SNZI: Scalable NonZero Indicators

Faith Ellen
Dept. of Computer Science
University of Toronto
faith@cs.toronto.edu

Yossi Lev
Brown University and Sun Microsystems Laboratories
levyossi@cs.brown.edu

Victor Luchangco and Mark Moir
Sun Microsystems Laboratories
victor.luchangco@sun.com
mark.moir@sun.com

ABSTRACT

We introduce the SNZI shared object, which is related to traditional shared counters, but has weaker semantics. We also introduce a resettable version of SNZI called SNZI-R. We present implementations that are scalable, linearizable, nonblocking, and fast in the absence of contention, properties that are difficult or impossible to achieve simultaneously with the stronger semantics of traditional counters.

Our primary motivation in introducing SNZI and SNZI-R is to use them to improve the performance and scalability of software and hybrid transactional memory systems. We present performance experiments showing that our implementations have excellent performance characteristics for this purpose.

Categories and Subject Descriptors

E.1 [**Data Structures**]

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

Counters, scalability, transactional memory

1. INTRODUCTION

We introduce the *Scalable NonZero Indicator*, or *SNZI* (pronounced "snazzy"), a shared object that supports `Arrive` and `Depart` operations, as well as a `Query` operation, which returns a boolean value indicating whether there is a *surplus* of `Arrive` operations (i.e., whether the number of `Arrive` operations exceeds the number of `Depart` operations). We present linearizable [4] implementations of SNZI objects, which are specified in Figure 1.

A SNZI object is easy to implement with a simple shared counter, which provides `Increment` and `Decrement` operations that return the value of the counter immediately before modifying it and a `Read` operation that returns the value of the counter without modifying it. It is straightforward to implement a shared counter by repeatedly using a common synchronization primitive such as compare-and-swap (`CAS`) to attempt to update the counter. While this approach is simple, nonblocking, linearizable, and reasonably fast in the absence of contention, it is not scalable. Severe performance degradation occurs under heavy use, as contention for the counter increases. Unfortunately, no known implementation of a shared counter is nonblocking, linearizable, scalable, and has low latency in the absence of contention—indeed, as discussed in Section 7, there are reasons to believe that such an implementation is impossible [5].

PODC'07, August 12–15, 2007, Portland, Oregon, USA.
ACM 978-1-59593-616-5/07/0008.

shared variable:
  Surplus: integer
    initially 0

bool `Query`()
  return (*Surplus* > 0)

void `Arrive`()
  Surplus ← *Surplus* + 1

void `Depart`()
  Surplus ← *Surplus* − 1

Well-formedness condition: the number of `Depart` operations invoked before any point in time is at most the number of `Arrive` operations completed before that time.

Figure 1: SNZI specification.

For many applications, the full functionality of shared counters is unnecessary; the weaker SNZI semantics is sufficient. For example, SNZI objects can be used to improve the scalability of reference-counting garbage collectors. Reference counting is a simple technique for determining when a resource can be reclaimed because it is no longer reachable. A garbage collector need not determine the exact number of references to a resource; it suffices to know whether there is *any* such reference. Thus, reference counters can be replaced by SNZI objects.

We can exploit the weaker semantics of SNZI to achieve implementations with better performance characteristics than a counter-based implementation. For example, when contention is high, the surplus may change much more frequently than the `Query` result, which changes only when the surplus changes from 0 to 1 and vice versa. Thus, a memory location read by `Query` may remain in cache even while many `Arrive` and `Depart` operations complete. This makes subsequent `Query` operations faster, and has other benefits that we explain later. Moreover, because SNZI objects need not determine the exact surplus—only whether it is zero or nonzero—we can avoid nonscalable centralized synchronization mechanisms such as simple `CAS`-based counters.

Our immediate motivation for considering SNZI objects is to improve the performance and scalability of hybrid transactional memory (HyTM) systems [1], in which transactions can be executed either directly by hardware or by using software. A key challenge in implementing HyTM systems is ensuring that hardware transactions detect conflicts with software transactions.

Suppose software transactions perform `Arrive` before beginning and `Depart` after completing. A hardware transaction that calls `Query` and gets false can infer that there are no software transactions in progress and can thus avoid the significant overhead of detecting conflicts with them for each transactional load or store. This is safe because, if a software transaction subsequently begins and completes its `Arrive` operation before the hardware transaction completes, the `Arrive` will cause a memory location previously read by the hardware transaction's `Query` operation to change, which will cause the hardware transaction to abort.

Furthermore, a good SNZI implementation can avoid modifying the memory location(s) read by the `Query` operation except when the surplus changes from 0 to 1 or from 1 to 0. Thus, if a hardware transaction's call to `Query` indicates that the surplus is nonzero (and thus that it must check for conflicts with software transactions on each load and store), then subsequent `Arrive` and `Depart` operations by software transactions need not always cause the hardware transaction to fail. In contrast, if we used a simple counter instead of a SNZI object, such operations would cause the counter to change, which would cause the hardware transaction to abort, often unnecessarily.

SNZI objects can also be used to improve "semivisible" read-sharing mechanisms [6], which allow a transaction that intends to write to a location to determine whether any transactions are reading the location. For this purpose, we do not need to know which transactions are reading nor how many there are, just whether the number of readers is nonzero. If software transactions perform `Arrive` before reading from the location and `Depart` when they end, a transaction that wants to modify the location can detect conflicts with readers by performing `Query`.

In addition to improving scalability under heavy read sharing by software transactions, using a SNZI object instead of a simple counter can again avoid unnecessarily aborting hardware transactions. In particular, a hardware transaction that wishes to read a location can query its associated SNZI object, and if it indicates that there is a nonzero number of software readers, it is safe for the hardware transaction to read share the location. This remains safe if another software transaction arrives, but this arrival would always cause the hardware transaction to fail if a simple counter were used to record the number of readers.

In this application of SNZI objects, if a location is being read shared by some transactions, another transaction can modify this location (thereby invalidating the reading transactions). After the location has been modified, we would like to be able to start read sharing the location again, without waiting for all of the previous readers to depart. To support this, we add a `Reset` operation and modify the `Query` operation to determine whether there is any reader that has arrived *since the previous reset* (if any) and not yet departed. The resulting object, SNZI-R (pronounced "snazzier"), is specified in Figure 2. `Arrive` and `Depart` operations pertain to a particular *epoch*, and the `Query` operation determines whether the number of `Arrive` operations exceeds the number of `Depart` operations *for the current epoch.* The `Reset` operation causes a transition to a new specified epoch, provided that this epoch is larger than the current epoch. Epochs are assumed to be totally ordered.

