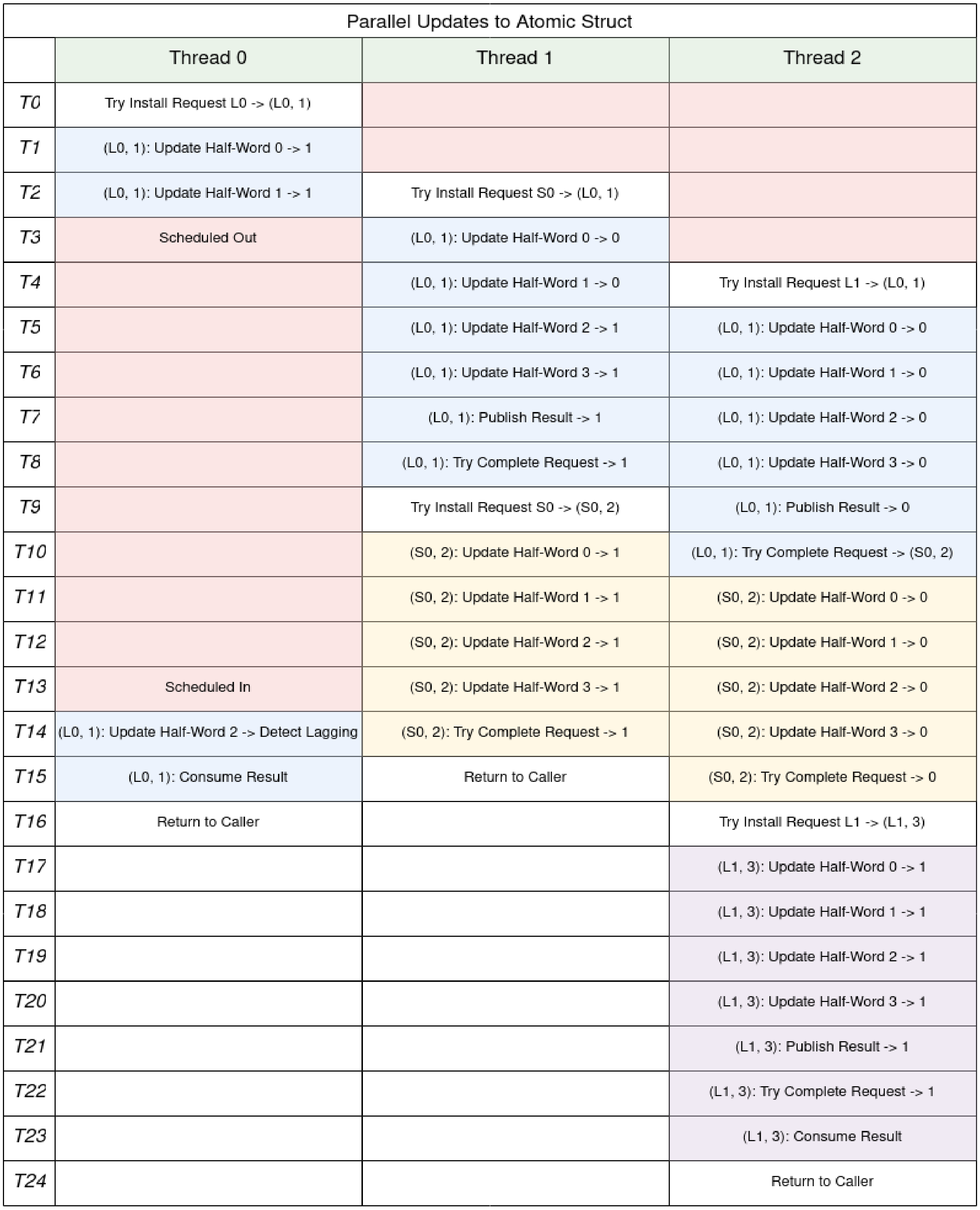
Load 1 Request	
out : atomic shared pointer	nullptr
type : enumeration	eLoad
func : Restartable Function	process

Store 0 Request	
contents : MyStruct	{...}
type : enumeration	eStore
func : Restartable Function	process

In the aforementioned scenario, each of the threads will allocate and fill out a request structure and then serialize its' invocation using an instance of *Atomic Serial Work*. The initial contents of relevant memory structures can be seen in *Figure 4*. Note that in the presence of dynamic memory allocation, it is necessary to pin the memory for each request with a hazard pointer, such that it is not reclaimed by the memory allocator until there are no more threads accessing it.

Same as with regular atomic load and store instructions, there are multiple ways that the proposed scenario can play out, depending on which thread wins the race. Irregardless of the outcome, it is guaranteed that all operations will be seen as atomic, meaning there cannot be any torn reads or writes. In order to simplify reasoning about the execution of the algorithm, it is acceptable to think of time as being quantized, where a single CPU instruction can complete in a single time quanta. To aid in further analysis, one possible outcome of the three competing requests is presented in *Figure 5*.

