A Universal Construction for Wait-Free Transaction Friendly Data Structures

Phong Chuong
Dept. of Computer Science
University of Toronto
Toronto, ON, Canada
chuongph@cs.utoronto.ca

Faith Ellen
Dept. of Computer Science
University of Toronto
Toronto, ON, Canada
faith@cs.utoronto.ca

Vijaya Ramachandran
Computer Science Dept.
University of Texas at Austin
Austin, TX, USA
vlr@cs.utexas.edu

ABSTRACT

Given the sequential implementation of any data structure, we show how to obtain an efficient, wait-free implementation of that data structure shared by any fixed number of processes using only shared registers and CAS objects. Our universal construction is transaction friendly, allowing a process to gracefully exit from an operation that it wanted to perform, and it is cache-efficient in a multicore setting where the processes run on cores that share a single cache. We also present an optimized shared queue based on this method.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; Lists, stacks, and queues

General Terms

Algorithms, Theory

Keywords

Universal construction, wait-free, abortable data structure, transaction friendly, cache-efficiency

1. INTRODUCTION

In a recent CACM article, Maurice Herlihy [12] said, "For the foreseeable future, concurrent data structures will lie at the heart of multicore applications, and the larger our library of scalable concurrent data structures, the better we can exploit the promise of multicore architectures". One way to obtain a large variety of provably correct concurrent data structures is to have methods for automatically constructing them from sequential implementations. These are called universal constructions.

A universal construction is *wait-free* if every process can complete its operation on the shared data structure within a finite number of its own steps, no matter how other processes are scheduled. It is *non-blocking*, a less stringent condition, if the operation of some process is completed within a finite number of steps. Non-blocking and wait-free universal constructions were first introduced by Herlihy [10], who proved that p-consensus objects and registers are sufficient to implement any sequentially specified data structure in a shared-memory system with p or fewer processes. Since then, a variety of other non-blocking and wait-free universal constructions have been proposed. (See Section 2.)

A transaction is a collection of shared memory operations, that either all fail (without changing the shared memory) or all succeed, as an atomic operation. Failure may occur as a result of a conflict between transactions, or because a process that is performing the transaction decides not to complete it. We call a universal construction *transaction friendly* if a process can exit from (or 'abort') an uncompleted operation on the shared data structure that it no longer wishes to perform. This might happen, for example, if a process decides not to complete an operation on the shared data structure because it observes too much contention.

If each process is responsible for performing its own operation, it is often easy for a process to exit from an operation that it has not yet completed. In wait-free universal constructions, where processes may help one another complete their operations, difficulties can arise. In particular, it is necessary to ensure that there is no other process that is still trying to help perform that operation and might eventually succeed in doing so.

In this paper, we present the first wait-free universal construction that is transaction friendly. It only performs read, write, and compare&swap (CAS) on relatively small single records, and hence can be implemented on existing machines. We represent the sequential data structure directly and apply operations to it in-place. A feature of our implementation is that queries (i.e. operations that do not change the sequential data structure) do not change our representation of the shared data structure. If a sequential data structure has size s and is shared by p processes, we use a total of $\Theta(s+p)$ words of shared memory. If t is the worst case time complexity to perform an operation on the sequential data structure and w is the maximum number of different words of memory accessed by an operation on the sequential data structure, then the worst case number of steps a process takes to perform an operation on the shared data structure is $\Theta(pt \log w)$, of which $\Theta(pw)$ are shared memory accesses.

When implemented on a multicore with p cores and a shared cache, the cache complexity of our algorithm matches the sequential cache bound Q for any sequence S of data structure operations in the following sense. Suppose the memory is organized into blocks and f is the worst-case number of blocks read when performing a single operation. If the shared cache can hold at least $4pf$ blocks, then the cache complexity of our universal construction when executing S remains $O(Q)$. Here, each operation in S can be invoked by an arbitrary process.

Like previous wait-free universal constructions, our universal method applies operations to the data structure one at a time. For the special case of a wait-free queue, we have an optimized implementation that allows a process to perform an enqueue concurrently with a process performing a dequeue. It also allows multiple processes to perform dequeue simultaneously if the queue is empty. Care is needed to ensure no bad interactions arise between concurrent operations.

The remainder of the paper begins with a description of previous non-blocking and wait-free universal constructions. Next, we present our universal construction and a sketch of its correctness proof. A full proof of correctness appears in [5, 6]. This is followed by an analysis of the caching performance of our universal construction on a multicore machine. Finally, we give a brief overview of our optimized implementation of a shared queue. We conclude with a discussion of future directions.

2. RELATED WORK

A classical approach to construct concurrent data structures is to use mutual exclusion to ensure that only one process accesses the data structure at a time. The problem here is that if a process crashes while it is in the critical section, further accesses to the data structure are blocked.

A more recent approach is to use software transactional memory [8, 14, 15], which provides support for implementing transactions. Each operation on a shared data structure can be treated as a separate transaction. A transactional memory system ensures that successful transactions do not interfere with one another, and hence are correct, even if they are performed simultaneously. A process whose transaction has failed may decide to continue to retry the transaction; however, it is possible that it never succeeds. Moreover, because of their generality, transactional memory systems may incur a lot of overhead as compared to a universal construction.

Herlihy's original approach [10] was to represent the data structure by a shared linked list of its states, as a sequence of operations are performed. To perform an operation, a process appends a record, containing the operation and its inputs, to the end of the shared list. Processes use a p-consensus, CAS, or LL/SC object stored in each record to agree on the record that will follow it in the list. When a record has been appended to the shared list, the sequence number of the operation (i.e., the distance of the record from the beginning of the list), the state of the data structure after the operation has been applied, and the output of the operation, are also stored in the record. For nondeterministic operations, processes must also agree on the new state and the output of the operation. To achieve wait-freedom, each process i begins an operation by creating a record containing the operation and announces it by storing a pointer to it in location i of an announce array. This record is given priority to be the m'th record of the list whenever $i = m \bmod p$. All processes help apply the operation in the last record of the list before they try to append a new record to the list. The paper also describes how a counter in each record and pointers to the last record each process has accessed can enable a process to determine when the record will no longer be accessed, and hence can be reused. This is important to prevent the space used by the shared data structure from becoming unbounded, but increases the time overhead. In this implementation, the worst case number of steps a process takes to perform its operation on the shared data structure is $\Theta(p^2 + ps + pt)$, of which $\Theta(p^2 + ps)$ are shared memory accesses. The number of records used by this implementation is $\Theta(p^3)$, each of which contains a copy of the entire data structure (of size at most s), an $O(\log p)$-bit counter, and an unbounded sequence number.