We describe novel, practical implementations for both SNZI and SNZI-R that exploit their weaker semantics. These

```
type SNZI_R_type =
  record
    Epoch: integer
    Surplus: integer

shared variable:
  S: SNZI_R_type
    initially ⟨0, 0⟩

⟨bool,integer⟩ Query()
  return
    ⟨S.Surplus > 0,
                S.Epoch⟩

integer Arrive()
  S.Surplus ← S.Surplus + 1
  return S.Epoch

void Depart(e: integer)
  if S.Epoch = e then
    S.Surplus ← S.Surplus − 1

bool Reset(e: integer)
  if e > S.Epoch then
    S = ⟨e, 0⟩
    return true
  else
    return false
```

Well-formedness condition: for any epoch e, the number of `Depart`(e) operations invoked before any point in time is at most the number of `Arrive` operations that completed before that time and returned e.

Figure 2: SNZI-R specification.

algorithms are nonblocking, linearizable, scalable, and have low latency in the absence of contention. We also present experimental results showing that they are substantially more scalable and efficient than the best practical alternative, namely simple `CAS`-based counters.

We have designed our implementations carefully to ensure that they are useful in our motivating applications. In particular, our `Query` operations consist of a single `Read` of a single bit (together with an epoch in the SNZI-R case). This ensures that the SNZI object can be queried very fast and that the information read by `Query` can be stored in a memory word already being used by an application, which may have only a small number of unused bits available.

The remainder of this paper is organized as follows: In Section 2, we briefly present some previous work relevant to the design of a SNZI implementation. Our SNZI implementation is presented in Section 3; an overview of its proof of correctness appears in Section 4. In Section 5, we explain how to augment our SNZI implementation to achieve a SNZI-R implementation. In Section 6, we present some preliminary performance results and explain why they demonstrate excellent performance characteristics for our motivating applications. We discuss some variations and future research directions in Section 7, and conclude in Section 8.

2. RELATED WORK

In this section, we briefly describe some related work that explains some of the intuition and motivation behind our algorithms.

Goodman et al. [3] proposed *combining trees* for implementing scalable counters. Operations start at a leaf and proceed towards the root, "combining" operations on the way to reduce contention on the counter at the root, which is crucial for scalability. The scalability of our algorithms is based on similar "filtering" mechanisms in which coordination at lower levels of a hierarchy reduces contention at higher levels. However, combining trees are blocking, need to know the maximum number of processes in advance, and have logarithmic *best-case* time complexity; our algorithms have none of these disadvantages.

Combining funnels, due to Shavit and Zemach [10], aim to address some of these problems by allowing processes to start anywhere in the tree, and adaptively changing the

depth and breadth of the part of the tree that is in use according to perceived load. Thus, when load is low, processes can go straight to the root of the tree, overcoming the logarithmic best-case time complexity of combining trees, as well as the need to configure the data structure based on the maximum number of processes. Our algorithm similarly allows processes to choose where to start in a hierarchy, thus achieving good throughput under high load and good latency under low load. However, like combining trees, combining funnels are blocking. On the other hand, our algorithms exploit the weaker semantics of SNZI objects to eliminate this problem: A process that relies on another to indicate its presence at the root of the hierarchy is not *required* to wait for that process, because for SNZI, it does not matter if the presence of both or only one is reflected at the root.

3. SNZI ALGORITHM

Our solution is organized as a rooted tree of SNZI objects, in which a child is implemented using its parent. That is, an operation on a child may invoke operations on its parent. We say that a parent's *surplus due to a child* is the difference between the numbers of `Arrive` and `Depart` operations invoked on the parent (henceforth *parent*.`Arrive` and *parent*.`Depart`) by operations of that child. Our implementation guarantees the following properties:

1. A parent's surplus due to a child is never negative.

2. A parent has a surplus due to a child if and only if the child has a surplus.

Given these properties, it is easy to see that the root of the tree has a surplus if any node in the tree does. Thus, considering the tree as a single SNZI object, processes can invoke `Arrive` and `Depart` on any node in the tree, and `Query` directly on the root (so the complexity of `Query` is independent of the depth of the tree).

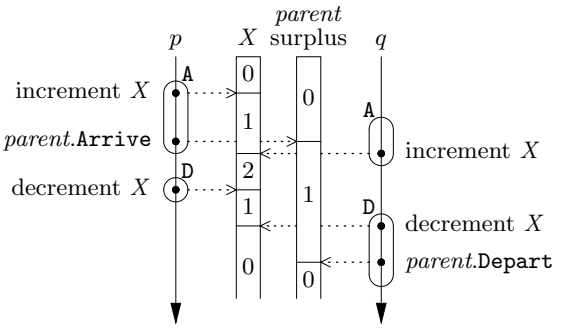
To reduce contention, we attempt to minimize the number of operations a child invokes on its parent. Thus, a child acts as a filter for its parent, and the tree structure greatly reduces contention for the root node, so we can use a nonscalable SNZI solution at the root without jeopardizing scalability overall.

Because some of our motivating applications require the indicator bit to be stored in the same word as application-specific information, at the root of the tree we use a special SNZI object that separates out the indicator bit.

Note that we do not rely on any special properties of the tree (other than rootedness); it need not have a fixed arity or depth, and processes can begin their `Arrive` operations at any node in the tree (as long as the corresponding `Depart` begins at the same node). This flexibility is useful because the optimal shape for the tree can depend heavily on details of both the application and the architecture.

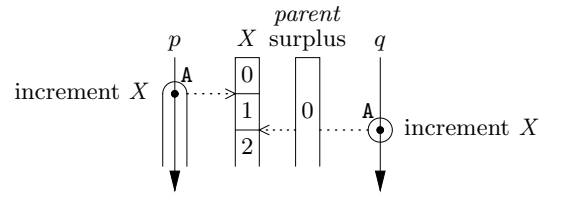
We first show how to implement a child node using its parent, and then present the SNZI implementation we use at the root. Our implementations use registers that support `Read` and `CAS`[1] operations.

[1] `CAS`(a, e, n) atomically compares the contents of address a to "expected value" e. If they are equal, it stores "new value" n to address a and returns true; otherwise it returns false and does not modify memory.



The two boxed columns in the middle indicate the values of the shared variables through time (time proceeds downward). Downward-pointing arrows represent processes, which invoke `Arrive` and `Depart` operations indicated by ovals labeled `A` and `D` respectively. Within the ovals, labeled points indicate access to shared variables, and the dotted arrows from these points indicate the value accessed.