Figure 5: One possible flow of execution for three competing requests



Parallel Updates to Atomic Struct			
	Thread 0	Thread 1	Thread 2
T0	Try Install Request L0 -> (L0, 1)		
T1	(L0, 1): Update Half-Word 0 -> 1		
T2	(L0, 1): Update Half-Word 1 -> 1	Try Install Request S0 -> (L0, 1)	
T3	Scheduled Out	(L0, 1): Update Half-Word 0 -> 0	
T4		(L0, 1): Update Half-Word 1 -> 0	Try Install Request L1 -> (L0, 1)
T5		(L0, 1): Update Half-Word 2 -> 1	(L0, 1): Update Half-Word 0 -> 0
T6		(L0, 1): Update Half-Word 3 -> 1	(L0, 1): Update Half-Word 1 -> 0
T7		(L0, 1): Publish Result -> 1	(L0, 1): Update Half-Word 2 -> 0
T8		(L0, 1): Try Complete Request -> 1	(L0, 1): Update Half-Word 3 -> 0
T9		Try Install Request S0 -> (S0, 2)	(L0, 1): Publish Result -> 0
T10		(S0, 2): Update Half-Word 0 -> 1	(L0, 1): Try Complete Request -> (S0, 2)
T11		(S0, 2): Update Half-Word 1 -> 1	(S0, 2): Update Half-Word 0 -> 0
T12		(S0, 2): Update Half-Word 2 -> 1	(S0, 2): Update Half-Word 1 -> 0
T13	Scheduled In	(S0, 2): Update Half-Word 3 -> 1	(S0, 2): Update Half-Word 2 -> 0
T14	(L0, 1): Update Half-Word 2 -> Detect Lagging	(S0, 2): Try Complete Request -> 1	(S0, 2): Update Half-Word 3 -> 0
T15	(L0, 1): Consume Result	Return to Caller	(S0, 2): Try Complete Request -> 0
T16	Return to Caller		Try Install Request L1 -> (L1, 3)
T17			(L1, 3): Update Half-Word 0 -> 1
T18			(L1, 3): Update Half-Word 1 -> 1
T19			(L1, 3): Update Half-Word 2 -> 1
T20			(L1, 3): Update Half-Word 3 -> 1
T21			(L1, 3): Publish Result -> 1
T22			(L1, 3): Try Complete Request -> 1
T23			(L1, 3): Consume Result
T24			Return to Caller

From the execution trace in *Figure 5*, it can be seen that ultimately the observable serialization order for the three requests was: *L0* → *S0* → *L1*. It can be observed that although *Thread 0* was scheduled out between *T3 and T13*, this did not block the progress of any of the other threads, as they were able to *help* in completing an already started request. Furthermore, concurrent and lagging request invocations running on different threads did not produce any excess side-effects, thereby giving the appearance that each request is invoked exactly once.

Fundamentally, *Atomic Struct* is an implementation of *Software Transactional Memory*, and has a broad range of applications. For example, it can be used to implement software emulation of the mythical DCAS instruction (double compare-and-swap operating on two disjoint, not necessarily contiguous, memory words), which is the basis of a number of classical lock-free algorithms.

4.0 The Two Queues Problem

The *Two Queues Problem* arises when two parallel threads attempt to conditionally enqueue a value to each other's queues, such that only one of the enqueues can be successful. For example, say there are two tasks running in parallel and one task wants to *wait* on an event and the other task wants to asynchronously *broadcast* the event. The awaiting task should first try to dequeue the event from the *event queue* and resume immediately if there is an existing enqueued event. If the event queue is empty, the task should block by inserting itself after the tail of the *awaiters queue*. Conversely, the notifying task should first try to process the event awaiters queue, unblocking any awaiters immediately, and, if necessary, enqueue the event to any remaining tasks' event queues.

Unfortunately, there is no way to realize the aforementioned scenario in a lock-free manner using only classical algorithms and data structures. Fundamentally, the Two Queues Problem can be distilled down to a necessity for a means to update an additional bit of state atomically together with an enqueue operation.

Figure 6: The Two Queues Problem

Awaiter Thread	Notifier Thread
`if(!notified) {` `blocked = true;` `awaiters_queue.Enqueue(this);` `}`	`if(!blocked) {` `notified = true;` `event_queue.Enqueue(notification);` `}`

Unless the entire sequence of the awaiter and notifier from *Figure 6* can be made atomic and lock-free, it is necessary to resort to locking in order to solve the Two Queues Problem. Otherwise, if the awaiter thread is halted between setting *blocked* and pushing to the awaiters queue, the notifier thread can miss unblocking the awaiter. Conversely, if the notifier thread is halted between setting *notified* and enqueuing to the message queue, the awaiter thread can get deadlocked polling for a message that never arrives.

