SAND2015-6136C

Tervel:
A Unification of Descriptor-based Techniques for Non-blocking Programming

Steven Feldman[1], Pierre LaBorde[1], Damian Dechev[1,2]
1. Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
2. Sandia National Laboratories, Livermore, CA 94551
feldman@knights.ucf.edu, pierrelaborde@knights.ucf.edu, dechev@eecs.ucf.edu

***Abstract*—The development of non-blocking code is difficult; developers must ensure the progress of an operation on shared memory despite conflicting operations. Managing this shared memory in a non-blocking fashion is even more problematic. The non-blocking property guarantees that progress is made toward the desired operation in a finite amount of time. We present a framework that implements memory reclamation and progress assurance for code that follows the semantics of our framework. This reduces the effort required to implement non-blocking, and more specifically wait-free, algorithms. We also present a library that demonstrates the ease with which wait-free algorithms can be implemented using our framework.**

1. Introduction

Non-blocking synchronization is a problem that programmers of real-time systems have been dealing with for years. Now, this problem is becoming mainstream as companies are being funded based on these technologies [15]. The usual method of synchronization is to use a lock-based approach that might include constructs such as a mutex. We can allow for more concurrency by choosing to use a non-blocking approach which does not employ any software locks.

There are two kinds of non-blocking synchronization that we will discuss — lock-freedom and wait-freedom. *Lock-free* algorithms require that at least one thread makes progress in a finite amount of time. In contrast to this system-level requirement, is *wait-freedom* which requires that all threads make progress in a finite amount of time. Achieving any level of synchronization without software locks is a difficult problem.

We present Tervel, which is a collection of descriptor-based techniques for non-blocking synchronization. Tervel contains an approach to the problem of implementing non-blocking synchronization that is based on a progress assurance algorithm [8], and combined with hazard pointers [17] and reference counting [5]. A progress assurance method is a way of ensuring that threads, that have conflicting writes, cooperate such that forward progress is made. This cooperation is facilitated by a descriptor object which stores algorithm-specific information about a thread's operation. Hazard pointers are lists of memory blocks that cannot be freed, and an associated algorithm that allows us to know which blocks belong in the list — reference-counting serves a similar purpose in this paper. Knowing which blocks cannot be freed is the biggest problem for any memory reclamation algorithm.

Tervel provides a progress assurance scheme constructed from an announcement table [11], [14], descriptor object [3], and association model [8]. Progress assurance allows the construction of wait-free algorithms by preventing scenarios of livelock in the event a thread is continually preempted by other threads. It has been shown in [14] that these cases of livelock are exceedingly rare. And by extension the times at which a thread will use the progress assurance is also rare.

To handle memory reclamation, Tervel uses thread-local and global memory pools. It uses implementations of hazard pointers [17] and reference counting [5] to ensure objects are not reused or freed while a thread is operating on them. The API for these implementations has been expanded to allow for their use with objects that have complex dependencies.

The aforementioned techniques are packaged as a framework that allows users easy access to these algorithms. To showcase the ease by which algorithms can be implemented in this framework, we describe the implementations of a wait-free multi-word compare-and-swap algorithm and a hash map data structure. In addition to these two algorithms, our initial release includes implementations of a wait-free ring buffer and wait-free vector.

Using data structures or algorithms implemented in Tervel in an application is a straight forward procedure. It requires a main thread calling an initialization function for the library and each thread that uses Tervel algorithms to call an attachment function. Freeing of Tervel resources is accomplished calling the destructor of the object returned from the initialization or attachment function. The only task that a user must complete in order to add their own algorithm to the Tervel framework, is to implement an abstract class for a descriptor object.

Potential applications for wait-free algorithms include real-time operating systems, and other pieces of software

. This research is funded by the National Science Foundation (NSF) under Grant Numbers ACI-1440530 and CCF-1218100.

that are used for mission-critical systems that require hard real-time limits. Non-blocking algorithms can have a positive impact on performance while also providing progress guarantees to general-use applications that may not need them. An example of this has been described in [6].

Our contributions are:

- We present a framework that allows the user to write non-blocking code that is reclaimed in a wait-free manner. This includes a methodology for allowing users to add their own descriptor-based algorithms. One of the key properties of our framework that allows this to be possible is the fact that all library structures are unified and composable.
- We extend all of the existing techniques that we use in order to provide additional features that are necessary for compatibility with our library; such as, stronger progress guarantees, and composability. Examples of this include the association model that we incorporated into the announcements which are placed in the announcement table, allowing for the design of more complex announcements that remove the risk of individual operations being executed more than once. We also add additional API to our implementation of hazard pointers and reference counting to allow for the protection of objects with complex dependencies. The issues of recursion in non-blocking algorithms that are caused by the inclusion of helping routines, is resolved by two detection methods that Tervel provides as part of its framework.
- We provide an open-source library that includes implementations of known wait-free algorithms; these include a wait-free multi-word compare-and-swap(MCAS), a wait-free hash map, a wait-free ring buffer, and a wait-free vector.

2. Background

This section provides an overview of techniques that have been used to implement non-blocking algorithms and data structures.