Figure 3: Illustration of naïve algorithm



Process q increments X after process p, but before p invokes *parent*.`Arrive`. Because $X = 1$ when q increments it, q returns immediately without invoking *parent*.`Arrive`. At this point, the child has a surplus (because q has completed its `Arrive`) but the parent does not, violating property 2.

Figure 4: Problem with naïve algorithm

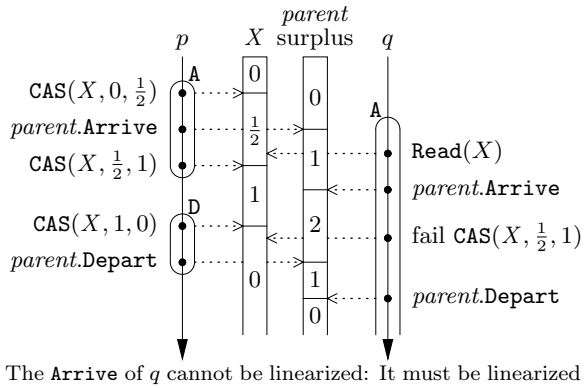
3.1 Hierarchical algorithm

To motivate our algorithm, we first consider a simple but flawed algorithm, illustrated in Figure 3, for implementing SNZI using a parent SNZI object. The child maintains a counter—call it X—which is incremented by `Arrive` and decremented by `Depart`. An `Arrive` that changes X from 0 to 1 invokes *parent*.`Arrive`; a `Depart` that changes X from 1 to 0 invokes *parent*.`Depart`. Other `Arrive` and `Depart` operations complete without invoking any operation on the parent. This algorithm violates property 2 above, as illustrated in Figure 4.

The basic idea in our algorithm is to introduce an intermediate value $\frac{1}{2}$ when incrementing X from 0 to 1. Any process seeing $X = \frac{1}{2}$ must first "help" the process that set X to $\frac{1}{2}$ by invoking *parent*.`Arrive` and then attempting to change X to 1 before retrying its own operation. Thus, before any of the `Arrive` operations on the child completes, at least one of them has completed a *parent*.`Arrive`.

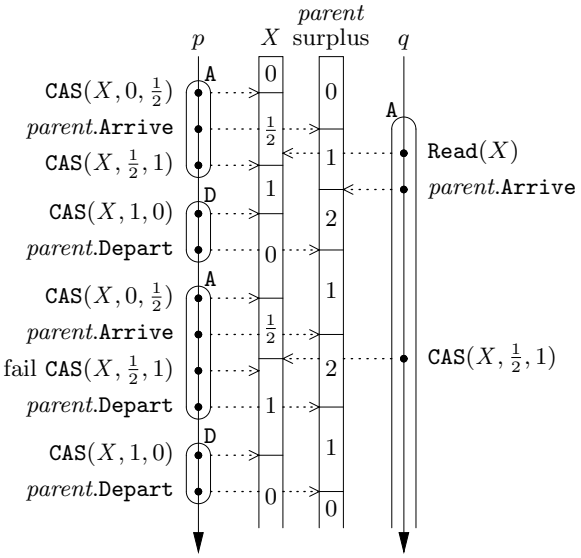
Because helping may cause *parent*.`Arrive` to be invoked several times for a single transition of X from 0 to 1, processes that invoke *parent*.`Arrive` but fail the subsequent attempt to change X from $\frac{1}{2}$ to 1 invoke a "compensating" *parent*.`Depart`. Thus, there is a compensating *parent*.`Depart` for all but one *parent*.`Arrive`. The remaining *parent*.`Arrive` is matched by a *parent*.`Depart` that is invoked by a process that changes X from 1 to 0 (in a `Depart` operation).

In the algorithm as described thus far, the surplus of the parent can "flicker" between 0 and 1 while an `Arrive` is in progress. There are two potential causes: a helping



The `Arrive` of q cannot be linearized: It must be linearized before p completes its `Depart` because at that point, the parent has a surplus, and thus by property 2, so does the child, which must be due to q's `Arrive`. On the other hand, it must not be linearized before q's *parent*.`Depart` because after that point, the parent has no surplus. (We elide `Read` operations that are immediately followed by successful `CAS` operations.)

Figure 5: Flicker due to eager compensation.



The `Arrive` of q cannot be linearized: It must be linearized before p completes its first `Depart` because at that point, the parent has a surplus, and thus, so does the child, which can only be due to q's `Arrive`. On the other hand, it must not be linearized before p completes its second `Depart` because after that point, the parent has no surplus.

Figure 6: Flicker due to late helpers.

parent.`Arrive` and its compensating *parent*.`Depart`, as illustrated in Figure 5; and a helping process that is so delayed that it changes X from $\frac{1}{2}$ to 1 for a later `Arrive` than the one that wrote the $\frac{1}{2}$ that it read, as illustrated in Figure 6.

Although this flicker is not a problem for the applications we have in mind, it violates property 2 above. Our algorithm, shown in Figure 7, avoids the first problem by deferring compensating *parent*.`Depart` operations until the end of an `Arrive` (using the *undoArr* variable), and avoids the second problem by adding a version number to X.

Because compensating *parent*.`Depart` operations are deferred, the parent's surplus may be unbounded even if the child's surplus never exceeds 1: a single `Arrive` can try

```
shared variables:
    X = (c, v): (ℕ ∪ {1/2}, ℕ); initially (0, 0)
    parent: scalable indicator

Arrive
    succ ← false
    undoArr ← 0
    while (¬succ)
1       x ← Read(X)
        if x.c ≥ 1 then
2           if CAS(X, x, (x.c + 1, x.v)) then
                succ ← true
        if x.c = 0 then
3           if CAS(X, x, (1/2, x.v + 1)) then
                succ ← true
                x ← (1/2, x.v + 1)
        if x.c = 1/2 then
4           parent.Arrive
5           if ¬CAS(X, x, (1, x.v)) then
                undoArr = undoArr + 1
6   while (undoArr > 0) do
        parent.Depart
        undoArr = undoArr − 1

Depart
    while (true) do
7       x ← Read(X)              /* assert X.c ≥ 1 */
8       if CAS(X, x, (x.c − 1, x.v)) then
            if x.c = 1 then
9               parent.Depart
            return
```

Figure 7: Code for hierarchical SNZI node.

and fail to help arbitrarily many other `Arrive` operations. Therefore, we have implemented an optimization (not shown in Figure 7) whereby an `Arrive` does at most two "extra" *parent*.`Arrive`s: If *undoArr* is 2 when a process would invoke *parent*.`Arrive`, the process instead decrements *undoArr*, eliminating one `Arrive`-`Depart` pair (thereby further reducing contention) on the parent. This is safe because the process has done at least one extra *parent*.`Arrive`, so the parent is guaranteed to have a surplus until the end of the `Arrive`. The experiments presented in Section 6 include this optimization, and in the full paper [2], we prove that the parent's surplus in the resulting algorithm is bounded by one plus twice the number of ongoing operations on the child.