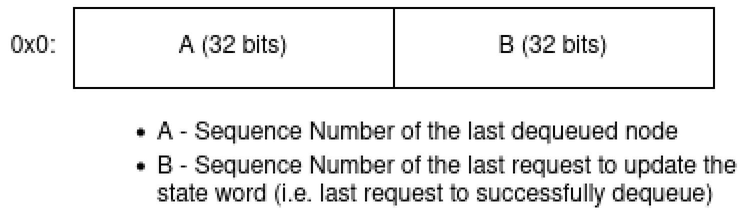
5.0 Lock-Free Sequenced Queue

Lock-Free Sequenced Queue is a lock-free data structure with a queue-like API and supporting additional methods like *ConditionallyEnqueue* and *ConditionallyDequeue*, which allow atomic and lock-free modification of the queue based on some external state.

All operations on the queue are serialized with an instance of *Atomic Serial Work*. From this, it follows that all modifications to the internal queue state must be done from a restartable function and thus must take care not to cause undesired side-effects if restarted multiple times. In order to support restartable enqueuing, the nodes of the queue are internally stored in a lock-free set, hashed by the sequence number of the enqueue request. This way, multiple insertions from different threads will simply atomically overwrite the value for the same key with the same value, ultimately appearing as if only a single insertion has ever taken place. However, given that a thread executing a restartable request can be halted for an arbitrary amount of time, it is possible that a *ghost insertion* might take place at any later time, inserting a node into the internal set that has already been dequeued by some later request. Such *ghost* nodes can be safely identified and discarded by the corresponding dequeuing logic.

In order to support a restartable dequeuing operation that also correctly discards ghost nodes, it is necessary to require some additional guarantees from the internal lock-free set structure. First of all, it is necessary that the set maintains the nodes in sorted order, with the lowest sequence number being first. Secondly, the set must provide an API such as *PeekHead*, to atomically poll the key and value of the head of the set, in essence the oldest successfully enqueued node. Fortunately, such a set can be created with only a very minor modification to Harris' classical lock-free linked list. The Harris list is already, in principle, a set as it does not allow duplicate values. Furthermore, the nodes in Harris' list are stored in sorted order. A slight modification is made to the API in order to allow inserting key-value pairs instead of values directly. Additionally, the *PeekHead* method is implemented in order to atomically read the head of the list without deleting it. While the head node can be deleted or shifted in the list immediately after polling, this does not affect the correctness of the dequeuing algorithm. Lastly, it is necessary to add an additional word of state, access to which is also serialized through the same *Atomic Serial Work* instance as the set. The *dequeue state* word encodes in it two sequence numbers – the sequence number of the last dequeued node and the sequence number of the last request to perform the said dequeue.

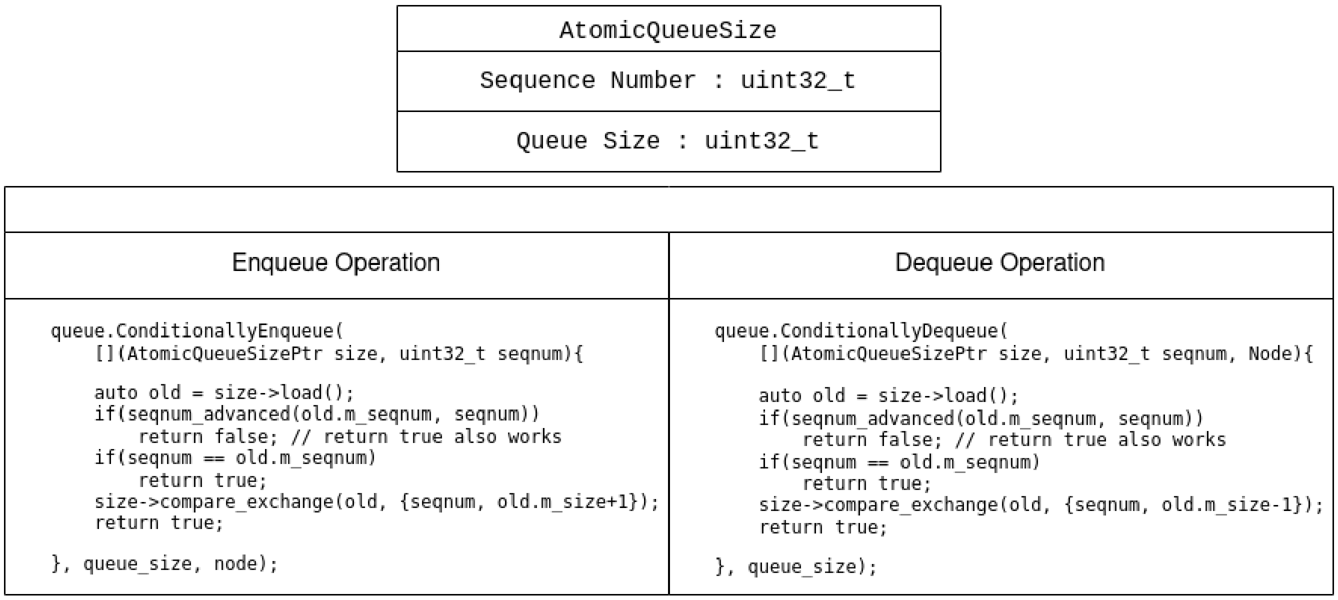
Figure 7: Additional internal dequeue state of the lock-free sequenced queue