2.1. Inter-Thread Helping Techniques

The most ubiquitous inter-thread helping technique used in non-blocking algorithms is the descriptor object. A descriptor object is used to describe a pending operation [3]. A thread places a reference to a descriptor object into shared memory. Operations that read a reference to a descriptor object will often perform a helping routine. For a descriptor-based operation to be lock-free, the descriptor object must contain enough information such that an arbitrary thread can complete the operation described within. Descriptor objects are often incorporated into the design of algorithms that modify the value of an address based on the value of one or more other addresses.

Lock-free algorithm designs that use descriptor objects often show correctness by stating that if a thread placed a descriptor, then it has made progress in completing its own operation. Any thread that sees this descriptor will perform a helping routine before continuing with its own operation. If a descriptor is placed using a compare-and-swap(`cas`)[1] operation, it is possible for a thread to fail at placing. In this event, some other thread must have successfully placed a descriptor object. As a result, the system has made progress which is the requirement of lock-freedom.

It is difficult to use descriptor objects to construct wait-free algorithms. This is because there is a theoretical danger where one or more threads will be perpetually helping other threads, thus making no progress in their own operation. Because of this, there is no guarantee that a thread will make progress on its own operation, which is contrary to the definition of wait-free.

An announcement table scheme can be used in conjunction with descriptor objects to achieve wait-freedom. This scheme allows a delayed thread to announce when it is unable to make progress [12]. Other threads will observe this announcement and help the delayed thread. An announcement differs from a descriptor object in that it describes an entire operation, as opposed to just a portion of an operation. Before commencing an operation, a thread is required to check this for table for announcements. If an announcement is found, a helping a routine is executed based on the contents of the announcement. A thread writes an announcement to its position in the table when it has failed to make progress a predefined number of times.

Herlihy's design requires each thread to check the entire table before commencing any operation. This check is very costly and makes the design impractical for systems with a large number of threads. However, the theoretical upper bound for the number of operations that can complete before an announced operation does is only $numberOfThreads$.

Kogan [14] proposes new methodology by which a thread checks for an announcement. This methodology uses a thread-local counter, $checkDelay$, to control how often a thread checks for an announcement. Another thread-local counter, $checkPosition$, is used to track the last checked position in the table. If after decrementing $checkDelay$, it is 0, the thread will read the value at $checkPosition$ in the table. If the position holds an operation, the thread will help complete that operation. Before the function returns, $checkPosition$ is incremented and $checkDelay$ is set equal to the user-defined constant $maxDelay$. This methodology reduces the number of atomic loads caused by including this check from the number of executing threads, $numberOfThreads$, to $(1/maxDelay)$. However, it raises the upper bound on the number of operations that can be completed before an announced operation is guaranteed to complete. The theoretical upper bound of this approach is $maxDelay * numberOfThreads^2$.

1. An atomic operation that writes a value to a memory location only if its current value is equal to a given expected value.

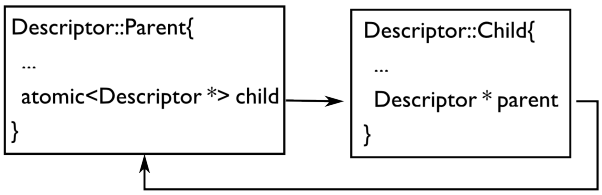


Figure 1: Associated descriptor objects.

The multi-word compare and swap algorithm of Feldman et al. [8] uses a bi-directional association model to ensure correctness when using multiple descriptor objects. Two objects have a bi-directional association when they hold references to one another. This association can be used to prevent multiple threads from successfully completing the same operation multiple times. In general, it is performed as follows: Let object A cause some thread to determine that it needs to place an object B. Before placing B, the thread will assign a reference variable in B equal to A. After placing B, the thread will use a `cas` operation to change a reference variable in A from null to B. Figure 1 provides an example of this association model.

If B references A and A reference B, the two are said to be *associated*. If however, A does not reference B, it implies B was placed after some other thread associated an object with A. This further implies, that B was placed after the operation that placed A has been completed. Therefore, B should be replaced by the value that B had initially replaced.

The following sections describe how we use and expand upon the above techniques in the implemented algorithms.

3. The Tervel Framework

Tervel is a framework for implementing non-blocking libraries. It is designed to unify, extend, and improve the usability of known techniques for developing non-blocking algorithms. Tervel provides memory reclamation, descriptors, and progress assurance constructs. These constructs provide fundamental building blocks by which developers can implement non-blocking algorithms.

3.1. Memory Reclamation

A number of papers that present concurrent data structures suggest the use of either hazard pointers or reference counting to support reusing memory, however they omit the necessary implementation details [7]–[9], [20]. Tervel provides a comprehensive interface by which developers can add either hazard pointer (HP) or reference counting (RC) protection to shared memory or objects. Throughout this paper we refer to the act of applying memory protection as *watching*, the act of removing memory protection as *unwatching*, and an object that has memory protection as *watched*. An object is watched by calling either a RC or HP specific watch function. This function returns a boolean indicating whether or not it was successful. FALSE is returned in the event the value at the address the object was read from has changed.

This describes the standard procedure by which HP or RC are typically used and it works well for objects that are only accessible through a single reference. For objects that maybe accessed through multiple references, such as those used in the association model, it does not provide adequate protection. To protect these objects, we allow the developer to define additional steps that are performed during the watching of an object. These steps are encapsulated by an `on_watch` member function, which is called by the `watch` function if the object was successfully watched. If the `on_watch` function returns FALSE, the `watch` function removes the watch on the object and also returns FALSE. In addition to the `on_watch` function, we also provide `on_unwatch` and `on_is_watched` functions. Section 5 shows how these three functions simplify our implementation of existing algorithms. A developer has access to these functions by extending the `HPElement` or `RCElement` abstract class.