3.2 Algorithm for the root node

We now describe the SNZI root node. As discussed above, the root node separates out the indicator bit I, which may need to be stored in the same memory location as application-specific data. To abstract away the interaction between accesses to this indicator bit by our algorithm and accesses to the same location by the host application, we present our solutions as if they access the indicator bit using only `Read`, load-linked (`LL`) and store-conditional (`SC`) operations. `Read` and `LL` return the value of the bit; an `SC` by process p sets the bit to a specified new value only if no process has performed a successful `SC` since the previous `LL` by process p. `SC` can fail "spuriously"; that is, the condition for success is *only if* not *if and only if*. To simplify our presentation, we also use a `Write` operation, which is easily implemented using `LL` and `SC` in a retry loop.

We require our solutions to tolerate spurious failures to abstract away interactions with the host application. For example, application-specific information stored together with

```
shared variables:
    X = (c, a, v): (ℕ, boolean, ℕ); initially (0, false, 0)
    I : boolean; initially false

Arrive
    repeat
10      x ← Read(X)
        if x.c = 0 then
            x' ← (1, true, x.v + 1)
        else
            x' ← (x.c + 1, x.a, x.v)
11  until CAS(X, x, x')
    if x'.a then
12      Write(I, true)
13      CAS(X, x', (x'.c, false, x'.v))

Depart
    repeat
14      x ← Read(X)              /* assert X.c ≥ 1 */
15      if CAS(X, x, (x.c − 1, false, x.v)) then
            if x.c ≥ 2 then return
            repeat
16              LL(I)
17              if Read(X).v ≠ x.v then return
18              if SC(I, false) then return

Query
    return Read(I)
```

Figure 8: Code for SNZI root node.

the indicator bit may change, causing SC to fail. Similarly, CAS-based implementations of LL/SC typically use a version number, and in some cases, this version number may be shared with the application, and so may change, even though no successful SC operation on the indicator bit is performed. In our algorithm, the only effect of a spurious failure is to cause another iteration of a small loop, degrading performance, but not affecting correctness.

As in the hierarchical algorithm, the root node maintains a counter that is incremented by Arrive and decremented by Depart. Transitions from 0 to 1 trigger setting I, those from 1 to 0 trigger clearing I. Processes that increment the counter after another process increments it from 0 to 1 and before that process sets I must "help" by setting I.

In this algorithm, shown in Figure 8, we implement helping using an "announce bit" a in the same word as the counter. A process that increments the counter from 0 to 1 also sets a. Other processes that increment the counter preserve a. Processes that set the announce bit, or preserve it as true, set I before attempting to clear the announce bit. Thus, I is set before any process completes Arrive.

A departing process that decrements the counter to 0 clears I using an LL/SC loop (we must use a loop only because of the possibility of spurious failures). However, it reads the counter between the LL and SC, and terminates without attempting the SC if the counter has changed; we add a version number to X to ensure that any change is detected. Thus, if the SC succeeds, there is no risk that some process has completed another Arrive, which would result in a Query operation incorrectly returning false.

4. PROOF OVERVIEW

In this section, we present an overview of the linearizability proof for our SNZI algorithms. A detailed proof appears in the full paper [2].

4.1 Hierarchical algorithm

The main proof obligation for the hierarchical algorithm is to show that the parent has a surplus (due to a child) exactly when the child has a surplus (i.e, property 2 above). We assume that the parent is linearizable, so we treat its operations as happening atomically.

The linearization point for a Depart is the last step before it returns, either when it decrements X to some nonzero value, or when it performs *parent*.Depart after setting X to 0. We cannot linearize when X is decremented in the latter case because a Query operation that occurs after X was set to 0 but before *parent*.Depart is performed will return true.

The linearization point for an Arrive is more complicated because of the possibility of helping. We partition Arrive operations by when they set the *succ* flag to true: either when changing X from 0 to $\frac{1}{2}$ or when incrementing X from some nonzero value at line 2. We call Arrive operations of the first kind *initiators*, and those of the second kind *joiners*. A joiner must be linearized by the time it successfully performs the CAS at line 2, and an initiator must be linearized by the time it calls *parent*.Arrive after setting its *succ* flag. However, in some cases, an Arrive must be linearized earlier because of the helping mechanism: If a process attempts to help an initiator by calling *parent*.Arrive but then fails its subsequent CAS at line 5 (because some other process already changed X), then we have to consider the helper to have already linearized because the initiator it was trying to help may have already departed. Therefore, a *parent*.Arrive by a process p may count either for an initiator it is helping or for p itself: the *parent*.Arrive counts for the initiator if p's subsequent attempt to change X from $\frac{1}{2}$ to 1 succeeds, and for p otherwise.

Unfortunately, when a process performs *parent*.Arrive, we may not yet know whether it will count for the initiator or itself. Fortunately, because Query indicates only whether there is a surplus, not which operations are responsible for the surplus, we can delay the linearization point until X is first changed after the initiator sets it. We call that point the *decision point* for the initiator. (If the helper calls *parent*.Arrive after this point—we call such a helper *late*—then we can linearize it immediately.) We must be sure, however, that the initiator or one of the helpers is linearized by the first time *parent*.Arrive is called on behalf of the initiator (i.e., when it is called because some process, possibly the initiator, saw that X had the $\frac{1}{2}$ value set by the initiator). For simplicity, we require that the initiator be linearized by that time.

Thus, the linearization point of an Arrive is the first point at which any of the following occur:

- it performs a successful CAS at line 2;
- it has performed a successful CAS at line 3 (i.e., it is an initiator) and then some process (possibly itself) performs *parent*.Arrive.
- it has performed *parent*.Arrive while $X = x$ and another process changes X from x; or
- it performs *parent*.Arrive late (i.e., when $X \neq x$).

We model this point by adding an auxiliary variable *done* to the Arrive operation. This variable is initially false, and it is set to true by the above events. An Arrive is linearized to the point that its *done* variable is set to true.

We now argue that the *done* variable of process p (henceforth $done_p$, and similarly for other local variables) is true if and only if:

- p is at line 4 or 5 and $succ_p$ = true (i.e., p's `Arrive` is an initiator) and either $X \neq x_p$ or some process q is at line 5 with $X = x_q$;
- p is at line 5 and $X \neq x_p$;
- $undoArr_p > 0$; or
- p is at line 6.