0x0:
A (32 bits)
B (32 bits)
• A - Sequence Number of the last dequeued node
• B - Sequence Number of the last request to update the state word (i.e. last request to successfully dequeue)

The dequeue operation makes use of the *Atomic Work Pipeline* primitive in order to perform the steps of the restartable request in two stages with the assurance that all side-effects from the first stage are guaranteed to have completed at the time that the second stage is invoked. Fundamentally, this acts like a nested *Atomic Serial Work* primitive which re-uses the sequence number from the outer *Atomic Serial Work* instance. This allows splitting the dequeue restartable request into two stages that both utilize the same sequence number. The first stage selects the victim node to be dequeued and the second stage commits the dequeuing of the victim node. Since the output of the first stage, which is the victim node, is published for consumption by the second stage through an atomic shared pointer, it is guaranteed that the same victim node will be seen by all threads executing the second stage of the request. Such a caching of the victim node is necessary as the state of the internal set can change between different invocations of the dequeue request. It would be unacceptable to select different victims on different invocations and ultimately delete two nodes in a single request. The second stage commits the dequeuing of the victim node by attempting to update the dequeue state word with a CAS loop. If it is detected that a later request has already overwritten the dequeue state word with a higher sequence number, then it is deduced that the current request has already completed and can be aborted. Otherwise, the sequence number of the deleted victim node is set and is guaranteed to be seen by all future dequeue requests, which can use it to discard ghost nodes. The value of the dequeued node is forwarded to the request instigator through an atomic shared pointer. In subsequent dequeue requests, when the victim node is selected, the head of the internal set is polled. Since the sequence numbers of enqueued nodes are strictly advancing, all nodes having a sequence number less than or equal to the maximum dequeued sequence number can be deleted from the internal set. Note that the state of the internal set is only weakly consistent with the observable state of the queue, since it can contain ghost nodes as well as already deleted nodes. However, this inconsistency is invisible to clients of the API.

Having restartable *Enqueue* and *Dequeue* operations, it is only a short step to realize additional *ConditionallyEnqueue* and *ConditionallyDequeue* methods. Specifically, an additional restartable function predicate must be passed in as part of the request. In the body of the request function, it can be invoked and the operation can be aborted if the predicate evaluates as false. For additional flexibility, an extra context argument can be passed to the restartable function predicate via a shared pointer.

Figure 8: Lock-free queue with exposed fully consistent size



Enqueue Operation

```
queue.ConditionallyEnqueue(
    [](AtomicQueueSizePtr size, uint32_t seqnum){

    auto old = size->load();
    if(seqnum_advanced(old.m_seqnum, seqnum))
        return false; // return true also works
    if(seqnum == old.m_seqnum)
        return true;
    size->compare_exchange(old, {seqnum, old.m_size+1});
    return true;

}, queue_size, node);
```

Dequeue Operation

```
queue.ConditionallyDequeue(
    [](AtomicQueueSizePtr size, uint32_t seqnum, Node){

    auto old = size->load();
    if(seqnum_advanced(old.m_seqnum, seqnum))
        return false; // return true also works
    if(seqnum == old.m_seqnum)
        return true;
    size->compare_exchange(old, {seqnum, old.m_size-1});
    return true;

}, queue_size);
```

Figure 8 shows how the lock-free sequenced queue can be used to implement a lock-free queue that exposes a size attribute that is fully consistent with the observable queue state. Note, however, that in the general case, it is not safe to access the size field in a manner not serialized on the queue's internal *Atomic Serial Work* instance. While the presented facility is only marginally useful, it does demonstrate that the lock-free sequenced queue data structure can be used to solve the aforementioned Two Queues Problem.

6.0 Lock-Free Scheduler Design

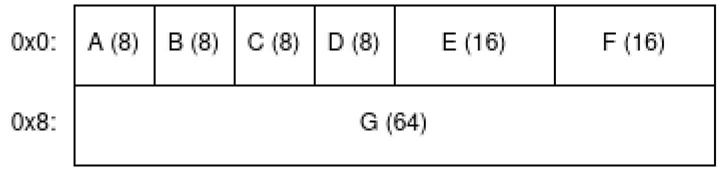
Peredvizhnikov Engine implements the *work-stealing* scheduling strategy. All of the tasks in the engine are scheduled on a set of worker threads, which hold a lock-free deque of tasks for each of the predefined priority levels. Newly spawned tasks will get pushed onto one of the creating thread's deques, though it is possible for an idle worker to *steal* work from another thread when necessary. This schema allows for a soft affinity between tasks and threads as well as an increased likelihood for parent and child tasks to execute on the same thread, while still tolerating migration to ensure optimal load-balancing. Higher priority tasks will always be executed before lower priority tasks in order to allow for a predictable latency when reacting to critical events.

