Theory Comput. Systems OF1–OF34 (2003)
DOI: 10.1007/s00224-003-1081-y

Theory of Computing Systems

Scalable Room Synchronizations*

Guy E. Blelloch,[1] Perry Cheng,[2] and Phillip B. Gibbons[3]

[1]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA
guyb@cs.cmu.edu

[2]IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
perryche@us.ibm.com

[3]Intel Research Pittsburgh, 417 South Craig Street, Pittsburgh, PA 15213, USA
phillip.b.gibbons@intel.com

Abstract. This paper presents a scalable solution to the group mutual exclusion problem, with applications to linearizable stacks and queues, and related problems. Our solution allows entry and exit from the mutually exclusive regions in $O(t_r + \tau)$ time, where t_r is the maximum time spent in a critical region by a user, and τ is the maximum time taken by any instruction, including a fetch-and-add instruction. This bound holds regardless of the number of users. We describe how stacks and queues can be implemented using two regions, one for pushing (enqueueing) and one for popping (dequeueing). These implementations are particularly simple, are linearizable, and support access in time proportional to a fetch-and-add operation. In addition, we present experimental results comparing room synchronizations with the Keane–Moir algorithm for group mutual exclusion.

1. Introduction

There has been a long history of developing data structures that support asynchronous parallel accesses—i.e., accesses for which neither the arrival times nor the number of

* This work was supported in part by the National Science Foundation under Grants CCR-9706572, CCR-0085982, and CCR-0122581. Much of the work was done while the second author was at Carnegie Mellon and while the third author was with Bell Laboratories, Murray Hill, New Jersey.

processors involved is known ahead of time. Unfortunately, it has been very difficult to develop truly efficient solutions for even some of the simplest asynchronous data structures, such as stacks and queues. Solutions based on locks are typically very simple, often relying directly on the sequential version. They are also linearizable [18], having the highly desirable property that the high-level data structure operations (such as push, pop, enqueue, dequeue) can be viewed as atomic. The problem is that solutions based on locks can fully sequentialize access to the data structures. Furthermore locks have the problem that if the process with the lock is blocked (e.g., swapped out by the operating system or dies), then all processes can become blocked.

To avoid problems with blocking, many nonblocking (or lock-free) algorithms for various data structures have been developed [2], [3], [13], [14], [23], [25], [32], [33], most of which are linearizable. As with the versions that use locks, however, these algorithms still sequentialize the access. For example, the algorithms for nonblocking queues [17], [23], [32] and stacks [30] sequentialize the inserts and/or deletes. Furthermore, many of these implementations have other problems such as requiring an atomic double compare-and-swap operation or requiring unbounded memory. We informally use the term *scalable* to refer to an algorithm that does not sequentialize access to the data structure it supports.

Gottlieb et al. [12] describe an algorithm for queues that implements enqueues and dequeues in time proportional to a fetch-and-add operation. The work was part of the Ultracomputer project [11] in which it was demonstrated that the fetch-and-add operation can be implemented as part of a multistage switching network so that it runs in about the same time as any access to shared memory (requests going to the same location can be combined in hardware in the network). The scalability of the fetch-and-add was also later justified theoretically by giving bounds on the run time on various networks [24], and several software implementations have been described [34], [10]. Gottlieb et al.'s queue algorithm is hence scalable assuming a scalable implementation of the fetch-and-add. Their algorithm, however, is not linearizable (see Section 5). It also seems unlikely that the technique can be extended to other data structures such as stacks. Shavit and Zemach describe a scalable and linearizable implementation of stacks based on combining funnels [29]. It is not clear, however, what the bounds on running time are, or how to extend the approach to queues. Neither the Gottlieb et al. nor the Shavit and Zemach algorithms are nonblocking.

We are interested in data structures that support asynchronous parallel accesses that are both scalable and linearizable, although not necessarily nonblocking. Furthermore we are interested in giving bounds on the time needed to access the data structure, at least under well-specified assumptions. We develop such algorithms using a scalable solution to the group mutual exclusion problem [19]. In the *group mutual exclusion problem*, multiple processes can simultaneously occupy a critical region of code, but no two processes can simultaneously occupy certain mutually exclusive critical regions. For example, our implementation of stacks allows multiple processes to occupy a push region or a pop region simultaneously, but requires that no process can be in a push region while another is in a pop region. Joung formalized the group mutual exclusion problem and gave an algorithm for supporting it [19]. Keane and Moir describe another algorithm that asymptotically improves performance under light load [20]. Both these algorithms, however, sequentialize the entry and/or exit to the exclusive regions. In Joung's algorithm the entry requires a loop over all processes, and in Keane and Moir's

algorithm the entry and exit are performed under a global lock. In our context this would sequentialize our data structures.

In this paper we describe an algorithm that supports group mutual exclusion with parallel (and scalable) entry and exiting of the critical regions. Our algorithm is based on implementing an `enterRoom(i)` operation that enters a critical region (room) `i`, and an `exitRoom()` that exits the current room. As with the work of Gottlieb et al. [12] we rely on a fetch-and-add primitive. A key property supported by our algorithm is that no user will wait more than $m(t_r + k\tau)$ time to enter or exit a room, where m is the number of rooms, t_r is the maximum time any user spends in a room, τ is the maximum time taken by any instruction, and k is a constant. We refrain from using $O(1)$ in place of $k\tau$ since the time τ for an instruction might be a function of the number of processors—e.g., a memory reference or fetch-and-add might require $O(\log p)$ time on p processors [24]. We informally use the term "constant time" to mean the time taken by a constant number of instructions.

We show how linearizable shared stacks and queues can be easily implemented using room synchronizations, all with constant time access regardless of the number of users. We also show how a dual concurrent write can be implemented. This operation along with a shared stack play an important role in a real-time garbage collector [4], and have been implemented using room synchronizations [6].

We present experimental performance results that compare our implementation of group mutual exclusion with the Keane–Moir algorithm [20]. The experiments were run on a 64 processor Sun UltraEnterprise 10000. Although this machine does not support the fetch-and-add operation in hardware, our implementation still outperforms the Keane–Moir algorithm under most situations. We expect that this is because the sequentialized region required to implement a fetch-and-add is significantly smaller than the sequentialized region required by the Keane–Moir algorithm. We also compare our implementation to a simple lock-based implementation under a variety of settings.

One disadvantage of using group mutual exclusion is that if a user fails or stops while inside a room, the user can block other users from entering another room. Approaches based on group mutual exclusion are therefore inherently not nonblocking. In our model this issue is captured by defining τ based on the gap in time between two consecutive instructions (actions) on a process. Therefore if a process is allowed to stall or fail, τ can be unbounded. We discuss this issue in Section 5.