Note that the first condition is always satisfied if p is at line 5 with $succ_p$ = true. Also, because $X.v$ is incremented whenever a process sets $X.c$ to $\frac{1}{2}$, and $x_p.c = \frac{1}{2}$ whenever p is at line 4 or 5, if p is at line 4 or 5 and $x_p \neq X$ then $x_p \neq X$ until p reads X on line 1. With these observations, it is easy to check that once any of these conditions is true, one of them will be true until p completes its `Arrive`, that one of these conditions is true immediately after any of the above events, and that if none of these conditions is true then none of them becomes true unless one of the above events occurs.

To verify property 2, note that the surplus of *parent* is always equal to the number of processes at either line 5 or line 9 for an operation plus the *undoArr* of each process in the midst of an `Arrive` if $X.c$ is 0 or $\frac{1}{2}$, and 1 more than that if $X.c \geq 1$. We want to show that this is at least 1 iff the surplus of the child is at least 1.

The surplus of the child is the number of processes in the midst of an `Arrive` with *done* = true (i.e., linearized but not completed) plus the number of processes in the midst of a `Depart` (i.e., invoked but not linearized) plus the number of completed `Arrive` operations minus the number of invoked `Depart` operations. Let N be the number of completed `Arrive` operations minus the number of invoked `Depart` operations. Note that $\lceil X.c \rceil$ is incremented exactly once by each `Arrive` (i.e., when *succ* is set to true) and decremented once by each `Depart`, so it is equal to N plus the number of `Arrive` operations with *succ* = true plus the number of `Depart` operations at line 7 or 8. Furthermore, if $X.c = \frac{1}{2}$ then $N = 0$, no process is at line 6, 7 or 8, and there is an initiator is in progress whose decision point has not yet occurred.

Suppose the surplus of the child is at least 1. If $undoArr \geq 1$ for any process in the midst of an `Arrive` or any process is at line 5 or 9, then the surplus of the parent is at least 1. Also, if $N > 0$ or some process is at line 6, 7 or 8 then $X.c \geq 1$, so again the surplus of the parent is at least 1. Otherwise, there must be some process p with $done_p$ = true, $undoArr_p = 0$, and not at line 5 or 6. By our argument above, this process must be at line 4 with $succ_p$ = true and either $x_p \neq X$ or there is some process q at line 5 with $x_q = X$. However, the last case is impossible because no process is at line 5, so p is an initiator whose decision point has occurred, and thus $X.c \geq 1$ (so the parent surplus is at least 1).

On the other hand, suppose the surplus of the child is 0. Then $undoArr = 0$ for every process, and no process can be at line 9. Furthermore, if a process p is at line 5, then $x_p = X$ and $succ_p$ = false. In that case, since $X.c = x_p.c = \frac{1}{2}$, we know there must be some initiator q whose decision point has not occurred, and in that case $done_q$ = true, contradicting our assumption that the surplus of the child is 0. Thus, it remains to show that $X.c < 1$. Note that if it were at least 1, then, since there is no `Arrive` that has been linearized but is not completed, and no `Depart` that has been invoked but not is linearized, there must be some initiator p that is not yet linearized, and thus, whose decision point has not yet occurred. In that case, we know that $X.c = x_p.c = \frac{1}{2}$.

4.2 Algorithm for the root node

For the root node, we show that the indicator bit is set if and only if there is a surplus. With this property, we can linearize `Query` to its only step. We can verify this property by induction, checking that it is maintained by any steps that modify I or change the surplus (i.e., lines 12 and 18, and the linearization points of `Arrive` and `Depart` operations). We say that a process is *notable* if it is at line 18 and its `SC` may yet succeed.

The linearization point of an `Arrive` is its successful `CAS` at line 11 if I = true and no processes are notable at that point; otherwise it is the first subsequent point that makes that condition true.

The linearization point of a `Depart` is its successful `CAS` at line 15 if it changes $X.c$ to some nonzero value; otherwise it is the first subsequent successful `SC` at line 18 *by any process* or the last step of the `Depart`, whichever comes first.

A key lemma is that I is set and no processes are notable whenever the counter is nonzero and the announce bit is not set. We use this lemma to show that the linearization point for `Arrive` occurs before the operation completes.

With these linearization points, it is easy to see that I is true immediately after an `Arrive` is linearized, and that when I changes from false to true (making all processes not notable), the `Arrive` that performed this step is linearized, and thus there is a surplus immediately after. Also, when a `Depart` decrements the counter to a nonzero value (and is thus linearized at that point), immediately afterwards, there must be still a surplus, and, by the lemma above, I must true. Thus, it remains to consider executions of lines 17 and 18 that end a `Depart` operation: these include all the steps that clear I or are linearization points of `Depart` operations that decrement the counter to 0.

First note that, by well-formedness, there is a surplus whenever an ongoing `Depart` has not been linearized, and whenever the counter is 0, any `Arrive` that incremented the counter before that point is completed by that point.

Consider the interval from when a `Depart` decrements the counter to 0 until the `Depart`'s last step; call this the *active interval*, and let p be the process executing the `Depart`. Immediately before the active interval, there must be a surplus, and so, by the inductive hypothesis, I is true.

If the `Depart` ends because it performs a successful `SC` on line 18, then immediately after, I = false, so we must show that there is no surplus. Any `Arrive` that increments the counter in the active interval must do so after p last executes line 17. Since p is notable from that point until the end of the active interval, no `Arrive` is linearized during the active interval. By the definition of the linearization points, any `Depart` that decremented the counter before the end of the active interval is linearized at or before the end of the active interval. Thus, there is no surplus immediately after the active interval.

If the `Depart` ends at line 17 and this step is not the linearization point of the `Depart`, then it changes neither I nor the surplus, so the property holds by induction.

Otherwise, the `Depart` ends at line 17, which is its linearization point. By definition, no successful `SC` occurs in the active interval, and by the inductive hypothesis, $I = \text{true}$ throughout. We show that there is a surplus immediately after the active interval.

If the counter is 0 at the end of the active interval, then some other `Depart` must have written X last within the active interval. That `Depart` cannot have returned from line 17, so it cannot have been linearized. Thus, there is a surplus immediately after the active interval.

Otherwise, at the end of the active interval, there are more `Arrive` operations that have incremented the counter than `Depart` operations that have decremented it, which include all linearized `Depart` operations. If the last such `Arrive` has been linearized, then all `Arrive` operations that have incremented the counter have also been linearized. Thus, there is a surplus immediately after the active interval.