In order to safely reuse objects or return memory to the system, Tervel provides both thread-local and shared memory pools. When an object is no longer needed, the *object's owner* will call a specialized free function based on how the object was allocated. An object's owner is a thread responsible for freeing that object. An object must be owned by only one thread or it may result in an object being freed by multiple threads. In general, an object's owner is determined as follows:

- An object is initially owned by the thread it was allocated to.
- A thread takes ownership of an object that it removed all references to it.
- An object's ownership transfers to a thread if it becomes associated with that thread's operation.

The last point is necessary for objects that contain references to other objects. For these objects, it is usually the case that neither can be freed while either is watched. An object is freed only if the call to `is_watched` returns FALSE. This function internally calls the `on_is_watched` member function of the passed object. This allows a developer to encode logic to prevent an object from being freed prematurely. It does require a *root* object to be identified and have that object's destructor call the appropriate free function for each object referenced by the root object.

When freeing an `HPElement` object, a thread adds the object to its thread-local `HPElement` memory pool. Then for each element in the pool, the `is_watched` function is called on it. If it returns FALSE, the object is removed from the pool and it is returned to the allocator.

An `RCElement` cannot be returned to the allocator, instead it is moved from an *unsafe* memory pool to a *safe* memory pool. This is because it is possible for the reference count member of the object to be incremented at any point, making it unsafe to return the object to the system. When allocating an `RCElement` the thread will attempt to

get an object from the following sources in order: thread-local safe pool, thread-local unsafe pool, shared safe pool, shared unsafe pool, and finally system allocator. To prevent a single thread from accumulating too many objects, we implemented a load balancing scheme. If a thread contains too many objects, it offloads the excess to the shared pool.

To simplify management of subclasses of `RCElement` that exhibit varying sizes, we force the allocated size of these objects to be a multiple of the system cache. A separate pair of unsafe and safe pools are used for each size. Our implementation improves memory utilization of an application by allowing all instances of all algorithms to share a common set of memory pools. This is in contrast to algorithms that contain their own independent reclamation scheme.

3.2. Descriptor-based Methodologies

Tervel provides an abstract descriptor class to guide the implementation of descriptor objects. Objects extending this abstract class must provide implementations of the `value` and `complete` member functions. The value function returns the logical value of the descriptor object. This is either the value that the descriptor object replaced, or a value determined by the operation that placed the descriptor. The latter is returned if the operation has been completed, but the descriptor has not yet been removed. The complete function is used to remove a descriptor object from an address.

These functions enable a developer to more readily reason about the correctness of two concurrent operations. The developer must only need to consider the case where a thread's calls the complete function of a descriptor object at an arbitrary point in time. Without such functions, the developer may have to consider more complex interactions; e.g., two or more different operations executing concurrently.

If the algorithm's operations are linearizable, then it can be shown that two concurrent descriptor based operations, operating on overlapping address spaces, are ordered by whichever placed a descriptor object at a common address first. The other will either see the descriptor and help, or the operation will be completed first. Cyclic dependencies may arise if an operation uses multiple descriptor objects. However, this can be prevented by placing descriptor objects in an ascending or descending address order.

The association model uses two or more descriptor objects, where one object is a *parent* and the rest are *children*. The parent contains atomic reference(s), which are initially NULL, and are set using a `cas` operation. The children contain a reference to the parent object. A child and a parent are said to be *associated* if the child references the parent and the parent references the child. When using this model, it is necessary to include specific logic in the `on_watch` function of a child descriptor object to ensure that it returns TRUE only if the child is associated with its parent. In general, the `on_watch` function attempts to acquire a watch on the parent object and if successful, it attempts to associate them. If this association fails, the descriptor object is replaced by its logical value, before returning FALSE Otherwise, the function returns TRUE.

The watching of the parent object is an important step. Consider the case where a thread attempts to associate a child object with a parent object. However, just before the `cas` operation is invoked, the parent object is freed and reused. When the `cas` operation executes, the application could experience undefined behavior.

By encapsulating this logic in the `on_watch` function, we reduce the number of places where an implementation error may occur. The `on_watch` function ensures that if an object is watched, it is also associated.

3.3. Recursive Helping

Recursive helping has not been discussed in the literature but its presence may lead to a scenario where a thread consistently sees new descriptor objects that it must remove before being able to finish executing its current operation. Tervel provides two mechanisms by which to detect such an event.

The first is that each thread tracks the number of operations it is currently helping. If this number exceeds the number of executing threads, the thread will return back to its own operation. For a thread to have gotten to this point, it implies that at least one of the operations the thread believes it is helping has completed. If this is the case, there no longer exists a dependency between the threads own operation, and the one it is currently helping.

The second mechanism has each thread store, in a thread-local variable, the address of a control word. When the value of this control word is no longer NULL, it implies the thread's operation is complete. This allows a thread to detect if some other thread has completed their operation, while it was performing a helping routine. For algorithms that use the association model, the control word is often the atomic reference to the child member inside the parent descriptor object.