We note that since the initial publication of our work [5], we discovered related work by Dimitrovsky presented as an Ultracomputer technical note [7]. Dimitrovsky suggests a similar technique for implementing stacks and queues. Instead of using multiple rooms, he uses a single "group lock." By splitting the group lock into two parts with a synchronization in the middle he is able to separate the pushes from the pops. The implementation of his group lock is quite different from our rooms, and requires many more fetch-and-adds to enter and exit. The work also does not formalize the techniques, show that it is linearizable, or consider dynamic stacks.

1.1. *A Motivating Example*

To motivate our problem, we consider implementing a parallel stack using a fetch-and-add. We assume the stack is stored in an array `A` and the index `top` points to the next free location in the stack (the stack grows from 0 up). The `fetchAdd(ptr,cnt)`

```
void push(val y) {
   int j = fetchAdd(&top,1);
   A[j] = y;
}

val pop() {
   val x;
   int k = fetchAdd(&top,-1);
   if (k <= 0) {
     top = 0;
     x = EMPTY;
   }
   else x = A[k-1];
   return x;
}
```

(a)

```
void push(val y) {
   enterRoom(PUSHROOM);
   int j = fetchAdd(&top,1);
   A[j] = y;
   exitRoom();
}

val pop() {
   val x;
   enterRoom(POPROOM);
   int k = fetchAdd(&top,-1);
   if (k <= 0) {
     top = 0;
     x = EMPTY;
   }
   else x = A[k-1];
   exitRoom();
   return x;
}
```

(b)

Fig. 1. The code for a parallel stack. (a) May not work if the push and pop are interleaved in time. (b) Avoids this problem using a room synchronization.

operation adds `cnt` to the contents of location `ptr` and returns the old contents of the location (before the addition). We assume this is executed atomically. Consider the stack code shown in Figure 1(a). Assuming a constant-time `fetchAdd`, the `push` and `pop` operations will take constant time. The problem is that they can work incorrectly if a `push` and `pop` are interleaved in time. For example, in the following interleaving of instructions

```
j = fetchAdd(&top,1);     // from push
k = fetchAdd(&top,-1);    // from pop
x = A[k-1];               // from pop
A[j] = y;                 // from push
```

the pop will return garbage. Without an atomic operation that changes the counter at the same time as modifying the stack element, we see no simple way of fixing this problem. One should note, however, that any interleaving of two or more pushes or two or more pops is safe. Consider pushes. The `fetchAdd` reserves a location to put the value `y`, and the write inserts the value. Since the counter is only increasing, it does not matter in what order relative to the increments the values are inserted.

Therefore, if we can separate the pushes from the pops, we would have a safe implementation of a stack. Room synchronizations allow us to do this as shown in Figure 1(b). The room synchronization guarantees that no two users will ever simultaneously be in the `PUSHROOM` and `POPROOM`, so the push and pop instructions will never be interleaved. However, it will allow any number of users to be in either the push or pop room simultaneously.

In Section 3 we prove that this stack is linearizable and that every access is serviced in time proportional to the time of a fetch-and-add along with a constant number of machine instructions. The experiments described in Section 4 are based on a variant of this stack in which multiple elements are pushed or popped within each room.

Outline. The paper is organized as follows. Section 2 presents our room synchronization algorithm and proves correctness and bounds on running time. Section 3 shows how room synchronizations can be effectively used to implement shared queues, dual concurrent writes, and dynamic shared stacks. Section 4 presents our experimental results. Section 5 discusses further issues and related work, and Section 6 concludes. The proofs in Section 2 assume that the counters used in the algorithm are unbounded; extensions to handle bounded counters appear in the Appendix.

A preliminary version of this paper appeared in the *Proceedings of the* 13th *ACM Symposium on Algorithms and Architectures*, Crete, Greece, July 2001, pp. 122–133 (see [5]).

2. Room Synchronizations

The group mutual exclusion problem involves a set of m sessions and a set of p independent processes who repeatedly request access to various sessions. It is required that distinct processes are not in different sessions concurrently, that multiple processes may be in the same session concurrently, and that each process that tries to enter a session is eventually able to do so [20]. Various authors have formalized these high-level requirements in slightly different ways (see [19] and [20]).

In this section we present a scalable algorithm for the group mutual exclusion problem. Although our algorithm meets the high-level requirements for group mutual exclusion, we propose a slightly different formalization (as detailed below), motivated by our target applications.

We first describe high-level primitives for supporting group mutual exclusion (Section 2.1). In Section 2.2 we present a scalable algorithm for supporting these primitives. In Section 2.3 we formalize our interface using the I/O Automata model. In Section 2.4 we prove the correctness of our algorithm. Finally, in Section 2.5 we present several variants of our algorithm and further discussion.

2.1. *Primitives*

We refer to the particular interface we use to implement group mutual exclusion as *room synchronizations*. In room synchronization a user wishing access to a room calls an Enter Room primitive, which returns once the user is granted permission to enter the room. When done with the room, the user exits the room by calling an Exit Room primitive. In further detail, the basic primitives of room synchronization are:

- **Create Rooms:** Given a positive integer m, create a rooms object R for a set of m rooms, and return a pointer (a reference) to R. There can be multiple rooms objects at the same time.

- **Enter Room:** Given a pointer to a rooms object R and a room number i, try to enter room i of R. Return when the user has succeeded in entering the room. When the primitive returns, the user is said to be *inside* the room. A room with a user inside is said to be *open*.
- **Exit Room:** Given a pointer to a rooms object R, exit the room in R that the user is currently inside. Because the user can be inside at most one room in R, there is no need to specify the room number. When a user requests to exit a room, it is no longer considered to be inside the room. If there are no users remaining inside the room, the room is said to be *closed*.
- **Destroy Rooms:** Given a pointer to a rooms object, deallocate the rooms object.

Other Primitives. It may be natural to define additional primitives for room synchronizations, as desired. We have considered two such primitives: Change Room and Assign Exit Code. The Change Room primitive is equivalent to an Exit Room followed by an Enter Room, but with the guarantee of entry into the requested room the next time the requested room is opened. The dynamic stack example described in Section 3 requires the Change Room primitive to prevent starvation and guarantee performance bounds. The Assign Exit Code primitive is discussed in Section 2.5.