Another wait-free universal construction [9, 11], also by Herlihy, uses a CAS (or LL/SC) object *root* that points to a record containing the current state of the data structure. To apply an operation to the data structure, the process makes a private copy, L, of the record, applies the operation to L, and then tries to change *root* to point to L. Each process has a finite collection of at most p records. When it needs a new record into which to copy the data structure, it selects one that is not being used by any other process. To facilitate this, each record stores the number of processes currently using it. Before using the current record, a process must increment this count and then check that the record is still current. After a process finishes using a record, it must decrement its counter. In [11], wait-freedom is achieved by operation combining: After announcing its operation, a process looks through the announce array for uncompleted operations and applies all of them to L, before trying to change *root* to point to L. If it is unsuccessful, it does this entire procedure again. Whether or not it is successful the second time, its operation is guaranteed to have been applied to the share data structure. Wait-freedom can also be achieved by having a mod p counter in *root* [9]. The value of the counter is the index of the process whose announced operation is to be given priority. It is incremented whenever the pointer in *root* is changed. Both variants use $\Theta(p^2)$ records, each of size $s + \Theta(\log p)$ and, in the worst case, a process does $\Theta(p^2 + ps + pt)$ shared memory accesses to perform its operation. For large linked data structures, when performing an operation, it suffices to copy only those parts of the data structure that are changed (plus, recursively, any parts that point to them). However, because storage that a process allocates may need to remain in the data structure, the memory management is more complicated.

To improve efficiency for large objects, Anderson and Moir [2] use an additional level of indirection. The data structure is viewed as being stored in a large array, which is divided into a fixed number of blocks and the LL/SC object *root* points to an array of pointers, which point to these blocks. When a process wants to write to a block, it makes a private copy of that block, and writes to its copy. Subsequent reads and writes of that block by the process are performed on its private copy. When the operation is complete, the process tries to change *root* to point to a new array containing pointers to these private blocks and all of the unmodified blocks, so that the private blocks become part of the array. If there are b blocks and each operation in the sequential implementation updates at most w' blocks, then their construction uses $\Theta(s + p^2 + pb + pw's/b)$ space and a process performs $\Theta(pb + p^2t + pw's/b)$ shared memory accesses in the worst case to perform its operation.

Barnes [4] describes a non-blocking universal construction based on LL/SC that does not require copying large parts of the shared data structure. Moreover, it allows different processes to perform operations on different parts of the data structure simultaneously. When a process wants to perform an operation on the shared data structure, it copies the variables it needs to access into local memory, to create a cached copy of the relevant parts of the data structure, and applies the operation on its cached copy. Then the process tries to lock all the variables in the shared data structure that it has cached and, if successful, changes all of the variables in the shared data structure that it changed locally. Finally, whether it is successful or not, the process releases all of its locks. If it was unsuccessful, it starts its operation over again. To ensure that *some* process will be successful, all processes try to lock variables in the same order. To prevent a process that has crashed from locking out all other processes, a process that encounters a locked variable will help the process j that locked it, by trying to lock the rest of the variables that j cached and, if successful, updating those variables that have changed values. This requires each process to write a list of the at most w variables in its cache, together with their original and final values, into shared memory. The total space used by this construction is $\Theta(s + pw)$. In the worst case, a process requires $\Theta(t)$ steps for each attempt it makes to perform its operation, including $\Theta(w)$ shared memory accesses. However, because this implementation is not wait-free, the number of attempts a process makes to perform its operation is unbounded.

Afek, Dauber, and Touitou [1] present two wait-free universal constructions that are more efficient than Herlihy's. In their Group Update algorithm, active processes (i.e. those that have operations they wish to perform) maintain a dynamic list of their identifiers stored at the root of a full binary tree of height $\log p$. Like [9, 11], there is a CAS or LL/SC object that points to the shared data structure. To perform an operation, a process announces it in an announce array and then tries to add itself into the list, while also helping other processes. This takes $\Theta(\min\{p, k \log k\})$ shared memory accesses, where k is the contention. After its identifier has been inserted into the list, a process copies the data structure to a new region of shared memory and does operation combining, applying all of the uncompleted operations of the processes on the dynamic list to the shared data structure. Then it attempts to update the pointer to the shared data structure to point to its updated version. If it was unsuccessful and its operation was not applied, the process repeats this set of steps a second time, after which its operation is guaranteed to have been performed. Finally a process removes itself from the dynamic list. The total space used is $\Theta(ps + p^2 \log p)$ and the worst case number of shared memory accesses performed by a process to perform its operation is $\Theta(\min\{p, k \log k\} + kt + s)$. Jayanti [13] observed that the step complexity of inserting a process into the list and removing a process from the list can be improved to $\Theta(\log p)$, but under the unrealistic assumption that a word of memory can store p identifiers and be accessed in a single step.

In Afek Dauber and Touitou's Individual Update algorithm, processes maintain a queue of active processes represented by a tree. After announcing an operation, a process enters the queue and moves towards the root. When a process reaches the root, its operation is applied to the data structure. While moving to the root, a process helps at most $k - 1$ other processes along the way to reach the root and apply their operations to the shared data structure. Depending on how operations are applied to the data structure, the worst-case step complexity of performing an operation is either $\Theta(ks + kt)$ or $\Theta(kt \log t)$ and requires either $\Theta(ps)$ or $\Theta(s + pt)$ words of shared memory.

More recently, Fatourou and Kallimanis [7] improved the Group Update algorithm, using two trees in a clever way, instead of one, to obtain space complexity $\Theta(ps + p^2)$ and worst-case step complexity $\Theta(\min\{k, \log p\} + kt + s)$, including $\Theta(k + s)$ shared memory operations. For large objects, they combined this universal construction with Anderson and Moir's construction, to improve the time complexity of Anderson and Moir's construction, without increasing its space complexity.

3. A UNIVERSAL CONSTRUCTION

In this section, we present a universal construction that creates a wait-free, linearizable implementation of an object from a deterministic sequential implementation.

We consider a system of p processes that communicate through a shared memory containing registers (which support read and write) and CAS objects (which support CAS, also known as compare&swap, and read), each of which is large enough to hold a small amount of information, for example pointers, names of operations, words from the sequential implementation, or arguments of operations, plus a sequence number. In addition, each process has a private local memory.

Section 3.1 describes algorithm Perform($op_i, input_i$), which is executed by process i when it needs to perform an operation on the given data structure. This execution includes a macro Help in which process i accesses the shared data structure to perform either its own operation or the operation of another process. The details of Help are deferred to section 3.2. Together, Perform and Help constitute our universal construction.