3.4. Progress Assurance

The majority of the wait-free algorithms implemented in Tervel depend on an announcement table [10] for their progress guarantees. In Tervel we refer to an announcement as an operation record. An operation record is a type of descriptor object that contains the information necessary for an arbitrary thread to execute an entire operation. To provide $\mathcal{O}(1)$ access when checking the table, Tervel uses the methodology described in [14].

For simple operations, we follow the *fast-path-slow-path* [14] design methodology. In this methodology, a thread examines the state of an operation record and executes it if its state is not in the complete state. When designing more complex operations, this design may allow the ABA problem, or data races to occur. For example, if it is uncertain which memory words will be affected by an operation, the same operation may be successfully executed on multiple

memory locations, and values could be reused, leading to the ABA problem.

To avoid these problems, we employ the association model when we implement complex operations. In general, a thread will replace a value with references to a child descriptor object that contains a copy of the value. Then a thread will attempt to associate the child descriptor with its parent. If successful, the reference to the child descriptor is replaced by the result of the operation. Otherwise, it is replaced by the value contained within.

4. Extending Tervel with User Algorithms

This section presents a brief and general description of how a developer can use Tervel's framework to implement a non-blocking algorithm. We assume the developer already has a descriptor-based design.

The majority of non-blocking algorithms use one or more types of descriptor object. Based on how these descriptor objects are used either RC or HP memory protection may be used.

For small short-lived objects that are used repeatedly, we recommend extending the `RCElement` class. The memory pools allow threads to maintain their own allocation of objects and reduce the contention on the system allocator. Since the objects are frequently used, the fact that they cannot be returned to the system is not a significant concern.

Hazard pointer memory protection is provided by extending the `HPElement` class. Since the `HPElement` class does not contain an atomically incremented member, these objects can be returned to the system once they are no longer watched. This makes it ideal for objects that are infrequently used, very large in size, or have variable sizes.

In contrast to RC-protected objects, the maximum number of HP-protected objects watched by any given thread must be known. If recursive functions are used, it may not be possible to use HP protection, as subsequent function calls could overwrite an address. The framework includes assertions to alert the developer in the event they inadvertently reuse a position in the hazard pointer table.

Regardless of the chosen memory protection scheme, each descriptor class must have these member functions:

- value: This function returns the logical value of a descriptor object.
- complete: This function should be implemented such that upon its return the descriptor object has been removed.

The `on_watch`, `on_unwatch`, and `on_is_watched` member functions may also be implemented.

If wait-freedom is desired, an operation record must be implemented for each operation that contains *unbounded loops*. An *unbounded loop* is a loop that may execute indefinitely if certain conditions are continually met. To create an operation record, a developer will extend the `OperationRecord` class and implement a `help_complete` function. The design of the help_compete function must be such that upon its return the operation is complete. The developer can take advantage of the following statement when reasoning about the threads executing the `help_complete` function. After making an announcement, only $(checkDelay * NumberOfThreads)^2$ more operations may begin before either all threads are helping to complete the same operation or the operation has been completed. This assumes that the algorithm is livelock-free, such that two operations cannot cause each other to fail (though it is fine for one operation to cause the other to fail).

The unbounded loops can be bounded by adding a fail counter that creates an operation record and calls the announcement function, when it reaches a compile time constant. Internally, this function calls the operation record's `help_complete` function.

5. Implementation Examples

Tervel provides a number of abstract classes and structures to guide a developer who is implementing non-blocking or wait-free algorithms. The following section present excerpts from two wait-free data structures we implemented in Tervel. These excerpts were selected because they showcase the expressiveness, functionality, and conciseness of the framework. For details, see: cse.eecs.ucf.edu

5.1. Multi-Word Compare-And-Swap

Using Tervel's design patterns, we re-implemented a wait-free Multi-Word Compare-and-Swap (MCAS) [8] algorithm. Compared to the original implementation, there was less redundancy and better encapsulation of helping routines. For example, we leveraged the `on_watch` function to associate objects, removing the need to handle unassociated or incorrectly placed objects in each function. Instead the handling of these events is done purely within `on_watch`.

The MCAS design uses two types of descriptor objects, which are partially described in Figure 2 and 3. It is performed by iteratively replacing the expected value at each address with a reference to an MCasHelper. To prevent the ABA problem from occurring, this design uses the association model (Section 2). After placing an MCasHelper, the next step is to associate it with its MCasOp. We express the association model by defining an `associate` function (Figure 3 Line 38). This function uses a `cas` operation to assign a child reference to the address of the MCasHelper. It returns whether or not the child references the MCasHelper. If it references some other MCasHelper, the function removes the MCasHelper. It is important to quickly remove incorrectly placed objects to prevent other threads from accessing them. For a more thorough description of these objects and their parent classes, see our website.