Remarks. The Enter Room and Exit Room primitives can be viewed as the counterparts to the "trying" (e.g., lock) and "exit" (e.g., unlock) primitives for the mutual exclusion problem. As with the mutual exclusion problem, what the users do while inside the room (or critical section) is part of the application and *not* part of the synchronization construct. This enables the generality of the primitive, as the same construct can be used for a variety of applications. The drawback is that, as with mutual exclusion, the construct relies on the application to alternate entering and exiting requests by a given user.[1]

The Create Rooms and Destroy Rooms primitives are executed once for a given rooms object, in order to allocate and initialize the object prior to its use and to deallocate the object once it is no longer needed. To simplify the discussions that follow, we mainly focus on a single rooms object, for which Create Rooms has already been executed, and Destroy Rooms will be executed once the object is no longer needed. Extending the formalizations and discussions to multiple rooms objects and to issues of creating and destroying objects is relatively straightforward.

2.2. *A Scalable Room Synchronization Algorithm*

Figure 2 presents our room synchronization algorithm (protocol). Shown is the C code for the rooms data structure, and for the Create Rooms, Enter Room, and Exit Room primitives. The protocol is designed to achieve the following goals:

- only one room open at a time,

[1] The alternative is to combine the enter and exit primitives with the inside-the-room code into one monolithic construct. This has the disadvantage that either a distinct primitive would be needed for each inside-the-room code segment, or the code and data would need to be assigned to the primitive or passed as an argument.

```
    struct rooms {
      int *wait, *grant, *done;
      int numRooms, active, activeRoom;
    } Rooms_t;

    Rooms_t *createRooms(int m) {
      Rooms_t *r = (Rooms_t *) malloc(sizeof(Rooms_t));
      r->wait = (int *) calloc(m, sizeof(int));              // three counters per room, initialized to 0
      r->grant = (int *) calloc(m, sizeof(int));
      r->done = (int *) calloc(m, sizeof(int));
      r->numRooms = m;
      r->active = 0;                                         // 0 implies no active room
      return r;
    }

 1  void enterRoom(Rooms_t *r, int i) {
 2    int myTicket = fetchAdd(&r->wait[i],1) + 1;           // get ticket for the room
 3    while (myTicket - r->grant[i] > 0) {                  // wait until your ticket is granted
 4      if (testSet(&r->active)) {                          // while waiting, if no active room
 5        r->activeRoom = i;                                // then make i the active room
 6        int currWait = r->wait[i];
 7        r->grant[i] = currWait;                           // grant tickets to enter room i
 8        return;
 9      }
10    }
11  }

12  void exitRoom(Rooms_t *r) {
13    int ar = r->activeRoom;                               // preparing to exit room ar
14    int myDone = fetchAdd(&r->done[ar],1) + 1;            // increment the done counter
15    if (myDone == r->grant[ar]) {                         // if last to be done
16      for (int k = 0, newAr = ar; k < r->numRooms; k++) {
17        newAr = (newAr + 1) % r->numRooms;                // go round robin through the rooms
18        int currWait = r->wait[newAr];
19        if (currWait - r->grant[newAr] > 0) {             // if ticketed waiters
20          r->activeRoom = newAr;                          // then make newAr the active room
21          r->grant[newAr] = currWait;                     // and grant tickets to enter room newAr
22          return;
23        }
24      }
25      r->active = 0;                                      // no waiters found, so no active room
26    }
27  }
```

Fig. 2. Room synchronization code.

- any number of users can be inside the open room, and
- parallel entry and exit to rooms.

(These goals are formalized in Section 2.3.)

Consider a rooms object with m rooms. The procedure `createRooms` is used to allocate the rooms data structure for this object. The data structure includes three arrays of size m: `wait`, `grant`, and `done`. The arrays hold three counters for each room, all initially zero. It includes a `numRooms` field, set to m. It also includes an `activeRoom` field, which holds the room number of the (only) room that may be open, and an `active` field, which is used to indicate when there is *no* active room, e.g., initially and whenever there are no users either inside a room or waiting to enter a room.

Protocol Assumptions. The protocol assumes a linearizable shared memory [18] supporting atomic reads, writes, fetch-and-adds, and test-and-sets on single words of

memory. We have explicitly avoided atomic operations on two or more words of memory, and we use only the weaker fetch-and-increment form of fetch-and-add. A `fetchAdd(&x, 1)` instruction atomically (1) returns the current value of x and then (2) increments x. A `testSet(&x)` instruction atomically (1) returns 1 if x is currently 0, and 0 otherwise, and then (2) sets x to 1.

The rooms data structure is stored in the shared memory. All other variables in the code are local to the process executing the code. The code is written so that each line contains at most one shared memory access, so that each line corresponds to an atomic action with respect to the shared memory. This facilitates the correctness proofs that follow. In our actual implementation, the code is simplified by removing this restriction, e.g., Steps 6 and 7 are merged to simply `r->grant[i] = r->wait[i]`. Moreover, for multiprocessors that do not support a linearizable shared memory [1], *memory barriers* (or similar constructs) may need to be inserted into the code, either for correctness or to expedite the shared memory access. On the TSO memory model [1] provided on Sun multiprocessors, for example, memory barriers are not needed for correctness.

The protocol also assumes that the maximum number of concurrent users is less than the largest number that can be stored in an "`int`." It may be helpful for the reader to assume for now that the counters are unbounded, i.e., there is no concern about overflow of any fixed sized `int` in the code. We treat issues of bounded versus unbounded counters in the Appendix. In fact, we show in the Appendix that the protocol in Figure 2 is correct even if the `wait`, `grant`, and `done` counters "overflow" (wrap around within) a fixed sized `int`.

Entering and Exiting a Room. Users enter a room by incrementing the `wait` counter to get a "ticket" for the room (Step 2), and then waiting until that ticket is granted (Steps 3–10). Users exit a room by incrementing the `done` counter (Step 14). Once the `done` counter matches the `grant` counter (Step 15), then all users granted access to the room have exited the room. The unique user to increment the `done` counter up to the `grant` counter (the *last done*) does the work of selecting the next active room (Steps 16–24). This user cycles through the rooms, reading each room's `wait` counter into `currWait` and comparing this with the room's `grant` counter. The first room discovered whose `wait` exceeds its `grant`—indicating waiting tickets—is selected as the next active room (Step 20). The `grant` counter of that active room is set to be equal to `currWait` (Step 21), thereby granting all tickets for that room up to and including `currWait`. If, after cycling through all the rooms once, the last done user has failed to discover a room with waiting tickets, it resets `active` to `0` (Step 25). Whenever `active` is `0`, a ticketed user can succeed in the test-and-set of Step 4, set its requested room as the next active room (Step 5), and grant tickets for that room (Step 7).

Note that the technique of using a `wait` counter and a `grant` counter is used in mutual exclusion protocols such as TicketME [8], [21]. Mutual exclusion protocols are simpler because only one user is granted access at a time. Thus the `grant` counter can double as the `done` counter and the test for granting access is simply whether your ticket *equals* the `grant` counter.