3.1 Procedure Perform

A process i begins an operation by reading *gate* on line P1 and announcing the operation in the CAS object $A[i]$, its element of the *announce array* A, on line P3.

The CAS object *gate* controls access to the stored representation of the shared data structure. It has two fields, *proc* and *seq*, which are initially $-$ and 1, respectively. When $gate.proc = i$, all processes help process i perform its announced operation on the shared representation. When this operation has been completed, *gate.proc* is reset to $-$ and *gate.seq* is incremented, on line P21 or F3. Thus the operation performed when $gate.seq = k$ is the k^{th} operation applied to the shared data structure.

The announce array A is an array of CAS objects indexed by process id, in which a process announces an operation it wants to perform on the shared data structure. Each component, $A[i]$, has four fields: *op*, *seq*, *flag* and *arg*. The type of the operation is stored in $A[i].op$. The value of *gate.seq* read by process i immediately before announcing an operation is stored in $A[i].seq$. We show that each time process i announces a new operation, *seq* is larger, so we can use $(i, A[i].seq)$ to identify this operation. The status of process i's current operation is stored in *flag*: *done* indicates that process i has not yet announced an operation or the current operation has been applied to the data structure;

active indicates that the current operation has not yet been applied; and *exit* indicates that process i has requested that the current operation not be performed, if it has not already been assigned a position in the linearization. The input arguments of the current operation are stored in $A[i].arg$, if $A[i].flag$ contains *active* or *exit*, and the output arguments are stored there if $A[i].flag = done$. If there are a lot of input or output arguments, $A[i].arg$ can, instead, contain a pointer to a list of arguments.

After announcing its operation, if process i sees, on line P11, that $gate = (-, k)$, then it helps to choose the process whose operation will be performed next. It gives priority to process $k \bmod p$ as in [10], to ensure wait-freedom. In macro ChooseNextOp, process i sees whether process $k \bmod p$ has an announced operation to be performed, by checking whether $A[k \bmod p].flag = active$ on line C3. If so, process i will decide to nominate process $k \bmod p$. Otherwise, it will check that its own operation has not been completed, on lines C4-C5, and decide to nominate itself, on line C6. Process i nominates process m_i by performing a CAS on line P13 that tries to change *gate* from $(-, k)$ to (m_i, k). It will read *gate* on line P14 to find out which process was chosen.

To help perform process $g_i.proc$'s announced operation, process i begins by reading $A[g_i.proc]$ into a_i, on line P16. It first checks if $a_i.flag = done$, on line P17, and, if so, does not need to help this operation, since it has been completed. To avoid an announced operation from being performed more than once, process i checks whether $a_i.seq \leq g_i.seq$ on line P17. If not, then g_i contains an old value of *gate*. In this case, the $g_i.seq^{th}$ operation in the linearization has been completed and process i will not perform Help. If $a_i.seq \leq g_i.seq$ and $a_i.flag \neq done$, then process i will execute Help on behalf of operation $(g_i.proc, a_i.seq)$. Afterwards, on line P19 or P20, process i writes the outputs of the operation into $A[g_i.proc].arg$ and changes $A[g_i.proc].flag$ to *done* using a single CAS. Finally, process i tries to update *gate* to $(-, g_i.seq{+}1)$ on line P21, so that the next operation can be performed.

Whenever a process i wants to exit its announced operation, it changes $A[i].flag$ to *exit* on line P6, so that other processes do not choose this operation, even if i has priority to be chosen. However, process i does not return with *exited* immediately after setting $A[i].flag$ to *exit*. The reason for this is that some other processes may already be in the midst of helping to perform this operation. Instead, after changing $A[i].flag$ to *exit*, process i executes the rest of the iteration. We show that, by the time process i finishes the iteration, *gate.seq* will have been incremented. Thus, when any process tries to help perform a subsequent operation, it reads $A[i]$ after $A[i].flag$ was set to *exit*. Since $A[i].flag$ does not become *active* until process i announces another operation on line P3, the exited operation will not be performed.

3.2 Procedure Help

When a process i performs Help, it tries to apply the operation $(g_i.proc, a_i.seq)$ as the $g_i.seq^{th}$ operation in the linearization. Help uses the caching mechanism in [4]: When a process first accesses a variable from the shared data structure during an invocation of Help, it caches it. Subsequent reads and writes of a cached variable are performed locally. After the operation has been completed locally, the process updates those records in the shared data structure whose values should be changed.

```
Perform($op_i$, $input_i$) by process $i$:

 P1.  $g_i \leftarrow$ read $gate$
 P2.  $a_i \leftarrow (op_i, g_i.seq, active, input_i)$
 P3.  $A[i] \leftarrow$ write $a_i$
 P4.  while $a_i.flag = active$ do
 P5.    if process $i$ should exit its operation then
 P6.      $CAS(A[i], a_i, (a_i.op, a_i.seq, exit, input_i))$
 P7.      $a_i \leftarrow$ read $A[i]$
 P8.      if $a_i.flag = done$ then
 P9.        Finish()
          end if
        end if
P10.    $g_i \leftarrow$ read $gate$
P11.    if $g_i.proc = -$ then
P12.      ChooseNextOp()
P13.      $CAS(gate, g_i, (m_i, g_i.seq))$
P14.      $g_i \leftarrow$ read $gate$
        end if
P15.    if $g_i.proc \neq -$ then
P16.      $a_i \leftarrow$ read $A[g_i.proc]$
P17.      if $a_i.flag \neq done$ and $a_i.seq \leq g_i.seq$ then
P18.        Help()
P19.        $CAS(A[g_i.proc], (a_i.op, a_i.seq, active, a_i.arg),$
              $(a_i.op, a_i.seq, done, output_i))$
P20.        $CAS(A[g_i.proc], (a_i.op, a_i.seq, exit, a_i.arg),$
              $(a_i.op, a_i.seq, done, output_i))$
          end if
P21.      $CAS(gate, g_i, (-, g_i.seq + 1))$
        end if
P22.    $a_i \leftarrow$ read $A[i]$
      end while
P23.  if $a_i.flag = exit$ then return($exited$) end if
P24.  Finish()
```

```
Finish() by process $i$:

 F1.  $g_i \leftarrow$ read $gate$
 F2.  if $g_i.proc = i$ then
 F3.    CAS($gate, g_i, (-, g_i.seq + 1)$) end if
 F4.  return($a_i.arg$)
```

```
ChooseNextOp() by process $i$:

 C1.  $m_i \leftarrow g_i.seq \bmod p \in \{0, \ldots, p-1\}$
 C2.  $a_i \leftarrow$ read $A[m_i]$
 C3.  if $a_i.flag \neq active$ then
 C4.    $a_i \leftarrow$ read $A[i]$
 C5.    if $a_i.flag = done$ then Finish() end if
 C6.    $m_i \leftarrow i$
      end if
```