Figure 3 Line 19 presents the complete function that is called by a thread to remove an MCasHelper placed by a different thread. Because a thread calls the complete function after it has acquired a watch on the object, it does not have to consider the case where a descriptor has been placed in error. For example, if an MCasHelper was placed in error, its `on_watch` function (which is called by the

```
1:  Class MCASop<T> extends OperationRecord
2:      atomic State state;
3:      const T* addresses[];
4:      const T expected_values[];
5:      const T new_values[];
6:      const atomic<MCasHelper*> helpers[];
7:
8:      Destructor MCASop
9:          for helper in helpers do
10:             RC::allocator::return(helper);
11:         end for
12:     end Destructor
13:
14:     function on_is_watched
15:         for helper in helpers do
16:             if RC::is_watched(helper) == true then
17:                 return true;
18:             end if
19:         end for
20:         return false;
21:     end function
22:
23:     function help_complete(x = 0)
24:         for x<helpers.length; x++ do
25:             if place_helper(x) == false then
26:                 return state.load() == pass;
27:             end if
28:         end for
29:         state.cas(undecided, pass);
30:         return state.load() == pass;
31:     end function
32: end Class
```

Figure 2: MCAS operation record

watch function) would have removed it when the call to associate returned false.

In order for an MCasHelper to be removed, the MCAS operation that placed it must be completed. This is accomplished by calling the MCASop's `help_complete` function. Upon its return, the state of the MCASop will have been changed from undecided to either passed or failed. Once the state has been decided, the MCasHelpers may be replaced with their logical values. The logical value of an MCasHelper is determined by calling their *value* function.

Our MCAS implementation uses Tervel's memory management features to safeguard the reclamation of descriptor objects. We use hazard pointers to protect MCASop objects and reference counting to protect MCasHelper objects. The MCASop object is responsible for freeing all MCasHelper objects referenced by it. As such, the thread that owns the MCASop also acquires ownership of those MCasHelpers. This ownership is expressed in the MCASop's `on_is_watched` and destructor functions.

5.2. Wait-Free Hash Map

The API of the wait-free hash map described in [7] mirrors that of the sequential hash map, but allows concur-

```
1:  Class MCasHelper extends Descriptor
2:      const MCASop op;
3:      const int idx;
4:
5:      function on_watch(address, cur)
6:          pos = HP::WatchPos::TempWatch
7:          if HP::watch(pos, address, cur, op) then
8:              bool res = this.associate();
9:              if res == false then
10:                 temp = op.expected_values[idx];
11:                 address.cas(cur, temp);
12:             end if
13:             HP::unwatch(pos);
14:             return res;
15:         end if
16:         return false;
17:     end function
18:
19:     function complete(address, cur)
20:         op.help_complete(idx+1);
21:         if op.state == passed then
22:             temp = op.new_values[idx];
23:         else
24:             temp = op.expected_values[idx];
25:         end if
26:         address.cas(cur,temp)
27:         return address.load();
28:     end function
29:
30:     function value()
31:         if op.state == passed then
32:             return op.new_values[idx];
33:         else
34:             return op.expected_values[idx];
35:         end if
36:     end function
37:
38:     function associate()
39:         op.helpers[idx].cas(null,this)
40:         if op.helpers[idx].load() ≠ this then
41:             temp = op.expected_values[idx];
42:             op.addresses[idx].cas(this, temp);
43:             return false;
44:         end if
45:         return true;
46:     end function
47: end Class
```

Figure 3: MCAS descriptor object

rent operations. To safeguard access to key-value pairs, we require that a thread acquire hazard pointer protection on an object before dereferencing. Unfortunately, this restriction breaks the wait-free guarantee of the algorithm. The authors describe a mechanism by which they use the `atomic or` operation to force the table to expand, in the event a memory address is experiencing heavy contention. However, we do not believe that this mechanism can be adapted to address possible livelocks introduced by applying hazard pointers.

To address this and other limitations of this concurrent design, we made the following adaptations:

- We use operation records to apply an operation in the event livelock is detected.
- We combined the `get` and `update` operation into an `access` operation.
- We changed `insert` to return true if it a key-value pair was inserted, and false if it already exists in the hash map.
- We changed `remove` to return an enum indicating one of three possible results.
  - The key-value pair was removed.
  - The key-value pair was not removed because it is currently being accessed.
  - The key-value pair is not in the hash map.

The `access` operation takes as arguments the key to find and a Tervel accessor object, and returns a boolean indicating whether or not the key exists in the hash map. The accessor object removes a lot of ambiguity that may occur in the original API where multiple updates and/or deletes may occur on the same key. A simplified example of this is presented in Figure 4. For brevity, implementation details of the `searchForKey` function has been omitted. In short it searches the hash map for the passed key and returns the following:

- found: This indicates whether or not the key was found.
- array: The array the key is or would be stored on.
- pos: The position on array the key is or would be at.
- pair: A value loaded from array[pos]. If it is a reference to an object, the object was successfully watched.

If the `access` operation returns true, the accessor is used to read and modify a key's value. Each key-value pair contains an atomic counter and internally the `access` operation performs a fetch-and-add on this counter. If the counter is now non-negative value, a reference to a pair is stored within the accessor. A negative result indicates that the object has been deleted. When the accessor is deleted, its destructor decrements the counter of the pair within.

A `remove` operation attempting to delete a key-value pair will first attempt to logically delete it (Figure 5 Line 6). Internally, the `logicalDelete` function attempts to change the atomic counter from 0 to $-1 * number\_of\_threads$. If successful, this prevents other threads from accessing the key-value pair. If it fails, the `remove` operation returns a value indicating that a thread is accessing the specified key.

```
1:  function BOOL ACCESS(Key key, Accessor *access)
2:      tervel::attemptToHelp();
3:      found, array, pos, pair = searchForKey(key);
4:      if found then
5:          res = pair→incAccess();
6:          hashmapUnwatch(pair);
7:          if res >= 0 then
8:              access→init(pair→counter, pair→value);
9:              return true;
10:         end if
11:     end if
12:     return false;
13: end function
```

Figure 4: Hash map access operation.