2.3. *Formalization*

To prove correctness and other properties of our room synchronization algorithm, we first formalize room synchronization (our particular interface for group mutual exclusion)

using the well-studied I/O Automaton model [21]. Our terminology and formal model are an adaptation of those used in [21] for formalizing mutual exclusion.

Each user j is modeled as a state machine that communicates with an agent process p_j by invoking room synchronization primitives and receiving replies. The agent process p_j, also a state machine, works on behalf of user j to perform the steps of the synchronization protocol. Each agent process has some local private memory, and there is a global shared memory accessible by all agent processes. The set of agent processes, together with their memory, is called the *protocol automaton*. An *action* (an instruction step) is a transition in a state machine. We say an action is *enabled* when it is ready to execute. Actions are low-level atomic steps such as reading a shared memory location or incrementing a local counter. An *execution* is a sequence of alternating states and actions, beginning with a start state, such that each action is enabled in the state immediately preceding it in the sequence and updates that state to be the state immediately succeeding it. Thus actions are viewed as occurring in some linear order.[2]

Asynchrony is modeled by the fact that actions from different agent processes can be interleaved in an arbitrary manner; thus one agent may have many actions between actions by another agent. A weak form of fairness among the agent actions is the following: an execution is *weakly fair* if it is either (a) finite and no agent action is enabled in the final state, or (b) infinite and each agent has infinitely many *opportunities* to perform an action (either there is an action by the agent or no action is enabled) [21].

Certain actions are specially designated as *external* actions; these are the (only) actions in which a user communicates with its agent. For room synchronization, the external actions for a user j (and its agent) are:

- **EnterRoomReq**$_j(i)$**:** the action of user j signalling to its agent p_j a desire to enter room i.
- **EnterRoomGrant**$_j(i)$**:** the action of agent p_j signalling to user j that its Enter Room request has been granted.
- **ExitRoomReq**$_j$**:** the action of user j signalling to its agent p_j a desire to exit its current room.
- **ExitRoomGrant**$_j$**:** the action of agent p_j signalling to user j that its Exit Room request has been granted.

The *trace* (*trace at* j) of an execution is the subsequence of the execution consisting of its *external* actions (for a user j).

The terminology above focuses on modeling the agents that act on behalf of user requests, as needed to formalize both room synchronization and room synchronization algorithms. Section 3, on the other hand, focuses on modeling *user applications*, such as stacks and queues, that make use of room synchronizations. All of the terminology stated above for agents can be similarly defined in order to model users. For example, each *user* process has some local private memory, and there is a global shared memory accessible by all *user* processes. From the perspective of the present section, however, all users do is make requests to enter or exit rooms.

---

[2] Although an execution is modeled as a linearized sequence of low-level actions, this is a completely general model for specifying parallel algorithms and studying their correctness properties: the state changes resulting from actions occurring in parallel are equivalent to those resulting from *some* linear order (or *interleaving*) of these actions, given that the actions are defined to be sufficiently low level.

Properties. We first state formally a condition on users of room synchronization and their agents. A trace at j of an execution for a rooms object with m rooms is said to be *behaved* if it is a prefix of the cyclically ordered sequence:

$$\text{EnterRoomReq}_j(i_1), \text{EnterRoomGrant}_j(i_1), \text{ExitRoomReq}_j, \text{ExitRoomGrant}_j, \text{EnterRoomReq}_j(i_2), \ldots$$

where $i_1, i_2, \ldots \in [1..m]$. In other words, (i) the Enter Room and Exit Room primitives by a given user alternate, starting with an Enter Room, (ii) the user waits for a request to be granted prior to making another request, (iii) conversely, the agent waits for a request before granting a request and only grants what has been requested, and (iv) the requested room numbers are valid. We say a user j's requests are *behaved* if no request is the first misbehaved action in the trace at j (formally, there is no EnterRoomReq_j or ExitRoomReq_j in the trace at j such that the prefix of the trace up to but not including this action is behaved, but the prefix including the action is not behaved). In a behaved trace at j, $\text{EnterRoomReq}_j(i)$ transitions user j from *outside* all rooms to *preparing to enter* room i, $\text{EnterRoomGrant}_j(i)$ transitions user j from *preparing to enter* room i to *inside* room i, ExitRoomReq_j transitions user j from *inside* to *preparing to exit*, and ExitRoomGrant_j transitions user j from *preparing to exit* to *outside*. A room i is *open* if there is at least one user inside room i, and otherwise *closed*.

We can now state formally our target properties for room synchronization. A protocol automaton A *solves the room synchronization variant of the group mutual exclusion problem* for a given collection of users *with behaved requests* if the following properties hold:

P1. **Trace behaved:** In any execution, for any j, the trace at j is behaved. One implication is that only users requesting to enter a room are given access to the room, and only after its EnterRoomReq and before any subsequent ExitRoomReq.

P2. **Mutual exclusion among rooms:** There is no reachable state of A in which more than one room is open.[3] Equivalently, in any execution, between any $\text{EnterRoomGrant}_j(i)$ in the trace and the next ExitRoomReq_j (or the end of the trace if there is no such action) there are no $\text{EnterRoomGrant}(i')$ actions for $i' \neq i$.

P3. **Weakly concurrent access to rooms:** There are reachable states of A in which more than one user is inside a room.

P4. **Bounded waiting** (i.e., no user starvation): In any weakly fair execution: (1) If all users inside a room eventually prepare to exit the room, then any user preparing to enter a room eventually gets inside the room. (2) Any user preparing to exit a room eventually gets outside the room.

Keane and Moir [20] formalize group mutual exclusion using three properties. Two of our properties (P2 and P4) are equivalent to two of theirs. Our property P1 is explicit in our I/O Automaton formulation, and more implicit in [20]. Our property P3 is weaker than their third explicit property, which can be stated in our terminology as follows.

[3] A stronger property is to require that at most one room can be open even if some user requests are not behaved.

- **Concurrent entering:** In any weakly fair execution, any user preparing to enter a room i, such that no other user is either preparing to enter, inside, or preparing to exit a different room i', eventually gets inside the room (even if no other user prepares to exit a room).

This property allows a late arriving user always to join an open room if no user is waiting on another room. In contrast, we consider an additional property that expressly forbids this:

P5. **No late entry:** In any execution, a user outside room i at any point when the room is open will not be permitted inside room i as long as it remains open.

Both the dual concurrent write and the dynamic stack algorithms we describe in Section 3 require this property. The algorithms are not correct if concurrent entering is allowed.