We represent each variable x in the sequential representation by a record, R_x, in the shared representation. A record is a CAS object with four fields: $val[0]$, $val[1]$, $toggle$ and seq. The fields $val[0]$ and $val[1]$ have the same type as x and the field $toggle$ is a single bit which indicates whether $val[0]$ or $val[1]$ is the current value of x. Whenever process i writes to the shared record, R_x, representing variable x, it stores the new value in $R_x.val[1 - R_x.toggle]$, complements $R_x.toggle$, and sets $R_x.seq$ to $g_i.seq$. When a process i reads R_x, it compares $R_x.seq$ with $g_i.seq$ to determine which of $R_x.val[0]$ and $R_x.val[1]$ to use. If $R_x.seq < g_i.seq$, then it uses $R_x.val[R_x.toggle]$ to get the current value of x, since no other process that is performing the operation has changed R_x. If $R_x.seq = g_i.seq$, then R_x has already been updated for the operation, so process i uses $R_x.val[1 - R_x.toggle]$ to get the previous value of x. If $R_x.seq > g_i.seq$, then the operation has already been completed. In this case, process i can stop performing the operation.

Process i begins Help by resetting its local dictionary D_i on line H1. Each entry in this dictionary is a triple containing the name of the variable (which serves as the key), the contents of the corresponding record in the shared representation of the data structure, and the current value of this variable. The second field is used to perform a CAS on the record if the value of the variable is changed by the operation. When process i accesses a variable x in the simulation of the sequential implementation, it checks, on line H20, to see if x is in the local dictionary. If x is not in the local dictionary, then, on line H21, process i reads the corresponding record, R_x, from the shared representation of the data structure and, on line H24 or H25, adds an entry to the dictionary for x. If the access is a read from x, then, on line H27, process i uses the local value of x that is stored in the dictionary. If the access is a write to x, then, on line H28, process i updates the local value of x to the new value. When process i uses an input in the simulation of the sequential implementation, then, on line H29, process i uses the corresponding value in $a_i.arg$. When process i produces an output value, then, on line H30, it writes the value to the corresponding location in $output_i$.

```
Help() by process $i$:

H1.  $R_i \leftarrow \phi$
H2.  $nl_i \leftarrow$ read $nl$
H3.  if $nl_i.seq > g_i.seq$ then exit Help end if
H4.  if $nl.seq < g_i.seq$
H5.  then $next_i \leftarrow nl_i.ptr[nl_i.toggle]$
H6.  else $next_i \leftarrow nl_i.ptr[1 - nl_i.toggle]$
     end if
     % Locally perform all the steps of operation $a_i.op$
     % with inputs $a_i.arg$, reading the record associated
     % with each variable from the shared object prior to
     % its first access by that operation.
H7.  for each access of a variable with name $x$ in the code
     for operation $a_i.op$ do
H8.    if $x$ is the name for a new variable in the code for
       $a_i.op$ then
H9.      $r_i \leftarrow$ read record pointed to by $next_i$
         % if $next_i$ points to the last record in $newlist$,
         % try to append a new record to $newlist$.
H10.     if $r_i = (-, -, 0, 0)$ then
H11.       CAS(record pointed to by $next_i$, $r_i$, $(new_i, -, 0, 0)$)
H12.       $r_i \leftarrow$ read record pointed to by $next_i$
H13.       if $r_i = (new_i, -, 0, 0)$ then
             % the record pointed to by $new_i$ was
             % successfully appended to $newlist$
H14.         get a new record from the memory manager
H15.         let $new_i$ point to this record
H16.         initialize this record to $(-, -, 0, 0)$
           end if
         end if
H17.     if $r_i.seq > g_i.seq$ then exit Help end if
H18.     add $(x, r_i, 0)$ to $D_i$
H19.     $next_i \leftarrow r_i.val[0]$
       end if
H20.   if there is no item with key $x$ in $D_i$ do
H21.     $r_i \leftarrow$ read record, $R_x$, associated with variable $x$
H22.     if $r_i.seq > g_i.seq$ then exit Help end if
H23.     if $r_i.seq < g_i.seq$
H24.     then add $(x, r_i, r_i.val[r_i.toggle])$ to $D_i$
H25.     else add $(x, r_i, r_i.val[1 - r_i.toggle])$ to $D_i$
         end if
       end if
H26.   let $(x, r, v)$ be the (unique) item with key $x$ in $D_i$
H27.   to read from $x$, use the value $v$
H28.   to write $v'$ to $x$, replace $(x, r, v)$ by $(x, r, v')$ in $D_i$
H29.   for inputs, use $a_i.arg$
H30.   place outputs in $output_i$
     end for
     % write changed records to the shared object:
H31. for each $(x, r, v) \in D_i$ do
H32.   if $r.seq < g_i.seq$ and $r.val[r.toggle] \neq v$ then
H33.     if $r.toggle = 0$
H34.     then $r' \leftarrow (r.val[0], v, 1, g_i.seq)$
H35.     else $r' \leftarrow (v, r.val[1], 0, g_i.seq)$
         end if
H36.     $CAS(R_x, r, r')$
       end if
     end for
     % update $nl$
H37. if $nl_i.seq < g_i.seq$ then
H38.   if $nl_i.toggle = 0$
H39.   then $nl'_i \leftarrow (nl'_i.ptr[0], next_i, 1, g_i.seq)$
H40.   else $nl'_i \leftarrow (next_i, nl'_i.ptr[1], 0, g_i.seq)$
       end if
H41.   $CAS(nl, nl_i, nl'_i)$
     end if
```

After process i complete its simulation of the operation, on line H31, it considers each entry (x, r, v) in the dictionary. If the record has not already been updated by another process performing the same operation (i.e., if $r.seq < g_i.seq$) and the value of x was changed during the simulation of the operation (i.e., $r.val[r.toggle] \neq v$), on line H32, then process i performs the CAS on line H36 to update R_x in the shared data structure. Lines H33–H35 ensure that the value of $toggle$ is changed and the new value of v is put in the correct field.