Figure 6 presents the `insert` operation and Figure 8 presents its operation record. For brevity, we only include the `insert` operation's operation record, the `remove` and `access` operation records are available on our website

This design places a helper object on an array, associates it with an operation record, and then replaces it with a reference to a key-value pair. The helper object allows the thread that made the operation record to determine if the insert was successful or not.

An alternative to using the helper object would be to include an additional variable in the key-value pair. However, we believe that such a design will require additional conditional statements in the key-value pair's `on_watch` function. We choose not to go with this approach because we believe that the conditions necessary for an operation record to be used are highly unlikely.

```
1:  function REMOVE(Key key)
2:      tervel::attemptToHelp();
3:      found, array, pos, pair = searchForKey(key);
4:      if found == false then
5:          return not found;
6:      else if pair→logicalDelete() then
7:          removeReference(array, pos, pair);
8:          return key removed;
9:      else
10:         return key in use;
11:     end if
12:     return false;
13: end function
```

Figure 5: Hash map remove operation.

6. Related Work

We are aware of several concurrent libraries that focus on fine-grained synchronization and progress guarantees. Below we provide a brief summary of them and key differences between them and Tervel.

```
1: function INSERT(Key key, Value value)
2:     tervel::attemptToHelp();
3:     pair = new HashMapPair(key, value);
4:     for c = 0; c < Tervel:MaxFails; c++ do
5:         found, array, pos, cur = searchForKey(key);
6:         if found == true then
7:             delete pair;
8:             return false;
9:         else if array[pos].cas(cur, pair) then
10:            return true;
11:        end if
12:    end for
13:    op = new HashMapInsertOp(this, pair);
14:    tervel::announceOp(op)
15:    res = op→res;
16:    op→safeDelete;
17:    return res
18: end function
```

Figure 6: Hash map insert operation.

```
1: Class HASHMAPHELPER EXTENDS HP::OBJECT
2:     const HashMapOp *op;
3:
4:     function ON_WATCH(address, cur)
5:         pos = HP::WatchPos::TempWatch
6:         res = HP::watch(pos, address, cur, op)
7:         if res then
8:             op→associate(address, this);
9:         end if
10:        return false;
11:    end function
12: end Class
```

Figure 7: HashMapHelper descriptor object.

The C++ Standard Template Library (STL) provides several sequential containers; none are concurrent.

Amino Concurrent Building Blocks (Amino) is an open source software project [1]. Its goal is to develop concurrent libraries or building blocks that can be used by programmers. It provides several implementations lock-free data structures, but not include any wait-free algorithms. This library was last updated on April 14th, 2010.

Boost [4] provides a lock-free queue and a lock-free stack algorithm. These algorithms are implemented based on the designs described in [13]. Like Amino, it does not provide any wait-free algorithms.

LibCDS [2] is a collection of lock-free and lock-based fine-grained algorithms of data structures like maps, queues, list etc. The library contains implementation of well-known data structures and memory reclamation schemas for modern processor architectures. While it provides more functionality and algorithms, than our initial library release, we believe that over time our library will grow to support these functionalities. In contrast to LibCDS, our goal to ensure that every component is wait-free.

STAPL (the Standard Template Adaptive Parallel Library) [19] is a framework for developing parallel programs

```
1: Class HASHMAPINSERTOP EXTENDS OPERATIONRECORD
2:     const HashMap *map;
3:     const HashMapPair *pair;
4:     atomic<HashMapHelper *> helper;
5:     function RESULT()
6:         return helper.load() != Tervel::FailPtr
7:     end function
8:
9:     function ASSOCIATE(atomic<void *> *address,
   Helper *h)
10:        helper.cas(null, h);
11:        if helper.load() == h then
12:            address.cas(h, pair);
13:        else
14:            address.cas(h, null);
15:        end if
16:    end function
17:
18:    function HELP_COMPLETE()
19:        h = new HashMapHelper(this);
20:        while helper.load() == null do
21:            found, array, pos, cur = searchForKey(key);
22:            if found == true then
23:                if cur != pair then
24:                    helper.cas(null, Tervel::FailPtr);
25:                end if
26:                delete h;
27:                return
28:            end if
29:            if array[pos].cas(cur, h) then
30:                this→associate(h);
31:                h→safeFree();
32:                return
33:            end if
34:        end while
35:    end function
36: end Class
```

Figure 8: Hash map insert operation record.

in C++. It is designed to work on both shared and distributed memory parallel computers. STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools. Its goal is to allow the user to work at a high level of abstraction and hide many details specific to parallel programming, to allow a high degree of productivity, portability, and performance.

The differences between all of these approaches are summed up in Table 1.