Another property we target is:

P6. **Demand driven:** When a user is inside a room or outside all rooms, there are no actions by its agent. Thus, an agent performs work only in response to a request by its user.

This property ensures that the total work performed by agents does not depend on the total number of *potential* users.

Finally, it is often useful to target liveness conditions that are stronger than the fact that a desired event "eventually" happens. For this, we introduce upper bounds on the time for all the salient operations. Let τ be an upper bound on the (wall clock) time for an action by an agent with at least one enabled action.[4] Let t_r be an upper bound on the time a user is inside a room. We target the following timing property:

P7. **Constant time to enter and exit:** In any execution, any user preparing to enter a room is inside the room within time $T_1 \leq T_1(t_r, \tau, m)$, and any user preparing to exit a room is outside the room within time $T_2 \leq T_2(\tau, m)$.

Because the timing property is with respect to τ, it is closely tied to the set of atomic actions in the protocol, and may hide a dependence on p. For example, actions in our room synchronization algorithm include concurrent reads, concurrent test-and-sets, and concurrent fetch-and-adds. Thus, we also perform a more detailed analysis that accounts separately for potentially more expensive operations such as fetch-and-adds.

2.4. *Proofs of Correctness and Scalability*

In this section we show that our room synchronization algorithm satisfies properties P1–P7. Although the algorithm is not presented as an I/O automaton, we can view each step of the C code as an atomic action in the corresponding I/O automaton. This is done without loss of generality, because each step accesses at most one shared memory location. For simplicity, we also restrict our attention to the case where `wait`, `grant`, and `done` are unbounded counters (with no overflow), and hence they are monotoni-

[4] This can be formalized using the *timed* I/O Automaton model [21]. Note that in the absence of a positive lower bound on the time for an action, we have not restricted the relative speeds of the agents. Moreover, the time bound applies only to the analysis of time performance, and not to any correctness (safety) properties.

cally nondecreasing. (The generalization to the bounded counters case appears in the Appendix.) Readers not interested in the proof may proceed to Section 2.5.

Theorem 1. *The room synchronization protocol in Figure* 2 (*with unbounded counters*) *satisfies properties* P1–P7, *with enter wait time* $T_1 \leq (t_r + O(\tau)) \cdot m$ *and exit wait time* $T_2 = O(\tau \cdot m)$ *for property* P7.

Proof. To simplify the notation, we omit explicit reference to the rooms object pointer r, e.g., we use `enterRoom(`i`)` instead of `enterRoom(`r, i`)`.

We use the following definitions. For an execution σ, let $\sigma|j$ be the subsequence of σ consisting of its actions for a user j or its agent p_j. A user j has a *ticket* for a room i after an execution σ for each Step 2 of `enterRoom(`i`)` in $\sigma|j$ with no subsequent Step 14 of `exitRoom` in $\sigma|j$. A user j with a ticket for a room i is *blocked* after an execution σ if `myTicket` at j is greater than `grant[`i`]`. A user j is in the *advance room region* after an execution σ if some step among Steps 5–7, 16–21, or 25 is enabled, or Step 15 is enabled with a successful conditional test.

*Property P*1. In the protocol of Figure 2, an EnterRoomReq$_j(i)$ (ExitRoomReq$_j$) action corresponds to the user j initiating a procedure call to `enterRoom(`i`)` (`exitRoom`, respectively). An EnterRoomGrant$_j(i)$ (ExitRoomGrant$_j$, respectively) action corresponds to the completion and return of this procedure. Consider any execution and any user j with behaved requests. An EnterRoomGrant$_j(i)$ (ExitRoomGrant$_j$) action cannot be the first misbehaving action in the trace at j, because it can occur in the trace only immediately after the matching EnterRoomReq$_j(i)$ (ExitRoomReq$_j$, respectively) that initiated the procedure call. Thus the trace at j is behaved.

*Property P*2. To prove mutual exclusion, we begin with the following lemma.

Lemma 1. *Each user* (*with a behaved trace*) *has at most one ticket.*

Proof. Suppose there exists an execution σ such that a user j has multiple tickets after σ. By the definition of having a ticket, for each such ticket, there is a Step 2 in $\sigma|j$ with no subsequent Step 14 in $\sigma|j$. For each such Step 2, there is a preceding EnterRoomReq$_j$, but no subsequent ExitRoomGrant$_j$ because Step 14 must precede any ExitRoomGrant. Thus the trace at j is not behaved, and hence property P1 fails to hold, a contradiction. □

The heart of the mutual exclusion proof is the following lemma, which presents four invariants that also provide insight into the protocol.

Lemma 2. *In any execution* (*with behaved traces*):

1. *If a user j is inside room i, then j is an unblocked user with a ticket for room i.*
2. *For all rooms i,* `wait[`i`]` $\geq$ `grant[`i`]` $\geq$ `done[`i`]`, *and* `wait[`i`]` $-$ `done[`i`]` *is the number of users with tickets for room i. Moreover, for each t in* `grant[`i`]` $+$ 1, ..., `wait[`i`]`, *there is exactly one blocked user for room i with* `myTicket` $= t$, *and no other blocked users for room i.*

3. *At most one user is in the advance room region. If a user j is in the advance room region, then* `active` $= 1$ *and for all rooms* i, `grant[`i`]` $=$ `done[`i`]`. *If Step* 7 (*one of* 19–21) *is enabled at a user* j, *then* `grant[`i`]` $\leq$ `currWait` *at* $j \leq$ `wait[`i`]` (`grant[newAr]` $\leq$ `currWait` *at* $j \leq$ `wait[newAr]`, *respectively*).
4. *If there exists an unblocked user with a ticket for room* i, *then* `active` $= 1$ *and* `activeRoom` $= i$. *If Step* 6 (7, 14, 21) *is enabled at a user* j, *then* `activeRoom` $= i$ (i, `ar`, `newAr`, *respectively*) *at* j.

Proof. The proof is by induction on the number of actions in the execution. Initially, `wait[`i`]` $=$ `grant[`i`]` $=$ `done[`i`]` $= 0$, and all four invariants hold for the start state. Assume that all four invariants hold for all executions of $t \geq 0$ actions. Consider an arbitrary execution σ with t actions and consider all possible next actions α. Without loss of generality, assume that α is an action by user j or its agent. Let s_1 be the last state in σ and let s_2 be the updated state after α occurs.