It remains to describe what happens during the simulation of an operation when the sequential implementation allocates a new variable, x. It would be problematic to have each process that tries to perform this operation allocate a different record for x in the shared data structure. To avoid this, we maintain a nonempty shared list, *newlist*, of records that can be allocated when the operation is applied. The first field of each record in *newlist*, $val[0]$, points to the next record in the list. If $r.val[0] = -$, then r is the last record in *newlist*. The shared variable in *nl* contains a pointer to the first record in *newlist*. The first time a value is written to a newly allocated record, it is written into $val[1]$. This allows other processes performing the operation to find the next element in *newlist* by following the pointer stored in $val[0]$ when that the test on line H17 is unsuccessful. If the test is successful, then process i can infer that the operation has already been completed and it can stop performing the operation. When a process uses the last record in *newlist* (i.e., if line H10 is successful), it immediately tries, on line H11, to append a new record to follow it. Each process i has one unallocated record, pointed to by new_i, that it can use for this purpose. If it successfully appends this record to the end of *newlist* (i.e., if line H13 is successful), then it gets a new record from the shared memory manager to replace it, on line H14–H16.

All processes performing an operation use the j^{th} record in *newlist* for the j^{th} new variable that is allocated during that operation. Process i keeps a pointer, $next_i$, which points to the record it should use for the next variable that needs to be allocated during the simulation of the operation. On lines H4–H6, $next_i$ is initialized to the beginning of *newlist*. To use the record for a newly allocated variable x, process i associates the name x with this record by adding a new entry with key x, a copy of the contents of the record to its local dictionary and the initial value 0, on line H18. Then, on line H19, it assigns $next_i$ to the next element in *newlist*. Subsequent accesses to x are performed on the local copy in D_i. At the end of the operation, *nl* is updated to point to the first unused record in *newlist*, on lines H37–H41. If the sequential implementation releases variables, the corresponding records could be added to *newlist*, so it serves as a free list.

A problem could arise if another process tries to perform the operation after *nl* has been updated, because it would use different records for the newly allocated variables. In this case, although the operation has been completely applied to the shared data structure, it is possible that the output of the operation has not been stored in its announcement. To avoid this problem, *nl* consists of four fields, $ptr[0]$, $ptr[1]$, *toggle* and *seq*. The fields $ptr[0]$ and $ptr[1]$ are pointers to records, *toggle* is a bit which indicates whether $ptr[0]$ or $ptr[1]$ points to the beginning of *newlist* and *seq* contains the value of *gate.seq* when *nl* was last updated on line H41. Before performing the operation, process i checks whether $nl_i.seq > g_i.seq$ on line H3. If so, then the operation has been completed and process i can stop performing the operation. If not, process i checks, on line H4, whether $nl_i.seq = g_i.seq$ to determine whether *nl* has been updated for this operation and hence, which of $ptr[0]$ or $ptr[1]$ to use.

3.3 Correctness

We say that the shared data structure at a particular configuration *correctly represents* a state of the sequential data structure if for all variables x in the sequential data structure, $R_x.val[R_x.toggle]$ is the value of x in that state.

For every execution, we linearize each operation announced by process i if and when $A[i].flag$ is changed to *done* between its announcement and when process i announces its next operation. Suppose we apply each linearized operation up to and including this one to the sequential data structure. We prove that the output of this operation is the same in both the shared implementation and the sequential implementation, and the shared data structure at the linearization point of this operation correctly represents the state of the sequential data structure after these operations have been performed.

We also prove that if process i invokes Perform(op_i, $input_i$), it will return after executing at most $p+1$ iterations of the while loop and either:
process i returns on line F4 after $A[i].flag$ is set to *done*, or process i returns on line P23 after setting $A[i].flag$ to *exit* on line P6, and $A[i].flag$ is not set to *done* between when the operation was announced and process i announces its next operation.

We first establish some basic properties of the shared variables that follow from observation of the code.

- When $gate = (-, k)$, it can only change to (i, k) for some $i \in \{0, \ldots, p-1\}$.
- When $gate = (i, k)$, it can only change to $(-, k+1)$.
- When $A[i].flag =$ *active*, it can only change to *done* or *exit*.
- When $A[i].flag =$ *done*, it can only change to *active*.
- $A[i].flag$ can change to *done* only when $gate.proc = i$.
- $A[i].flag$ can change to *active* only when process i performs P3.
- $A[i].seq \leq gate.seq$.
- $R_x.seq \leq gate.seq$, for every variable x in the sequential implementation.
- $nl.seq \leq gate.seq$.

Then the following lemma establishes key invariants needed to prove correctness.

Lemma 1.

1. *Whenever gate.proc changes from i to $-$, $A[i].flag =$ done.*
2. *After process i performs line P3:*
   (a) if it next returns on line P23, then gate.proc does not change to i
   (b) if it next returns on line F4, then gate.proc changes to i exactly once, and
   (c) between when process i next returns and when it performs line P3 after that, $gate.proc \neq i$.
3. *Between when $A[i].flag$ changes to done and when process i last performed line P3 prior to that, gate.proc changes to i exactly once.*

Consider any execution of our algorithm that ends in a configuration in which $gate.seq = k + 1$. Then at least k operations have been linearized. For any variable x, let v be its value in the sequential implementation immediately after the first k operations in this linearization have been performed. Then $R_x.val[R_x.toggle] = v$, if $R_x.seq < k+1$, and $R_x.val[1 - R_x.toggle] = v$, if $R_x.seq = k+1$. Similarly, either $nl.ptr[nl.toggle]$ or $nl.ptr[1 - nl.toggle]$ points to the first element in *newlist*, if $nl.seq < k+1$ or $nl.seq = k+1$, respectively.

We prove that, after i's operation has been announced in $A[i]$, *gate.proc* changes to i mod p within $p+1$ iterations of the while loop by process i. When that occurs, every process executing the while loop will help process i to complete its operation (if it has not yet been completed) by performing Help and then will try to update $A[i]$ so that $A[i].flag$ will be set to *done* and *arg* will contain the output of the operation.

We also show that if process i attempts to set $A[i].flag$ to exit on line P6, then it exits Perform within one iteration of the loop on lines P4–P22: If the CAS on line P6 is unsuccessful, then we prove that $A[i].flag$ has been changed to *done* and process i's operation has been linearized. If the CAS on line P6 is successful, then $A[i].flag$ is changed to *exit* and we prove that it does not change back to *active* until process i next announces another operation. Hence, when process i next performs line P22 at the end of the iteration, $A[i].flag \neq active$ and the test on line P4 is unsuccessful.

We prove that, immediately after process i performs Help for an operation that has not yet been linearized, then $output_i$ contains the output of that operation applied to the sequential data structure. For every variable x, we show that:

–if x is allocated, then there is a unique record, R_x, that corresponds to x in the shared data structure,

–if it is not changed during the sequential implementation of the operation, then R_x remains unchanged, and

–if it is changed to v, then $R_x.val[r_x.toggle] = v$.