7. Performance

In order to demonstrate the performance of our library as compared to the others, we chose to test the one data structure that was common in most of them — the hash map. The following available (STAPL is excluded based on this requirement) libraries include a parenthesized number of hash maps: STL (1), Tervel (1), Amino (0), Boost (1), LibCDS (2). The lack of a hash map excludes Amino from

TABLE 1: Library Features by Degree (none, low, some, high)

	STL	Tervel	Amino	Boost	CDS	STAPL
Reliance on a Runtime System	none	none	low	none	low	high
Non-blocking Algorithms	none	high	some	low	some	none
Non-blocking Memory Reclamation	none	high	low	low	some	none
Source Code Availability	high	high	high	high	high	none

our tests; all libraries that make hash map implementations available are tested. LibCDS provides two lock-free hash maps [16] and [18]. Despite the fact that neither Boost nor the STL provide a concurrent hash map, we have used their sequential map implementations with a global lock; we include these approaches for completeness, and because this is a traditional approach to achieving a degree of parallelism.

The tests were conducted on a 64-core workstation running 64-bit Ubuntu Linux version 11.04, and all code was compiled with g++4.8, with level three optimizations enabled. The hash maps were all given an initial size of 1024, and were then filled halfway. Then, each algorithm would perform thirty repetitions of a five-second-long test that consisted of randomly choosing and performing an operation based on a probability distribution. The operation was chosen from one of seven probability distributions. The probability of each operation `get`, `insert`, `update`, `remove` is shown as a set of percentages here: {10, 18, 70, 2}, {10, 70, 18, 2}, {10, 88, 0, 2}, {25, 25, 25, 25}, {34, 33, 0, 33}, {88, 10, 0, 2}, {88, 8, 2, 2}. Not all of the algorithms have update functions, so these were synthesized with a combination of a `remove` operation for the old value and an `insert` operation for the new value.

Representative results of these tests are shown in Figure 9. Figure 9a shows typical hash map usage (assuming an API that does not include `update`) [18]. Figure 9b shows the opposite of this use case, Figure 9c shows an even distribution without update operations, and Figure 9d shows an even distribution of all operations.

In all of the graphs, the non-blocking approaches increase performance with the number of threads up to the hardware limit. In contrast, the blocking approaches experience non-increasing performance as the number of threads increase — usually decreasing. On average, Tervel performs better than the other non-blocking designs by a factor of 2. Tervel's performance improvement is an even larger factor of 6 compared to the blocking approaches.

8. Conclusion

Writing non-blocking code is a challenging task that can be made easier by employing our framework to solve the more challenging problems of memory reclamation, and progress assurance. We demonstrate this using our own framework to implement two non-blocking algorithms: a wait-free hash map, and a wait-free multi-word compare and swap. This framework is built using established techniques such as reference counting, hazard pointers, and the announcement scheme. We plan to extend this work in the future by adding new algorithms and data structures to the library; as well as, studying the ability of students with no experience in non-blocking programming to implement existing non-blocking approaches using our framework.

Appendix

We omit explanation and analysis of the remaining three graphs due to space concerns, but present the graphs themselves in Figure 10 for completeness.

Acknowledgements

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000.

The authors would like to thank the anonymous reviewers for their detailed and helpful suggestions. This research is funded by the National Science Foundation (NSF) under Grant Numbers ACI-1440530 and CCF-1218100.

References

[1] "Concurrent building blocks," March 2010. [Online]. Available: http://amino-cbbs.sourceforge.net/

[2] "Concurrent data structures (libcds)," September 2014. [Online]. Available: libcds.sourceforge.net

[3] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '93. New York, NY, USA: ACM, 1993, pp. 261–270.

[4] Boost C++ Libraries, http://www.boost.org/ , Retrieved 07/12/2012.

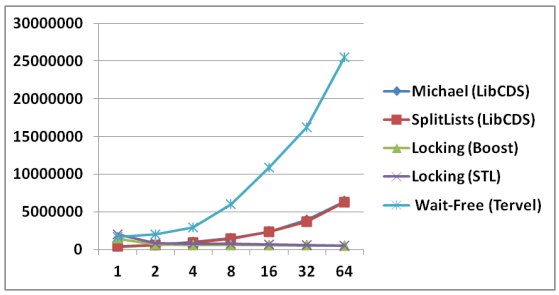
[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr., "Lock-free reference counting," in *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, ser. PODC '01, New York, NY, USA, 2001, pp. 190–199.

[6] S. Feldman, A. Bhat, P. LaBorde, Q. Yi, and D. Dechev, "Effective Use of Non-blocking Data Structures in a Deduplication Application," in *2013 ACM conference on Systems, Programming, Languages and Applications: Software for Humanity (ACM SPLASH)*, Indianapolis, IN, USA, October 2013.

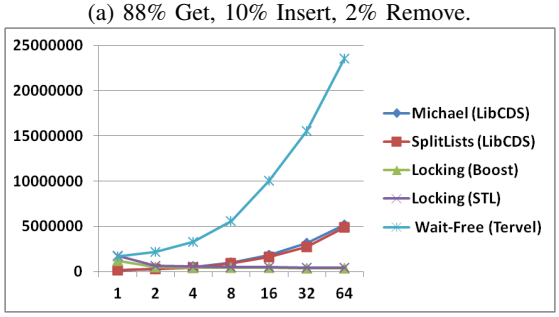
[7] S. Feldman, P. LaBorde, and D. Dechev, "Concurrent multi-level arrays: Wait-free extensible hash maps," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2013, pp. 155 – 163.

[8] ——, "A wait-free multi-word compare-and-swap operation," *International Journal of Parallel Programming*, January 2014.