We first show invariant 1 holds in s_2. If user j is inside room i, then because the trace at j is behaved, the last external action at j is EnterRoomGrant$_j(i)$. The last occurrence of Step 2 of `enterRoom(`i`)` in $\sigma|j$ precedes the EnterRoomGrant$_j(i)$ in σ, and there can be no subsequent Step 14 because there is no subsequent ExitRoomReq in $\sigma|j$. Thus j has a ticket for room i. Moreover, suppose j were blocked. Then `myTicket` at j is greater than `grant[`i`]` after σ. Let $\sigma = \sigma_1\alpha\sigma_2$ where α is the above Step 2, σ_1 is the prefix of σ prior to α, and σ_2 is the suffix of σ after α. By Lemma 1 and examination of the code, we see that there is no possible step in σ_2 by user j that modifies `myTicket` at j: its value is the same in all states in σ_2. Moreover, only Steps 7 and 21 set `grant[`i`]`, and hence it follows inductively by invariant 3 that `grant[`i`]` is nondecreasing. Thus user j's `myTicket` $>$ `grant[`i`]` in all states in σ_2. Thus in all such states, j is not enabled to exit the `enterRoom while` loop, and hence there would be no EnterRoomGrant$_j(i)$ in $\sigma_2|j$, a contradiction. Thus user j is not blocked, and invariant 1 is maintained.

To show invariant 2 holds in s_2, we must consider all the cases where α updates either `myTicket`, one of the counters, the number of ticketed users, or the number of blocked users, namely, Steps 2, 7, 14, and 21. Step 2 of `enterRoom(`i`)` increments `wait[`i`]`, and creates a blocked user with `myTicket` equal to the new value of `wait[`i`]`. By Lemma 1, this is the only ticket for user j. Thus invariant 2 is maintained. As for Step 7 of `enterRoom(`i`)`, inductively by invariant 3, `grant[`i`]` has not decreased due to this step and `grant[`i`]` $\leq$ `wait[`i`]` in s_2. It follows that the step maintains invariant 2. Likewise, Step 21 maintains the invariant for room `newAr`. If Step 14 is enabled in s_1, then user j has a ticket (by definition) for some room i. Moreover, by an argument similar to the one above for invariant 1, j is not blocked. Thus inductively by invariant 4, `activeRoom` $= i =$ `ar` in s_1. Step 14 increments `done[ar]`, and, by Lemma 1, it decrements the number of users with tickets for room `ar`, so the invariant is maintained. Hence, in all cases, invariant 2 holds in s_2.

To show invariant 3 holds in s_2, we again consider each relevant case for α, namely, Steps 2, 4–7, 14–21, and 25. If α is a Step 2, this only increases `wait[`i`]`, so the last part (and hence all) of the invariant is maintained at all users. If α is a Step 4 that succeeds in enabling Step 5 in s_2, then `active` $= 0$ in s_1 (otherwise, the `testSet` would return 0). Inductively by invariants 2 and 4, `grant[`i`]` = `done[`i`]` for all rooms i in s_1, and hence in s_2. Inductively by invariant 3, there are no users in the advance room region in

s_1. Moreover, α sets `active` $= 1$. Thus the invariant is maintained. If α is a Step 14, then, as argued above, Step 14 increments `done[ar]`, where j is an unblocked user with a ticket for a room `ar` in s_1. Inductively by invariant 4, for all rooms $i \neq$ `ar`, there are no unblocked users with a ticket for room i in s_1, and so, inductively by invariant 2, `grant[`i`]` $=$ `done[`i`]` in s_1 and hence in s_2, and `grant[ar]` $>$ `done[ar]` in s_1. Thus inductively by invariant 2, there is no user in the advance room region in s_1. In order for j to be in the advance room region in s_2, `myDone` at j must equal `grant[ar]` in s_2 (so that Step 15 is enabled with a successful conditional). This occurs only if `grant[ar]` = `done[ar]` in s_2, because `myDone` = `done[ar]` after α.

Next, note that if α is a Step 5–7, 15 (with a successful conditional), 16–21, or 25, then user j is in the advance room region in s_1. Inductively by invariant 3, j is the only such user in s_1, and hence no other user has a Step 7, 19, 20, or 21 enabled. Thus if α is a Step 7 or 21, its setting of `grant` does not violate the last part of the invariant. Moreover, there are no users in the advance room region in s_2, so the invariant is maintained. If α is a Step 25, there are no users in the advance room region in s_2, and the invariant is maintained. If α is a Step 6, then `currWait` at j equals `wait[`i`]` in s_2, and, inductively by invariant 2, `wait[`i`]` $\geq$ `grant[`i`]`, so the invariant is maintained. Similarly, the invariant is maintained if α is a Step 18. If α is a Step 5, 15–17, or 19–20, then inductively by invariant 3 and the fact that none of these steps add a user to the advance room region, set `active` $= 0$, update `grant` or update `done`, the invariant is maintained. Hence, in all cases, invariant 3 holds in s_2.

Finally, to show invariant 4 holds in s_2, we consider each relevant case for α, namely, Steps 2, 4–7, 13, 20, 21, and 25. If α is a Step 2, then by invariant 2 applied to both s_1 and s_2, the number of unblocked users with tickets for room i is unchanged. Moreover, `active` and `activeRoom` are unchanged, so the invariant is maintained. If α is a Step 4, it can only set `active` to 1, so the invariant is maintained inductively. If α is a Step 5, then user j is in the advance room region in s_1, and, hence, inductively by invariant 3, there is no other user in the advance room region in s_1. Thus inductively by invariants 2 and 3, there are no unblocked ticketed users in s_1, and hence in s_2. As argued above, Step 14 is enabled at some user j' only if j' is an unblocked ticketed user. Thus Step 14 is not enabled at any user in s_2. Moreover, a Step 6, 7, or 21 is enabled at some user j' only if j' is in the advance room region. Thus none of these steps are enabled in s_1, and hence in s_2, with the exception of Step 6 being enabled at user j in s_2. However, α sets `activeRoom` = i, as is required. Hence, the invariant is maintained. Likewise, Step 20 maintains the invariant for `activeRoom` = `newAr` in s_2. If α is a Step 6 or 7, then, inductively by invariant 4, `activeRoom` $= i$ in s_1 and hence in s_2. User j is in the advance room region in s_1, so, inductively by invariant 3, `active` = 1 in s_1 and hence in s_2. Step 7 can only unblock users with tickets for room i, so the invariant is maintained. The case for Step 21 is symmetric. If α is a Step 13, then `active` and `activeRoom` are the same in s_1 and s_2. For user j, α sets `ar` = `activeRoom` and enables Step 14. As argued above, j is an unblocked user with a ticket for a room `ar` in s_1; thus, inductively by invariant 4, `active` $= 1$ in s_1 and hence in s_2. Step 13 does not create a new unblocked ticketed user, so, inductively by invariant 4, the invariant is maintained. Finally, if α is a Step 25, then user j is in the advance room region in s_1, and hence, as argued above for Step 5, there are no unblocked ticketed users in s_2, and the invariant is maintained. Hence, in all cases, invariant 4 holds in s_2.