In order to not block any of the worker threads, an *IO Pool* is implemented to allow offloading any blocking calls. The IO Pool contains a set of threads that are meant to be primarily scheduled out. They make use of futexes to wait until some request is enqueued and proceed to perform it on behalf of the instigating task. Upon completion of a blocking call, the IO thread will, in turn, unblock the task which instigated the blocking request. This allows all the worker threads to remain active performing CPU-intensive workloads.

There are two primary means of inter-task communication in Peredvizhnikov Engine, namely *messages* and *events*. A task's *Send*, *Receive*, *Reply*, and *PollMessage* methods deal with messages while the *Subscribe*, *Unsubscribe*, *Broadcast*, *Event*, and *PollEvent* methods deal with events. While the mechanism of their delivery is asynchronous, messages can be thought of as being synchronous because the sending task gets blocked until the receiver unblocks it with a *Reply* call. They are, in essence, akin to a method call on a different task. Events, on the other hand, can be thought of as being asynchronous because they don't require a response and don't block the broadcasting task. Events are also multicast, as they will be delivered to all subscribers of that particular event. In the presence of multiple event produces and multiple event subscribers, it is guaranteed that all subscribers will receive all the events in exactly the same order.

The scheduler maintains a lock-free iterable list of subscribers for each kind of event. The iterable list allows taking concurrent linearizable snapshots and thus enables querying all current subscribers of an event in a lock-free manner. The scheduler also holds a set of lock-free sequenced queues, the *awaiter queues*, to hold descriptors for all tasks blocked on every kind of event. Each task has its' own set of lock-free sequenced queues, the *event queues*, corresponding to each type of event. Publishing and consuming events is serialized through the scheduler's internal *Atomic Serial Work* instance. In addition, every task contains a lock-free sequenced queue of messages, the *message queue*, as well as a 16-byte atomic control block describing the task's current state.

Figure 9: Atomic Task Control Block



- A - Current Task State
- B - Message Sequence Number
- C - Notify Unblock Counter
- D - Notify Sequence Number
- E - Event Sequence Numbers (1 bit per event for 16 different events)
- F - Event Blocked Mask (1 bit per event for 16 different events)
- G - Pointer to Task's Awaiter

Figure 9 shows the structure of the task's atomic control block. There is room to re-allocate some of the bits as the scheduler's design is evolved and perfected. Field *B* is used for synchronization of message delivery and access to it is serialized through the *Atomic Serial Work* instance of the task's message queue. Fields *C*, *D*, *E*, and *F* are used for synchronization of notifications and their access is serialized through the scheduler's internal *Atomic Serial Work* instance. Another detail which must be addressed is the case of sequence number overflow, which is especially relevant with a small sequence number space such as 8 bits.

Figure 10: Algorithm to detect sequence number advancement with overflow

```
bool seqnum_passed(uint8_t a, uint8_b b)
{
  return (static_cast<int8_t>(b - a) > 0);
}
```

Figure 10 presents a bitwise trick that can be used to detect overflow of sequence numbers. For 8-bit numbers, it works so long as the the difference between the compared sequence numbers is less than or equal to 2^7 (128). Strictly speaking, this means that it is not possible to correctly handle lagging requests that are more than 128 invocations behind the latest sequence number. Practically speaking, this is tolerable as such a case is *de facto* impossible when running the engine in the context of modern operating systems with a moderate number of processor cores.

A task's *Receive* method will poll the task's message queue and resume immediately if a message is found. Otherwise, it will put the current task into a *SendBlocked* state. To allow this sequence to happen atomically, the logic for updating the task's control block is put into a restartable function that is invoked in the body of the message queue's *ConditionallyDequeue* request. Each time the message queue is dequeued from, the *Message Sequence Number* in the task's control block is atomically set to the lowest 8 bits of the sequence number of the dequeue request. In the case that there is no message in the queue, the state field of the task's control block is atomically updated to the *SendBlocked* state and the task is descheduled. The *Send* method will deliver the message directly and unblock the receiver task if it is found to be in the *SendBlocked* state. Otherwise, it will enqueue the message in the receiver's message queue. The method will also unconditionally block the sender task. Similarly, updates and queries to the receiver task's control block are placed into a restartable function which is invoked as part of the *ConditionallyEnqueue* request that is serialized on the receiver's message queue. The *Message Sequence Number* set by the receiver is used by the sender to detect a lagging request. That is, if the read sequence number is greater than that of the current enqueue request, then it is certain that the receiver has already dequeued the current message and its' state does not need to be updated. If the receiver task is found to be in the *SendBlocked* state, a CAS loop is used to advance it to the *Running* state and schedule it for execution. The receiver may then unconditionally unblock the sender with a non-blocking *Reply* call.