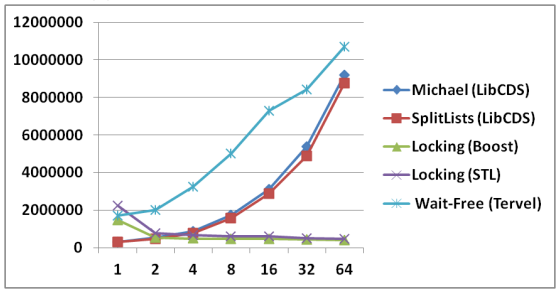
[9] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK: Springer- Verlag, 2001, pp. 300–314.



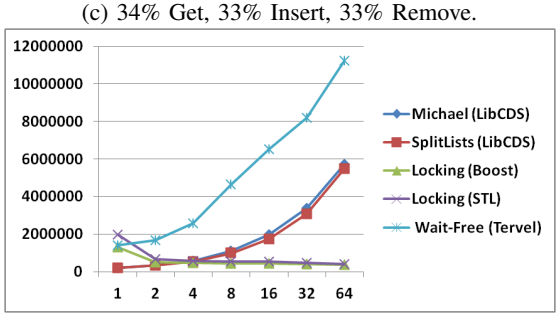
(a) 88% Get, 10% Insert, 2% Remove.



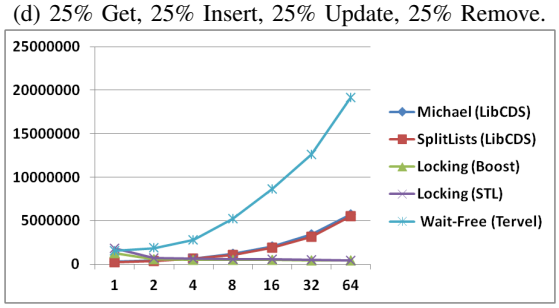
(b) 10% Get, 88% Insert, 2% Remove.



(c) 34% Get, 33% Insert, 33% Remove.

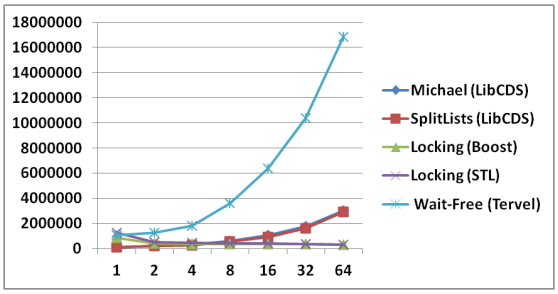


(d) 25% Get, 25% Insert, 25% Update, 25% Remove.

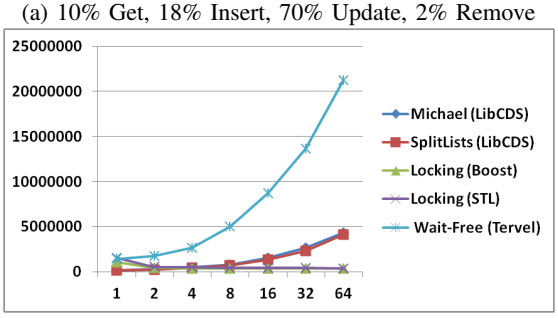


(e) An average of all tested scenarios.

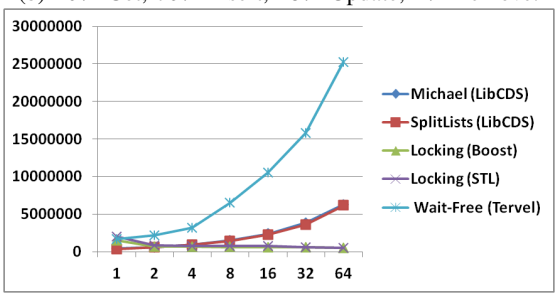
Figure 9: Graphs of number of operations versus number of threads for different operation mixes.



(a) 10% Get, 18% Insert, 70% Update, 2% Remove



(b) 10% Get, 70% Insert, 18% Update, 2% Remove.



(c) 88% Get, 8% Insert, 2% Update, 2% Remove

Figure 10: Graphs of number of operations versus threads.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier Science, 2011.

[11] M. Herlihy, "Wait-Free Synchronization," in *Trans. on Programming Languages and Systems*. ACM, 1991, pp. 124–149.

[12] ——, "A methodology for implementing highly concurrent data objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 5, pp. 745–770, Nov. 1993.

[13] ——, *The art of multiprocessor programming*. Amsterdam London: Elsevier/Morgan Kaufmann, 2008.

[14] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," *SIGPLAN Not.*, vol. 47, no. 8, pp. 141–150, Feb. 2012.

[15] MemSQL, MemSQL v3.0 (http://www.memsql.com). Retrieved 02/02/2014.

[16] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM Press, 2002, pp. 73–82.

[17] ——, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, pp. 491–504, June 2004.

[18] O. Shalev and N. Shavit, "Split-ordered lists: lock-free extensible hash tables," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2003, pp. 102–111.

[19] G. Tanase, A. Buss, A. Fidel, H. Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 235–246.

[20] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," *SIGPLAN Not.*, vol. 47, no. 8, pp. 309–310, Feb. 2012.