In addition, we prove that $nl.ptr[nl.toggle]$ points to an unallocated record.

3.4 Complexity

During each iteration, process i executes $O(1)$ steps in addition to executing Help at most once. If every sequential operation accesses at most w variables and the local dictionary of each process is implemented using, for example, a red-black tree, it takes $O(\log w)$ steps, in the worst case, to cache a variable or access a cached variable. (This takes $O(1)$ expected time using a hash table with chaining.) If every sequential operation takes time at most t, the worst case number of steps a process takes to perform Help is $O(t \log w)$, of which $\Theta(w)$ are shared memory accesses.

If process i does not want to exit its operation, then it executes at most $p+1$ iteration of the loop on lines P4–P22. Hence a process takes $O(pt \log w)$ steps during Perform. If process i wants to exit its operation, then after it performs line P5, it does not perform another complete iteration of the loop and, hence, takes $O(t \log w)$ more steps before it returns.

For each variable x in the sequential implementation, there is exactly one record, R_x,that corresponds to x in the shared representation. In addition, each process, i, has a location $A[i]$ in the announce array and has one unallocated record, pointed to by new_i. It follows that the resulting shared implementation of the data structure uses $s + O(p)$ shared objects, where s is the size of the sequential data structure.

4. CACHING PERFORMANCE

Consider the execution of Perform on a sequence of r operations of the shared data structure on a multicore with p cores (each corresponding to a process), which share a cache of size M arranged in blocks of size B. We assume LRU. We will say that a caching complexity $R(r, M, B)$ is *well-behaved* if $R(r, M, B) = \Theta(R(r, \Theta(M), B)$. (All commonly known cache complexities are well-behaved.)

LEMMA 2. *Let the shared data structure bring in at most $Q(r, M, B)$ blocks into cache and incur at most $f(r, M, B)$ cache misses for any single operation in its sequential execution of a sequence S of r operations.*

If $M \geq 4 \cdot p \cdot B \cdot f(r, M, B)$, and Q and f are well-behaved cache complexities, then the number of cache misses incurred by any execution of Perform on the sequence S is $O(Q(r, M, B))$.

As an example, a stack or queue implemented as a standard semi-infinite array has $Q(r, M, B) = r/B$, $f(r, M, B) = 1$, hence by the above lemma, the number of cache misses remains $O(r/B)$ when executed with Perform if $M \geq 4pB$. (Typically, we will have $M \gg p \cdot B$). Note that $Q(r, M, B) = \Omega(r/B)$ for any queue or stack — consider a queue (the stack is similar) with a sequence of r operations which alternate between $2M$ enqueues and $2M$ dequeues. Thus this result is optimal for stacks and queues in a strong sense.

Lemma 2 can be proved as follows. The announce array $A[1..p]$, the most recent value of *gate*, and the data read for the last operation, which together require no more than $3f(r, M, B)$ blocks in cache (assuming *seq* and *toggle* are in one word) will be in cache, as will data read for other recent operations. Additionally, the cache may contain data used in older operations, which are being accessed by slow processes that have not yet read the more recent value of *gate*. Even if all but one process is executing slowly, there is at least $M' = M - p - 3pBf(r, M, B)$ space available in the cache for the shared records accessed by recent operations. We have $M' = \Theta(M)$ since $M \geq 4pBf(r, M, B)$.

The main additional source for cache misses occurs with a process that slows down considerably. Consider a slow process i. It may incur additional $O(f(r, M', B))$ cache misses for its current operation op_i if the data is so old that it has been evicted from the cache. But if the data is this old, then at least M'/B blocks of data were brought into cache since the time when op_i was executed by the fast processes. Further, process i will complete its execution of op_i with at most $3f(r, M', B)$ cache misses, and then will read the current value of *gate*. Hence, process i will incur this penalty of $3f(r, M', B)$ cache misses at most $Q(r, M', B)/(M'/B)$ times. Hence the total number of additional blocks that could be read because of slow processes (across all p processes) is $O\left(p \cdot f(r, M', B) \cdot \frac{B \cdot Q(r, M', B)}{M'}\right)$. Since $p = O(\frac{M}{Bf(r, M, B)})$ by assumption, this is $O(Q(r, M, B))$.

The caching performance looks less promising at the private cache at each core: If process i performs n_i operations, it could incur $\Theta(p \cdot n_i)$ cache misses in the worst case, in addition to the cost of reading in its private copy of the data structure values. This could occur if for each operation, process i needs to cycle through $p - 1$ iterations of helping, and needs to read in an updated value from array A each time. It is unclear how one can improve the caching performance of private caches, and determining if this is possible is left as an open question.

5. TRANSACTION FRIENDLY QUEUE

We have adapted Perform to obtain a refined version of a wait-free transaction friendly queue, implemented in the standard way as a semi-infinite array. In a queue, enqueues and dequeues have no interaction, except possibly when the queue is empty. Our queue allows an enqueue ($op = E$) and a dequeue ($op = D$) to occur concurrently, and while the queue is empty, all dequeues return within a constant number of steps with $\perp$ (to denote an empty queue).

We use Perform-Enq for enqueues and Perform-Deq for dequeues, with separate gates *egate* and *dgate*. At quiescence (i.e. when there are no enqueues or dequeues in progress), their sequence numbers give the locations of the tail and head of the queue respectively. EHelp is the Help routine for Perform-Enq and and DHelp for Perform-Deq. At each location of the queue $Q[1..\,]$ is a pair $(val[0], val[1])$, which starts with value $(-,-)$. When a value v is enqueued at $Q[i]$, its entry is updated to $(v,-)$, and when this value is dequeued from $Q[i]$, it is updated to (v,v). No further updates to this location can occur, hence we do not need the toggle bit. Since position i in the queue is also the current sequence number being used for gate (*dgate* or *egate*), we do not need the sequence number field either.