If the `Arrive` that last incremented the counter is not linearized by the end of the active interval, then its increment occurs during the active interval. From that point to the end of the active interval, there is some notable process; let q be a notable process at the end of the interval. If q's `Depart` is not yet linearized, then there is a surplus after the end of the interval. Otherwise, it is linearized at the first successful `SC` after it decremented the counter, which must be before p decremented the counter since no such `SC` occurs during the active interval. Thus, q decrements the counter before p, and some `Arrive` increments it in between (since p and q both decrement it to 0). Because the first `Arrive` to do so changes the version number, the last time q checks X before becoming notable must occur before this point, and thus, so too must q's last `LL`. However, the `Arrive` that does this increment must complete before the active interval (because the counter is 0 at the beginning), and it sets I before doing so, thus making q not notable, which is a contradiction.

5. SNZI-R

Our SNZI-R algorithm is similar to our SNZI algorithm, but it has an associated epoch; the primary changes for SNZI-R are to ensure that operations for previous epochs have no effect. Below we explain how our SNZI algorithm can be transformed to implement SNZI-R. Due to space constraints, we show the resulting code only for the root node in Figure 9; the changes to the code for the hierarchical algorithm are similar.

The internal SNZI-R objects have a slightly different interface than specified in Section 1. In particular, rather than returning the current epoch, the arrive operation takes an epoch as a parameter and does not return anything; it increments $S.Surplus$ if the specified epoch is current, and does nothing otherwise. This change supports our recursive implementation, and also enables some optimizations, as explained below. To distinguish this internal operation from `Arrive`, we call it `arr`.

The indicator word of the root node stores the current epoch and an indication of whether there is a surplus for that epoch; a successful `Reset` operation simply changes to the new epoch and sets the surplus indicator to false.

The SNZI-R nonroot nodes have an epoch stored together with their counters. If a nonroot node contains an epoch other than the current one, it is logically equivalent to containing the current epoch with the counter being 0. Therefore, steps of operations for an epoch e that encounter a node with an earlier epoch can simply update the node as if it contained epoch e and counter 0. If such a step is itself for an epoch before the current one, such a modification has no effect as the node still logically contains the current epoch and a counter value of 0 after the modification.

```
shared variables:
    X = (c, a, v, e): (ℕ, boolean, ℕ, ℕ); initially (0, false, 0, 0)
    I = (i, e): (boolean, ℕ); initially (false, 0)

arr(e)
    repeat
        x ← Read(X)
        if x.e > e then return
        if x.c = 0 ∨ x.e < e then
            x' ← (1, true, x.v + 1, e)
        else        /* assert x.c > 0 ∧ x.e = e */
            x' ← (x.c + 1, x.a, x.v, e)
    until CAS(X, x, x')
    if x'.a then
        repeat
            if LL(I).e > e then return
        until SC(I, (true, e))
        CAS(X, x', (x'.c, false, x'.v, e))

Depart(e)
    repeat
        x ← Read(X)
        if x.e ≠ e then return
        if CAS(X, x, (x.c − 1, false, x.v, e)) then
            if x.c ≥ 2 then return
            repeat
                if LL(I).e > e then return
                if Read(X).v ≠ x.v then return
                if SC(I, false) then return

Query
    return Read(I)

Reset(e)
    repeat
        if LL(I).e ≥ e then return false
    until SC(I, (false, e))
    return true
```

Figure 9: Code for SNZI-R root node.

It is easy to implement `Arrive` with a simple wrapper. An `Arrive` operation begins by invoking `Query` (on the root node) to determine the current epoch—call it e—and then invokes `arr`(e) on some node. The `Arrive` operation is deemed to have joined that epoch, whether the epoch remains current or not. If it does, then the `arr`operation behaves essentially as in the SNZI algorithm, except that it is modified to treat variables with earlier epochs in them as if they contained epoch e and a counter of 0.

If a `Reset` operation changes to a new epoch, all changes made by operations for previous epochs become irrelevant, because the variables containing previous epochs become logically equivalent to the new epoch (with counter value 0) as soon as the `Reset` takes effect. In this case, we can still linearize a concurrent `Arrive` to immediately before the `Reset`, because `Depart` operations for previous epochs have no effect according to the specification, and will have no effect on the shared object because any variables they modify have an out-of-date epoch both before and after the modification. The same observation enables various optimizations that allow an operation to return immediately when it determines that its epoch is no longer current.

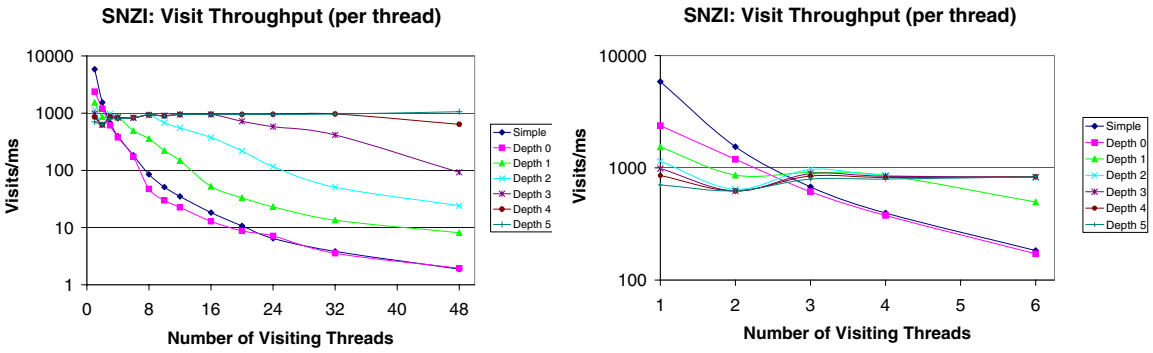


Figure 10: Experiments with SNZI and Simple.

6. PERFORMANCE EXPERIMENTS

We implemented our SNZI and SNZI-R algorithms, as well as simple counter-based implementations called Simple and Simple-R, in `C++`. We present results from experiments run on a 48-processor Sun Fire™ 6800 server, a cache-coherent multiprocessor with 24 dual-core 1350MHz UltraSPARC® IV chips, 97 GB of shared memory, and a 150 MHz system clock. Each processor chip has a 64 KB instruction cache, a 128 KB data cache on chip and a 16 MB level-2 cache off chip. We ran similar experiments on a 136-processor Sun Fire E15K, a cache-coherent multiprocessor with 68 dual-core 1.5GHz UltraSPARC® IV+ processor chips, and achieved qualitatively similar results. In particular, the simple counter-based algorithms continued to degrade with larger numbers of threads, while our SNZI algorithms continued to scale well up to the size of the machine, as they do on the 48-processor machine.