Due to the inherent complexity of supporting multicast scenarios while retaining strong ordering guarantees, the *Broadcast* and *Event* primitives are naturally the most sophisticated. The *Event* method will first poll the corresponding event queue of the task through a request serialized on the scheduler's internal *Atomic Serial Work* instance. If an event is found in the queue, the task is resumed immediately, otherwise it is placed in the *EventBlocked* state and enqueued to the appropriate awaiters queue, also through a request serialized on the scheduler's *Atomic Serial Work* instance. When the task is blocked, the bit corresponding to the event is set in the control block's *Event Blocked Mask* field. This is to allow determining exactly which kind of event this task is waiting for.

The *Broadcast* method must, in turn, either unblock each event subscriber or, alternatively, enqueue the event to the subscriber's appropriate event queue. It is important that the entire procedure of notifying all the subscribers collectively appears as one atomic operation. Otherwise multiple *Broadcast* requests could be interleaved and it would be possible for tasks to receive events out-of-order, greatly complicating the engine's consistency model. It follows that the entire sequence of notifying each subscriber must be made wholly restartable and serialized on the scheduler's *Atomic Serial Work* instance.

At first, a snapshot is taken of all the current event subscribers and it is copied to the notification request. The notification request makes use of a three-stage *Atomic Work Pipeline* primitive, effectively inserting two *sequence points* in the body of the request. The pipeline is as wide as the subscribers snapshot, enabling for the processing of each individual subscriber to be done by a different thread. The first stage of the pipeline will poll and cache the appropriate *Event Sequence Number*, *Notify Sequence Number*, and *Notify Unblock Counter* for each subscriber. The *Notify Sequence Number* is a counter for the number of times a task has been notified in general, while the *Event Sequence Number* is a single bit corresponding to a specific event type that is flipped each time a task is notified for that particular type of event. Ideally, each task would have a separate sequence number for every kind of event, however there is not sufficient bit space in the control block to accommodate this, so a single-bit per event type is used in conjunction with the global notification counter. It is necessary to cache these values once in the first stage as they are subject to change as soon as the second stage begins execution.

In the second stage, all subscribers that are not currently event-blocked on the specified event will be notified asynchronously. Those subscribers that are detected to be *EventBlocked* are forwarded to the third stage to be unblocked. A lock-free set that is shared between all invocations of the request is used as a cache to keep track of descriptors for tasks that have been identified as *EventBlocked*. At this point, the awaiters queue is cooperatively processed by the request executors to detect all blocked tasks. By inserting a copy of the node into a shared lock-free set in the body of a *ConditionallyDequeue* request, it can be guaranteed that no dequeued node is ever private to any one thread and thus cannot be lost due to thread termination. Care must be taken as a subscriber that has already been notified by the current request may reappear back in the shared awaiters queue. Such tasks can be identified by comparing the current *Notify Sequence Number* of the task with the value cached in the request. To allow skipping such tasks when processing the queue, an additional *ProcessHead* method is added to the lock-free sequenced queue that allows the head of the queue to be atomically accessed and conditionally cycled back to the end of the queue. This is acceptable since the order of the awaiters in the queue does not actually matter. If a subscriber is not detected as being blocked, an attempt is made to enqueue the event to its' event queue. This is done with a *ConditionallyEnqueue* request that attempts to advance the global *Notify Sequence Number* and flip the type-specific *Event Sequence Number* bit. These two values are used conjointly to detect a lagging request and abort it if necessary. If it is detected that a task has already become *EventBlocked*, the enqueuing request is aborted and the subscriber is forwarded to the third stage of the pipeline. In summary, the second stage is responsible for enqueuing the event argument to any tasks that are not blocked, draining the awaiters queue, and recording and caching the set of blocked tasks in the request. Lastly, the third and final stage is responsible for re-scheduling all the *EventBlocked* tasks. The *Notify Unblock Counter* is atomically incremented and is used to prevent duplicate unblocking by lagging requests.

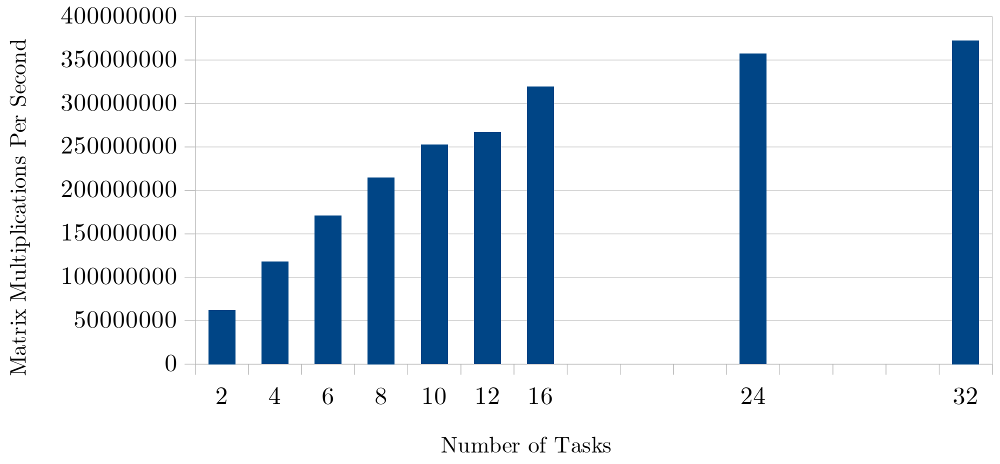
Although there is some necessary complexity in making the inter-task communication routines restartable, and consequently lock-free, the benefits of lock-freedom and a strong consistency model cannot be understated. The resulting set of task primitives is well-suited for creating performant and highly-parallel applications within the engine's framework.