The pseudocode is in the appendix, where lines numbers with prefix P refer to the lines in our universal construction. We change ChooseNextOp slightly since, in addition to needing $a_i.flag$ to be *active* in order to determine if the operation should be performed, we also need to check if the operation is E when executing Perform-Enq, and D when executing Perform-Deq. Perform-Enq is the same as Perform except that *gate* is now *egate*, and Help is replaced by the two-line EHelp. Perform-Deq has a few changes from Perform, most notably to handle the case of dequeue when an empty queue is detected. As seen in the pseudocode in the appendix, this is handled using a mechanism similar to that used to handle an exit in the universal construction. We use a flag X_i to denote that an empty queue has been detected by process i, and then we use the method used earlier to handle exit to now handle both of these cases, and use the flag X_i to return the correct value (*exit* or $\perp$). The other main difference from Perform is that when the queue is empty, no dequeue can occur and hence there is no call to DHelp, nor should there be a change to *dgate* during this time (in contrast to the handling of exit). The correctness follows from the fact that if process i returns with $\perp$, then after X_i is set to $True$ and $A[i].flag$ is set to *exit*, either the queue is verified to be empty, or *dgate.seq* has been incremented from the value it had when $A[i].flag$ was set to *exit*. Hence, if $A[i].flag$ has not yet been set to *done*, any process that tries to help with this dequeue request of process i will find $A[i].flag$ to be *exit*, and hence will not assign process i to *dgate.proc* in ChooseNextOp.

The linearization point for an enqueue operation is when line X2 is executed, and for a dequeue when line Y2 is executed. This is in contrast to the universal construction, where the linearization point is when $A[i].flag$ is set to *done*. It is readily verified that the corresponding statement to lines X2 and Y2 in the universal construction (namely the last statement in Help) is a valid linearization point there. For our queue, it is important to linearize at these two points rather than when $A[i].flag$ is set to *done*, due to our support of concurrent execution of enqueues and dequeues.

6. DISCUSSION

In this paper, we have described a universal construction that implements any given deterministic sequential data structure as a transaction friendly, wait-free data structure shared by any fixed number of processes. The method is efficient and uses the shared cache efficiently in a multicore implementation. We have also briefly described a shared queue based on this universal construction that appears to be quite practical. In the interests of presenting a construction that is easier to understand, we did not include some natural refinements. Instead, we discuss them briefly here.

Our universal construction uses CAS objects that contain a small, constant number of fields. For example, a record R_x in the shared data structure is a CAS object that contains two values from the sequential data structure, a toggle bit, and a sequence number. This can be avoided using indirection: Instead, each CAS object could contain a single pointer that points to a record of registers. The resulting implementation would be slower and a mechanism for managing the allocation and deallocation of these records, such as a separate free list for each process, is needed.

Randomized operations can be supported in our universal construction by using a shared array, S, of CAS objects that allow the processes to agree on all of the random choices made while applying the operation: Each of these CAS objects holds a random choice, together with the current value of *gate*. When process i wants to make its k'th random choice during the operation, it reads $S[k]$. If the sequence number stored there is greater than g_i, then the operation has already been completed and the process can stop helping it. If it is equal to g_i, process i uses the random choice recorded in $S[k]$. If it is less than g_i, process i makes a local random choice, which it tries to record, together with g_i, by performing a CAS on $S[k]$. Then it reads $S[k]$ again and uses the random choice recorded there. Note that the array S is reused for each subsequent operation.

It is necessary to assume a weak adversary, which is not aware of the local random choice of a process until it next performs a shared memory operation. A strong adversary could bias the outcome of a random choice by seeing the local random choices of a number of processes and then scheduling the process whose choice it likes best. In fact, it is impossible to have a wait-free implementation of a fair coin against a strong adversary [3].

As presented, our universal construction will not perform an operation announced by process i if process i sets $A[i].flag$ to *exit* before it is chosen or given priority to be the next operation. This is useful when process i has waiting too long to perform its operation, because the operation will be terminated, either successfully or unsuccessfully, without undue delay. For more general situations, such as when a process detects a problem during the execution or detects a conflict with a concurrent operation, it would be better if the operation exits even after it has been partially applied to the shared data structure. This can be accomplished by having a process undo all the changes the operation has made to the shared data structure if it sees that the operation is supposed to exit. Specifically, P20 is replaced by lines U1 to U15.

```
U1.  $a'_i \leftarrow$ read $A[g_i.proc]$
U2.  if $a'_i.flag = exit$ and $a'_i.seq = a_i.seq$
U3.  then for each $(x, r, v) \in D_i$ do
U4.    $r' \leftarrow$ read $R_x$
U5.    if $r'.seq = g_i.seq$ then
U6.      if $r.seq < g_i.seq$
U7.      then $old \leftarrow r.val[r.toggle]$
U8.      else $old \leftarrow r.val[1 - r.toggle]$
         end if
U9.      $CAS(R_x, r', (old, old, 0, g_i.seq))$
       end for
U10.   $nl'_i \leftarrow$ read $nl$
U11.   if $nl'_i.seq = g_i.seq$ then
U12.     if $nl_i.seq < g_i.seq$
U13.     then $old \leftarrow nl_i.val[nl_i.toggle]$
U14.     else $old \leftarrow nl_i.val[1 - nl_i.toggle]$
         end if
U15.     $CAS(nl, nl'_i, (old, old, 0, g_i.seq))$
       end if
     end if
```

When process i performs this code, it first check whether the operation it just helped is supposed to exit on lines U1–U2. For each variable x whose record, R_x, was changed by the operation and has not been changed by a later operation (i.e., the test on line U5 was successful), the value of x prior to the operation is computed on lines U6–U8 and put into both $R_x.val[0]$ and $R_x.val[1]$ on line U9. $R_x.seq$ is also set to $g_i.seq$, so that a slow process still performing the operation will not modify this location. Note that process i's local dictionary D_i still contains all the records that are accessed during the operation. Correctness follows from the observation that any record R_x can change at most twice while $gate.seq$ has the same value. Similarly, nl is restored to its old value on lines U10–U15.

Currently, in Perform, a process executes lines P10–P22 after announcing that its operation should exit. We should remove line P23 and add return($exited$) after lines P8–P9, so that process i returns immediately after successfully setting $A[i].flag = exit$. Furthermore, instead of allowing a process to exit an operation only on lines P5–P9 at the beginning of an iteration of the while loop on lines P4–P22, lines P5–P9 could be put anywhere or lines P6–P9 could be executed in response to an interrupt indicating that the announced operation of process i should exit.

There are other refinements that can improve efficiency of Perform. For instance, in our current method, a process i that wants to exit and finds that its operation has not yet completed, executes one further iteration of ChooseNextOp and Help before returning. It is possible to avoid this additional iteration except when the values read for $gate.seq$ in lines P10 and P14 are congruent to i mod p; otherwise the process can exit immediately. Also, at the end of Help, we could have a pointer as a field in nl to the output needed by the operation being executed so that other (later) processes helping with the same operation do not need to execute the operation in order to determine the output.

It should be possible to avoid using sequence numbers, perhaps with standard techniques such as bounded timestamps, handshaking, or using LL/SC instead of CAS.