This concludes the proof of Lemma 2. □

To complete the proof of property P2, suppose there were an execution resulting in two distinct rooms i and i' with users j and j' inside the respective rooms. Then by invariant 1 of Lemma 2, j (j') is an unblocked user with a ticket for room i (i', respectively). Thus by invariant 4 of Lemma 2, `activeRoom` equals both i and i', a contradiction.

*Property P*5. Let $\sigma = \sigma_1\, \alpha_1\, \sigma_2\, \alpha_2\, \sigma_3$ be a trace behaved execution where α_2 is an ExitRoomReq action that closes a room i, α_1 is the (corresponding) last EnterRoomGrant(i) that opened room i, and σ_1, σ_2, and σ_3 are sequences of alternating states and actions. Let j be a user outside room i in some state s in σ_2. We must show that there is no EnterRoomGrant$_j(i)$ between s and the end of σ_2.

First note that throughout σ_2, there exists an unblocked user with a ticket for room i (invariant 1 of Lemma 2). Thus by invariants 2 and 4 of the same lemma, `grant[`i`]` < `wait[`i`]` and `active` = 1 throughout σ_2.

Because user j is outside room i in s, its last external action for i prior to s (if any) is an ExitRoomGrant$_j$ and hence, because the trace is behaved, its next external action for i after s (if any) is an EnterRoomReq$_j(i)$. If Step 2 of `enterRoom` is executed by agent j, then by the observations in the previous paragraph, `myTicket` at j is greater than `grant[`i`]`. Moreover, `active` $= 1$, and so j will be unable to exit the `while` loop. Thus, there is no EnterRoomGrant$_j(i)$ between s and the end of σ_2.

*Property P*7. Let m be the number of rooms, let p be the number of users, let t_r be an upper bound on the time a user is inside a room, and let τ be an upper bound on the time for an action by an agent with at least one enabled action. Property P7 holds due to the following observations. (1) A user desiring a ticket will get a ticket within a constant number of its agent's actions. (2) The *last done* when exiting a room i starts at room $i+1$ and cycles through all m rooms (including back to i), granting access to the first room it encounters with ticketed users (specifically, with users that grabbed tickets prior to last done's setting of its `currWait`—such users become unblocked). (3) Each unblocked ticketed user in the `enterRoom while` loop will get inside the room within a constant number of its agent's actions once it is unblocked. (4) A room with ticketed users gets its turn (i.e., is granted access) within m turns or less. Moreover, in the interim, Step 6 is executed at most once and Step 18 is executed fewer than $2m$ times. This holds even if the last done finds no rooms with ticketed users, because no turns were granted by the last done and the worst case for a user with a ticket for room i occurs when Step 5 selects room $i+1$ for the next turn, and each room gets a turn before finally room i gets a turn. (5) Each turn for a room takes at most $t_r + O(\tau)$ time.

Thus in any execution, any user preparing to enter a room is inside the room in time $T_1 \leq (t_r + O(\tau)) \cdot m$. Moreover, in any execution, any user preparing to exit a room is outside the room in time $T_2 = O(\tau \cdot m)$ for the *last done* and $O(\tau)$ for all others. (Note that the m in T_2 does not result in an m^2 in T_1, due to observation (4) above.)

In the above time analysis, we are implicitly assuming that τ is not a function of p, e.g., the time for a fetch-and-add is independent of p. If instead, we let $t_f = t_f(p)$ be an upper bound on the time for a fetch-and-add with p processors (e.g., $t_f = \log p$), then $T_1 \leq (t_r + 2 \cdot t_f + O(\tau)) \cdot m$ and $T_2 \leq t_f + O(\tau \cdot m)$.

Other Properties. The remaining properties (P3, P4, P6) are either obvious or are easy consequences of the above properties. Note that all the time bounds and properties

hold even in the presence of arbitrarily fast agents who do their best to starve other agents.

This concludes the proof of Theorem 1. □

2.5. *Discussion*

Note that `active` may be set to 0 even when there are users waiting with their tickets. However, such users must have grabbed their tickets after the last agent done checked to see if there were ticketed waiters. Moreover, although all agents granted access to a room are spinning waiting for that room to open, and hence will tend to proceed together into the room, in the worst case the agents may proceed to enter the room and exit the room at very different rates. Thus a room may open and close multiple times before all the granted agents are done with the room. Furthermore, an agent that is slow to grab a ticket may be bypassed by faster agents an unbounded number of times. This does not contradict property P7, because within $O(\tau)$ time the slow agent will grab a ticket, and from there will proceed in constant time (a function of t_r, τ, and m) to enter inside the room. On the other hand, if all agents run at roughly the same speed, an agent can be bypassed for its desired room at most once.

Our algorithm performs a test-and-set in Step 4 of `enterRoom`. A well-known performance optimization when using test-and-set is first to test whether the value is 0 and only perform the test-and-set if the value is 0. Thus the test-and-set is only performed when there is evidence that it might succeed.

A desirable property of our algorithm, when the preceding optimization is used, is that it is a *local spin* algorithm when executed on a cache-coherent multiprocessor. On such multiprocessors, when a shared variable is read by a processor, it is cached locally to that processor. As long as the value remains in the cache, any subsequent reads of the shared variable will hit in the cache and be serviced without any global communication. When a processor seeks to update the value of the shared variable, invalidation messages are sent to all the cache copies, and these copies are removed from the local caches. Thus a processor learns when the value has changed and only then does any global communication take place. In our algorithm, a user waiting to enter a room is spinning waiting for a change to `grant[`i`]` or `active`. Thus no global communication occurs until one of these values changes. The values of `grant[`i`]` and `active` can each change at most twice before the user can enter the room (recall that the user can be bypassed once). If the test-and-set is implemented such that invalidations are not sent if the value does not change, then *regardless of the relative timings* of user actions, each user performs only $O(1)$ global communications to enter and exit a room (except for the *last done*, who performs $O(m)$ global communications to exit the room). As an optimization, when the number of processes exceeds the number of processors, a spinning process can explicitly give up the processor (e.g., through a `thr_yield()` command), thereby minimizing the number of processor cycles wasted doing spinning.