7.0 Conclusion

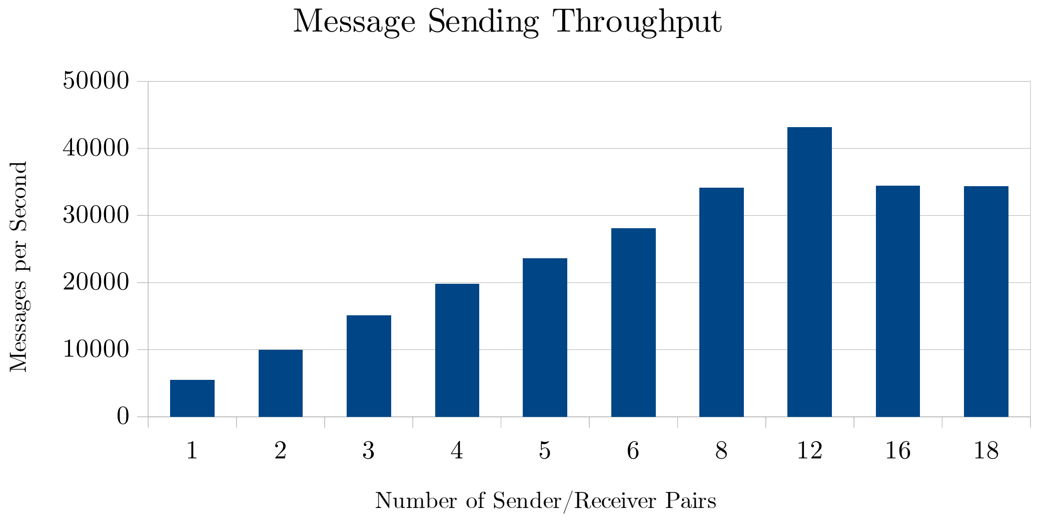
To verify the operation of the scheduler and gauge different performance metrics, a number of tests were conducted and several benchmarks were collected. All tests were run on a Linux machine with 32 GB DDR4 3200MHz RAM and an Intel Core i7-11800H Processor with 8 cores, 16 threads, and 24 MB Cache running at 45W TDP. During the execution of the experiments, the engine was configured with 16 worker threads, one per CPU thread.

In the first experiment, a simple CPU worker task that would multiply matrices in a loop was created. The test spawned different numbers of worker tasks and measured the total number of multiplications in a fixed time interval for varying worker counts. The results showed linear scaling of the number of matrix multiplications with the worker task counts, up to the number of CPU threads, after which the number of multiplications stayed nearly constant. This is a favorable result, as it demonstrates that the scheduler is able to fully saturate the system CPUs and transparently provide a high degree of parallelism.

Scaling of CPU-Intensive Workloads

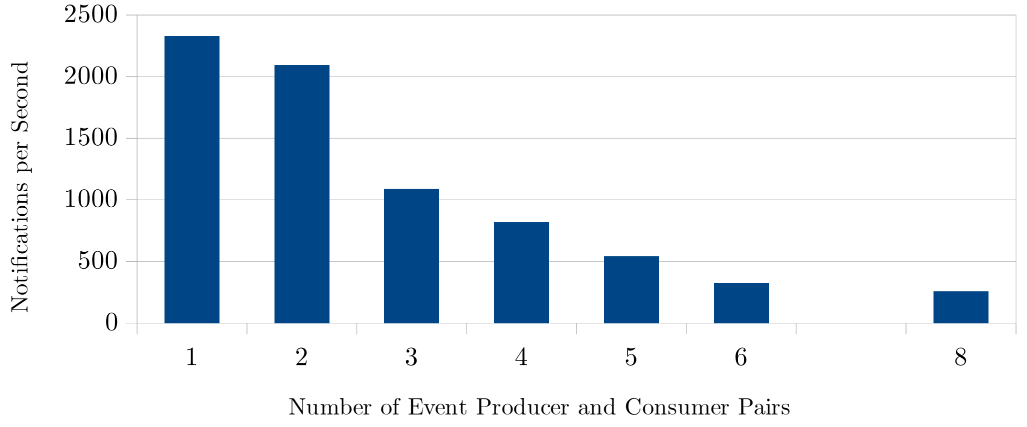


The second experiment measured the throughput of messages in the system. Varying numbers of pairs of sender and receiver tasks were spawned. The sender task would send messages to the receiver in a busy-loop for some fixed time interval and the total number of sent messages was recorded. Again, the results showed good scaling with the number of tasks up to the point where the total number of tasks approached the number of CPU threads. The peak message throughput on the test system was recorded as approximately 43500 messages per second.