Currently our universal construction executes operations one at a time. We are currently trying to find ways generalize it to allow different operations to be performed concurrently on disjoint parts of the shared structure, without sacrificing correctness or wait-freedom. One approach we are considering is to have a lock, owned by an operation, associated with each record in the shared data structure, as in [4], but to give priority to operations in a way that ensures wait-freedom.

Acknowledgments. This work was supported by the Natural Science and Engineering Research Council of Canada and NSF grants CCF-0850775 and CCF-0830737.

7. REFERENCES

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. ACM SPAA*, pages 538–547, 1995.

[2] J. Anderson and M. Moir. Universal constructions for large objects. *IEEE Trans. Parallel Dist. Syst.*, 10(12):1317–1332, 1999.

[3] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, 1990.

[4] G. Barnes. A method for implementing lock-free shared-data structures. In *Proc. ACM SPAA*, pages 261–270, 1993.

[5] P. Chuong. A wait-free abortable universal construction. Master's thesis, University of Toronto, 2010.

[6] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. Manuscript, 2010.

[7] P. Fatourou and N. Kallimanis. The redblue adaptive universal construction. In *Proc. DISC*, volume 5805 of *LNCS*, pages 127–141, 2009.

[8] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[9] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc ACM PPoPP*, pages 197–206, 1990.

[10] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[12] M. Herlihy. Technical perspective - highly concurrent data structures. *Commun. ACM*, 52(5):99, 2009.

[13] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. ACM PODC*, pages 201–210, 1998.

[14] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proc. ACM PPoPP*, pages 227–236, 2008.

[15] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

```
Perform-Enq($E$, $v_i$) by process $i$:
 P1.  $g_i \leftarrow$ read $egate$
 P2.  $a_i \leftarrow (E, g_i.seq, active, v_i)$
 P3.  $A[i] \leftarrow$ write $a_i$
 P4.  while $a_i.flag = active$ do
 P5.    if process $i$ should exit its operation then
 P6.      $CAS(A[i], a_i, (a_i.op, a_i.seq, exit, v_i))$
 P7.      $a_i \leftarrow$ read $A[i]$
 P8.      if $a_i.flag = done$ then
 P9.        Finish()
          end if
        end if
P10.    $g_i \leftarrow$ read $egate$
P11.    if $g_i.proc = -$ then
P12.      ChooseNextOp()
P13.      $CAS(egate, g_i, (m_i, g_i.seq))$
P14.      $g_i \leftarrow$ read $egate$
        end if
P15.    if $g_i.proc \neq -$ then
P16.      $a_i \leftarrow$ read $A[g_i.proc]$
P17.      if $a_i.flag \neq done$ and $a_i.seq \leq g_i.seq$ then
P18.        EHelp()
P19.        $CAS(A[g_i.proc], (a_i.op, a_i.seq, active, a_i.arg),$
              $(a_i.op, a_i.seq, done, output_i))$
P20.        $CAS(A[g_i.proc], (a_i.op, a_i.seq, exit, a_i.arg),$
              $(a_i.op, a_i.seq, done, output_i))$
          end if
P21.      $CAS(egate, g_i, (-, g_i.seq + 1))$
        end if
P22.    $a_i \leftarrow$ read $A[i]$
      end while
P23.  if $a_i.flag = exit$ then return($exited$) end if
P24.  Finish()
```

```
Finish() by process $i$:

Lines F1 to F4 as in the universal construction with:
$gate = egate$ when called from Perform-Enq, and
$gate = dgate$ when called from Perform-Deq.
```

```
ChooseNextOp() by process $i$:

Lines C1 to C6 as in the universal construction with line
C3 replaced by:
In Perform-Enq: 'if $a_i.flag \neq active$ or $a_i.op \neq E$'
In Perform-Deq: 'if $a_i.flag \neq active$ or $a_i.op \neq D$'
```

```
**EHelp() by process $i$:**
 X1.  $output_i \leftarrow -$
 X2.  CAS$(Q[g_i.seq], (-, -), (a_i.arg, -))$
```

```
Perform-Deq($D, -$) by process $i$:
 P1.  $g_i \leftarrow$ read $dgate$
 D1.  $X_i \leftarrow False$
 P2.  $a_i \leftarrow (D, g_i.seq, active, -)$
 P3.  $A[i] \leftarrow$ write $a_i$
 P4.  while $a_i.flag = active$ do
 D2.    $g_i \leftarrow$ read $dgate$
 D3.    $q_i \leftarrow$ read $Q[g_i.seq]$
 D4.    if $q_i = (-, -)$ then $X_i \leftarrow True$
 D5.  [P5]
        if (process $i$ should exit) or ($X_i$) then
 P6.      $CAS(A[i], a_i, (a_i.op, a_i.seq, exit, -))$
 P7.      $a_i \leftarrow$ read $A[i]$
 P8.      if $a_i.flag = done$ then
 P9.        Finish()
          end if
        end if
P10.    $g_i \leftarrow$ read $dgate$
 D6.    $q_i \leftarrow$ read $Q[g_i.seq]$
 D7.    if $q_i \neq (-, -)$ then
P11.      if $g_i.proc = -$ then
P12.        ChooseNextOp()
P13.        $CAS(dgate, g_i, (m_i, g_i.seq))$
P14.        $g_i \leftarrow$ read $dgate$
          end if
P15.      if $g_i.proc \neq -$ then
P16.        $a_i \leftarrow$ read $A[g_i.proc]$
P17.        if $a_i.flag \neq done$ and $a_i.seq \leq g_i.seq$ then
P18.          DHelp()
P19.          $CAS(A[g_i.proc], (a_i.op, a_i.seq, active, a_i.arg),$
                $(D, a_i.seq, done, output_i))$
P20.          $CAS(A[g_i.proc], (a_i.op, a_i.seq, exit, a_i.arg),$
                $(a_i.op, a_i.seq, done, output_i))$
            end if
P21.        $CAS(dgate, g_i, (-, g_i.seq + 1))$
          end if
        end if
P22.    $a_i \leftarrow$ read $A[i]$
      end while
 D8.  [P23a] if $a_i.flag = exit$ and $X_i = True$ then
                return($\perp$) end if
 D9.  [P23b] if $a_i.flag = exit$ and $X_i = False$ then
                return($exited$) end if
P24.  Finish()
```

```
**DHelp() by process $i$:**
 Y1.  $q_i \leftarrow Q[g_i.seq]$
 Y2.  $output_i \leftarrow q_i.val[0]$
 Y3.  CAS$(Q[g_i.seq], q_i, (output_i, output_i))$
```