6.1 Overview of experimental design

We designed some simple experiments to evaluate our algorithms with respect to performance characteristics that we believe will be important for use in software and hybrid transactional memory systems, independently of workload, transactional memory design decisions, etc. In each experiment, a single SNZI object is accessed by one *querying thread*, which repeatedly invokes `Query`, and some number of *visiting threads*, which alternately invoke `Arrive` and `Depart`. We ran each experiment for a fixed amount of time, and recorded the throughput of the querying thread and the throughput of the visiting threads (averaged over all threads, and expressed in visits per millisecond, where a visit is an `Arrive` followed by a `Depart`). Note that query throughput provides an indication not only of how quickly the SNZI object can be queried, but also of how often the indicator word read by the `Query` operation is modified. The more often it is modified, the lower the query throughput will be because it will suffer more cache misses.

As explained earlier, our SNZI algorithms allow any tree structure, and allow `Arrive` operations to begin in any tree node. Thus they can be configured for different architectures, workloads, etc. and processes can dynamically adapt the level and node they begin at, according to observed contention or any other factor. For these experiments, however, we use binary trees of hierarchical SNZI nodes, with the special root node, as described in Section 3. Each visiting thread is assigned to a leaf of the tree, where it begins all `Arrive` and `Depart` operations. We assign the threads roughly evenly to the leaves, except that, when possible, we avoid assigning just one thread to a leaf, since operations of such a thread will always invoke the parent's operations. We experimented with tree depths from 0 to 5.

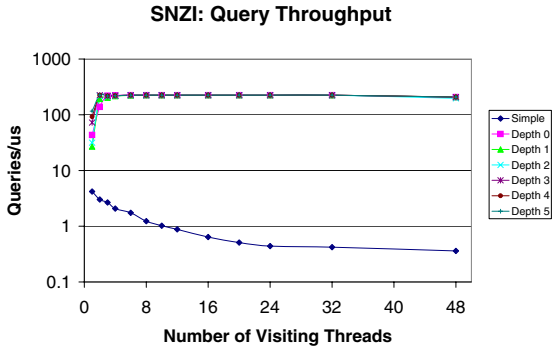


Figure 11: Experiments with SNZI and Simple.

6.2 Simple vs. SNZI

Figure 10 presents the visit throughput for Simple and SNZI with different tree depths (note that the y-axis of every graph is log scaled); visit throughput generally decreases as contention increases in all cases. However, deeper trees provide better scalability, as expected because they provide more levels of filtering, reducing contention at higher levels. For example, the ratio between the visit throughput going from 1 visiting thread to 48 is about 3100 with Simple, 1200 with a depth-0 SNZI tree, and only 1.3 with a depth-4 SNZI tree.

While deeper trees provide better visit scalability, the closeup graph on the right shows that deeper trees have lower visit throughput under low contention: The SNZI solutions are 2.5–8 times slower than Simple with 1 visiting thread, but up to 550 times faster with 48 visiting threads. For 3 or more visiting threads, there is always some SNZI solution that achieves higher visit throughput than Simple. In Section 6.3 we describe an improved SNZI solution that achieves the scalability of the basic SNZI algorithm, while matching the throughput of Simple with low contention.

Figure 11 shows the query throughput. For Simple, query throughput degrades rapidly as contention increases: its query throughput with 1 visiting thread is about 12 times higher than with 48 visiting threads. All of the SNZI solutions provide *improved* query throughput up to about 3

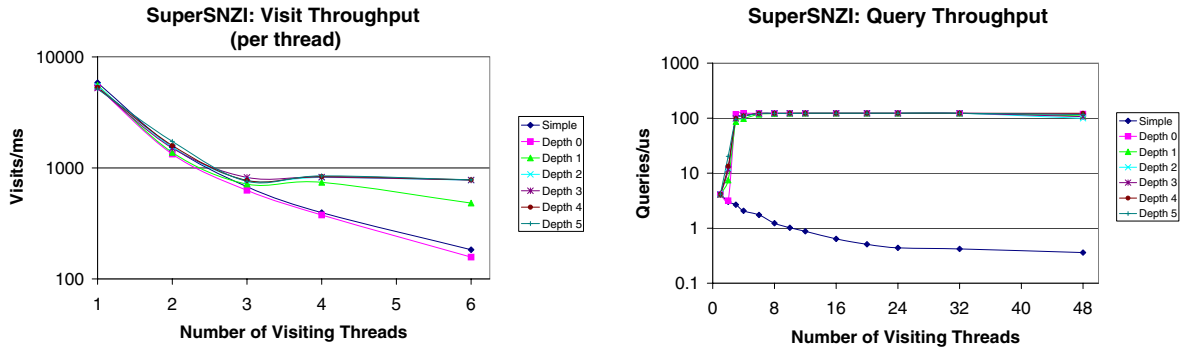


Figure 12: Experiments with Simple and SuperSNZI.

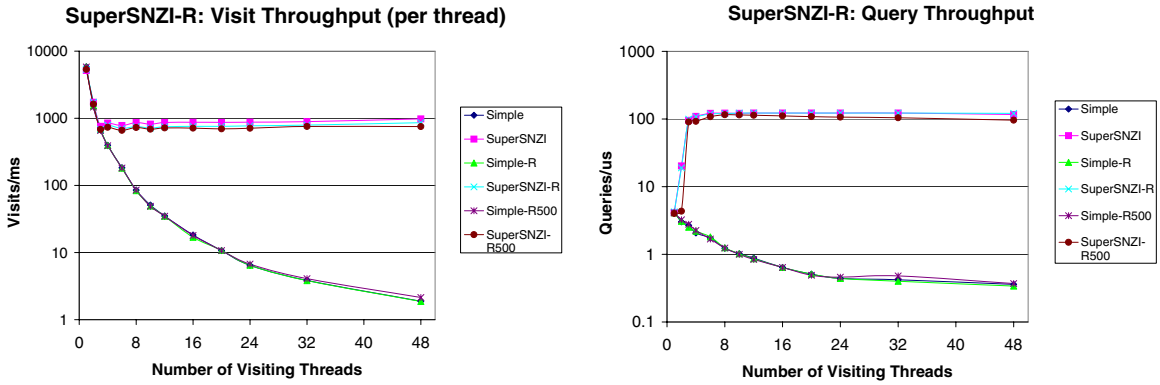


Figure 13: Experiments with resettable variants.

threads. The throughput level reached at 3 threads is within about 7% of the "optimal" throughput (measured by querying the SNZI in the absence of any visiting threads). With 48 visiting threads, all SNZI solutions have about 550 times higher query throughput than Simple.