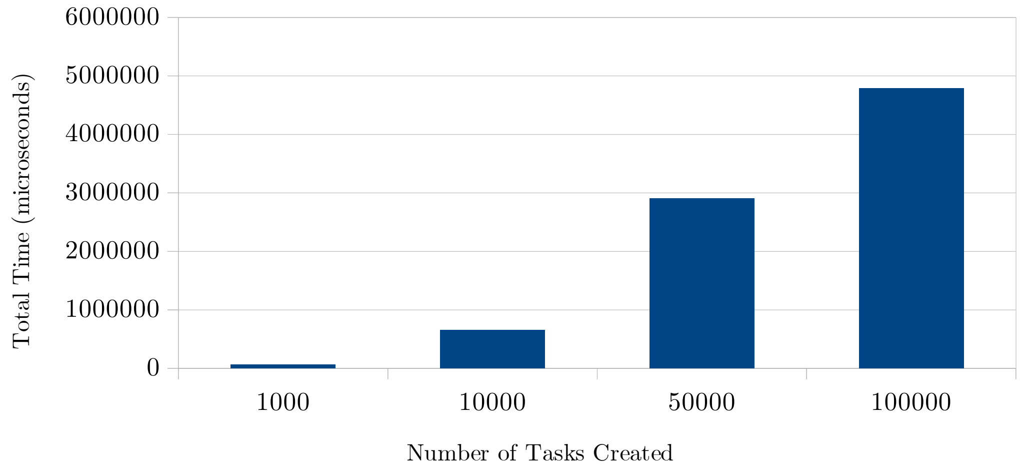
The third experiment measured the throughput of notifications in the system. Varying numbers of event producer and consumer tasks were spawned. The producer tasks would broadcast notifications in a busy-loop to all the consumer tasks for some fixed time interval and the total number of notifications was recorded. The results showed decreased throughput as the number of producers and consumers was increased. This can be explained, first of all, by the fact that all notifications are globally serialized, so a higher number of notifications would increase contention. Secondly, as the number of subscribers for an event increases, the amount of work to be done for each notification request also increases. Due to the complex implementation, the peak notification throughput was recorded as only approximately 2300 notification per second. The takeaway here is to use events sparingly, and not to use them in favor of messages unless there is good reason to do so.

Notification Throughput



Next, the throughput of task creation was measured. Tasks that would immediately exit were spawned in a loop and the time to spawn the tasks was recorded. The results showed that, on the test system, tasks could consistently be created at a rate of 15,000-20,000 tasks per second, a rate greater than that of creating OS threads. Furthermore, it was shown that given sufficient RAM resources, the engine could handle as many as 100,000 dormant tasks.

Task Creation Throughput



The final experiment tested the fault tolerance of the system. A number of idle tasks as well as a single monitor task, which was pinned to the main thread, were created. The idle tasks would consistently send "keep-alive" updates to the monitor task, either by messages or by events. The monitor would periodically report to the terminal about the idle tasks that it has received updates from in the last time interval. A "killer" thread was created which would periodically send an asynchronous *SIGSEGV* signal to one of the remaining worker threads via a *tgkill* system call. The worker thread would trap into its' installed signal handler and simply exit. The experiment showed that, consistently, the engine would keep running even when worker threads were arbitrarily killed. This result really brings to light one of the main advantages of using a wholly lock-free system, namely that it is possible to create extremely robust and resilient systems. It is conceivable that in a fully lock-free engine, it is possible to isolate low-trust user logic from core system functionality, in a similar vein that user processes are isolated from core OS kernel functionality.

The primitives devised for the creation of the lock-free scheduler are powerful and applicable to a multitude of problem domains. *Atomic Work* is a universal lock-free serialization primitive that can be used as the foundation for original lock-free data structures and algorithms. *Atomic Struct* is a novel, lock-free implementation of Software Transactional Memory. Besides demonstrating the usefulness of *Atomic Work*, it is a broadly-applicable building-block of lock-free programs in its' own right. *Lock-Free Sequenced Queue*, which was designed to solve the Two Queues Problem, has shown itself to be an indispensable component in building lock-free messaging systems.

The prospects of highly parallel computing machines consisting of dozens, hundreds, or even thousands of independent microprocessors which motivated the conception of the actor model in the 1970s are rapidly becoming a present-day reality, so it is appropriate to revisit this classical idea under a new light. Peredvizhnikov Engine exhibits that using the actor model abstraction, it is possible to develop complex, parallel logic while being wholly isolated from the details of inter-thread synchronization. Furthermore, it demonstrates the feasibility of a completely lock-free implementation of the model, reaping the aforementioned benefits of guaranteed progress, deadlock-freedom, fault-tolerance, and predictable latency. The lock-free actor model architecture put forward by Peredvizhnikov Engine should be justly considered as a viable approach during the next stage in the evolution of game engine design.