Compared to Simple, the SNZI solutions achieve both higher visit throughput *and* higher query throughput, except when contention is very low. Finally, it is worth noticing that even though the depth-0 tree does not improve visit throughput (as all threads contend on the root node), it still significantly reduces the contention on the indicator word (as evidenced by higher query throughput); this is exactly what the root node is designed for.

6.3 SuperSNZI: Best of Both Worlds

As shown above, the SNZI solutions achieve better performance than Simple except under very low contention. We have implemented a variant of SNZI, which we call SuperSNZI, that incorporates a small counter into the indicator word. The `Query` method returns false if and only if both the SNZI bit and the small counter are 0. While contention is low, the algorithm uses this counter as Simple uses its counter. But as contention increases, `Arrive` operations begin to access the SNZI tree to ensure scalability.

There are many alternatives for deciding which method to use when arriving. Our implementation uses the SNZI algorithm if it fails to modify the counter too many times, if the counter is saturated, or if the SNZI indicator bit is already set. Thus, SuperSNZI starts using the simple counter, and switches to the SNZI algorithm if contention is high.

Figure 12 show that SuperSNZI achieves visit throughput similar to Simple with low contention *and* throughput similar to SNZI with higher contention—the best of both worlds. The price we pay is that modifications to the small counter at low contention result in cache invalidations that cause query throughput to suffer, but it recovers its advantage over Simple as soon as there is more than 1 thread. SuperSNZI is clearly superior when visit performance and throughput are most important.

SuperSNZI has the additional advantage that we can delay allocation of the SNZI tree until it is used for the first time, thus avoiding excessive space overhead for SNZI objects that are not heavily contended. Such an optimization would be important, for example, when using SNZI to improve the scalability of reference counts: we do not want to incur the space overhead of a SNZI tree for every object in the system, only for those whose reference counts are updated frequently.

6.4 Overhead of SNZI-R variants

We also evaluated resettable variants of Simple and SuperSNZI, measuring the overhead of supporting `Reset`, and the effect of periodic `Reset`s. For SuperSNZI, we used depth-5 trees, as this achieved the best results for high contention, and similar results for low contention.

The graphs in Figure 13 present 6 curves: Simple, Simple-R, SuperSNZI, SuperSNZI-R, Simple-R500 and SuperSNZI-R500. The first four are without any `Reset`s; for the last two, a separate thread called `Reset` every 500 microseconds. These results show that the overhead paid for the `Reset` feature is small for both Simple-R and SuperSNZI-R, and that there is little degradation in the face of occasional `Reset`s. We have not yet studied the effect of more frequent `Reset`s.

7. DISCUSSION

Intuitively, the semantics of traditional counters is hard to achieve with a scalable nonblocking implementation that is fast in the absence of contention: If two threads that increment the counter at around the same time, they must receive different and consecutive return values, and thus they must synchronize. An implementation that must be fast in the absence of contention must detect such contention quickly when it occurs. These observations suggest that an operation on the counter should quickly access a shared variable that will also be accessed by every other operation on the counter. Such an approach leads to centralized synchronization, which undermines scalability. Herlihy et al. [5] show a tradeoff for linearizable counters between latency and the worst-case number of processes that can simultaneously access one shared variable.

High worst-case contention for a variable does not imply poor scalability in practice. For example, in principle, all processes could access the indicator bit simultaneously in our SNZI implementations. However, in practice, they scale very well because this does not occur. We are interested in studying how the choice of semantics for counters, SNZI, and other related objects influences fundamental limitations with respect to measures that are important in practice.

In some cases, a weaker semantics for SNZI would suffice: `Query` could be permitted to return true, even if there is no surplus. When SNZI is used to detect the possibility of conflicts between hardware and software transactions, this may result in unnecessary overhead due to false conflict detection, but would not cause incorrect behavior. The weaker semantics might admit better SNZI implementations, but it wold provide less precise information, and would require a nontriviality condition to forbid solutions that *always* return true, thereby complicating the specification and raising the question of whether we had chosen the "right" condition.

If the word containing the indicator bit is updated only when the indicator bit changes value, then a hardware transaction will abort as a result of a call to `Query` only when the results of `Query` are no longer valid. Thus, HyTM can avoid unnecessary aborts of transactions executed in hardware. Our algorithm exhibits this behaviour in the common case, but does not guarantee it. Whether it is possible to make such guarantees without overly complicating the algorithm or introducing other disadvantages is unclear.

Stronger well-formedness conditions, for example, that processes alternate between `Arrive` and `Depart` operations, can be useful for some implementations. Similarly, it may be convenient to extend the interface, for example, to require each `Depart` operation to pass in an identifier returned by the corresponding `Arrive` operation.

Our algorithms allow rooted trees of arbitrary depth and arity, and allow processes to start at any node in the tree. This is advantageous because the optimal structure and layout of synchronization structures can depend heavily on application and architectural details. For example, work on scalable mutual exclusion locks for large multiprocessors has shown the benefit of having processes on one node of a large system first synchronize on local data, and electing one process to synchronize with processes on other nodes [9, 7, 8].

8. CONCLUDING REMARKS

We introduced SNZI, a scalable nonzero indicator, and its resettable variant SNZI-R. We presented practical, nonblocking, linearizable, scalable implementations that are fast in the absence of contention and can be instantiated for a variety of machine architectures. We explained how SNZI and SNZI-R objects can be used to improve the performance and scalability of software and hybrid transactional memory systems. Our performance experiments show that our algorithms have excellent performance characteristics for this purpose. We therefore plan to experiment with SNZI in a transactional memory implementation in the near future.

Acknowledgements: We thank Dave Dice for the observation that SNZI is applicable to reference counting, and Dan Nussbaum for useful conversations. We are also grateful to Russ Brown, Guy Delamarter, and Brian Whitney for their help with machines for our experiments.

9. REFERENCES

[1] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[2] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. Technical Report 2007-170, Sun Microsystems Laboratories, 2007.

[3] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[4] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[5] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[6] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory, 2004. http://research.sun.com/scalable/pubs/PODC04-Poster.pdf.

[7] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical CLH queue lock. In *Euro-Par*, 2006.

[8] Virendra Marathe, Mark Moir, and Nir Shavit. Composite abortable locks. In *Proc. International Parallel and Distributed Processing Symposium*, 2006.

[9] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *HPCA-9*, 2003.

[10] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *J. Parallel and Distributed Computing*, 60(11):1355–1387, 2000.