To satisfy the stronger mutual exclusion among rooms property discussed in the footnote to property P2, it suffices to ignore all misbehaving requests, as follows. Associate with each room a vector V, one two-bit entry per user, indicating the effective status of the user, as outside (0), preparing to enter (1), inside (2), or preparing to exit

(3). At the beginning of `enterRoom` (prior to Step 2), determine the id of the user, and if $V[id] \neq 0$ or the requested room number is invalid, the user is misbehaving and a failure code is returned. Otherwise, set $V[id] = 1$ and permit the user's agent to proceed with Step 2. At the end of `enterRoom` (just prior to Step 11), set $V[id] = 2$. Similarly, at the beginning of `exitRoom` (prior to Step 13), determine the id of the user, and if $V[id] \neq 2$, the user is misbehaving and a failure code is returned. Otherwise, set $V[id] = 3$ and permit the user's agent to proceed with Step 13. Finally, set $V[id] = 0$ just prior to Steps 22 and 27.

We have used round-robin scheduling of rooms, although more sophisticated scheduling strategies might be implemented instead. For example, one could more frequently schedule commonly requested rooms. Note that the round robin performed by the *last done* implies that in the case of only a single user, the user's agent wastes time cycling through all the rooms each time it enters a room. Again, one could devise a more clever scheduling of rooms, while possibly sacrificing fairness.

We implemented a version of the Exit Room primitive that includes a special *exit code*. Exit code is assigned to a room using an Assign Exit Code primitive (`assignExitCode`) that takes a pointer to a function and a pointer to the arguments to the function. The exit code is executed by the last user to be done, prior to searching for the next active room (i.e., between Steps 15 and 16 of `exitRoom`). Thus we are guaranteed that the exit code is executed once, and only after all users granted access to the room are no longer inside the room, but before any users can gain subsequent access to any room. We have found the exit code to be quite useful in our applications of room synchronization (an example is given later in Figure 6 and is also used in our experiments of Section 4). Intuitively, the exit code can be viewed as enabling functionality such as "the last one to leave the room turns out the lights." The need for this functionality does not arise with mutual exclusion, because there is only a single user inside the critical section at a time.

Recall that the Change Room primitive (`changeRoom`) is equivalent to an Exit Room followed by an Enter Room, but with the guarantee of entry into the requested room the next time the requested room is opened. It can be implemented by acquiring a ticket for the next room (as in Step 2 of `enterRoom`) before exiting the current room.

3. Applications

In this section we describe four applications of room synchronizations: a shared stack, a shared queue, a dual concurrent write, and a dynamic shared stack. The shared stack is a more detailed description of the example covered in the Introduction, and the dual concurrent write is a problem that came up in a real-time garbage collector [4].

3.1. *Shared Stack*

Our implementation of a linearizable shared stack is given in Figure 3. The code is a more detailed version of the code given in the Introduction. The `newStack` routine creates a new stack object, including allocating an array of a fixed size `mysize` and calling

```
 1  int PUSHROOM = 0;
 2  int POPROOM = 1;

 3  int SUCCESS = 0;
 4  int OVERFLOW = 1;

 5  struct stack {
 6     val *A;
 7     unsigned size;
 8     int top;
 9     Rooms_t *r;
10  } Stack_t;

11  Stack_t *newStack(int mysize) {
12    Stack_t *s = (Stack_t *)
13        malloc(sizeof(Stack_t));
14    s->A = (val *) malloc(
15         mysize * sizeof(val));
16    s->size = mysize;
17    s->top = 0;
18    s->r = createRooms(2);
19    return s;
20  }

21  int push(val y, Stack_t *s) {
22    int status = SUCCESS;
23    enterRoom(s->r, PUSHROOM);
24    int j = fetchAdd(&s->top,1);
25    if (j >= s->size) {
26      s->top = s->size;
27      status = OVERFLOW;
28    }
29    else s->A[j] = y;
30    exitRoom(s->r);
31    return status;
32  }

33  val pop(Stack_t *s) {
34    val x;
35    enterRoom(s->r, POPROOM);
36    int j = fetchAdd(&s->top,-1);
37    if (j <= 0) {
38      s->top = 0;
39      x = EMPTY;
40    }
41    else x = s->A[j-1];
42    exitRoom(s->r);
43    return x;
44  }
```

Fig. 3. The code for a shared stack using room synchronizations.

`createRooms` to create two rooms associated with the stack (one for pushing and one for popping).

Theorem 2. *The algorithm of Figure* 3 *implements a linearizable stack, such that both the push and the pop operations take* $O(\tau)$ *time, where* τ *is an upper bound on the time for any instruction* (*action*).

Proof. We first show that the stack is linearizable. Consider a collection of p users executing push and pop operations on a stack according to the code in Figure 3, and p agents executing `enterRoom` and `exitRoom` operations in response to user requests. Let σ be an arbitrary execution involving these users and agents. We assume that no push or pop has been executed before the initial state of this execution (`top` = 0, and both rooms are closed). Consider the subsequence of the actions in σ comprised of the fetch-and-add actions generated by Step 24 of the push operation and Step 36 of the pop operation. We call these the *commit* actions. We argue that the ordering of these commit actions specifies a proper linearized order of the corresponding stack operations.

We call each interval of σ in which the PUSHROOM or POPROOM is open a *push* or *pop* interval, respectively. The push and pop intervals cannot overlap because of property P2 of rooms. We note that since all push code (lines 24–29) is within a push room, all actions for an individual push must occur within a single push interval.

Similarly for pops. Hence we only need show that the pushes and pops properly linearize within their respective intervals, and that the state when leaving an interval is consistent.

Consider the push operation. If a push interval has commit actions that cause an overflow, we call the first such action the *first overflow*. We partition each push interval into two subintervals, the part preceding a first overflow, called the *successful push interval*, and the part at or after, called the *overflow interval*. If there is no overflow, then the whole push interval is successful. Note that no action in a successful push interval can decrement `s->top`. Within a successful push interval, each committing fetch-and-add reserves a location in which to put the new item. Any commits from other pushes coming after but within the successful push interval will reserve higher, and hence later in "time," locations in the stack. Also, all writes to the reserved locations in Step 29 will complete within the push interval since they fall before the `exitRoom` instruction.

Now consider a first overflow action. This action increments `s->top` to `s->size + 1`. In the remaining overflow interval `s->top` will never go below `s->size`. This is because `s->top` is only incremented or reset to `s->size`. Hence all commits in the overflow interval will cause an overflow, which is what one would expect from a linear ordering of pushes. Furthermore when exiting the push interval, `s->top` will have value `s->size`, as desired, since each overflow finishes by setting it to this value. Together with the discussion in the previous paragraph, this means that all pushes within a push interval will have the proper linear order defined by the commit actions. Furthermore, the state when leaving the push interval will have all pushed values written, and the counter set properly. A similar argument can be made for the pops (where empty takes the place of overflow), and hence any interleaving of pushes and pops will be linearizable, and the linear order will be based on their commit actions.

We now show the time bounds. Note that the time that a user is in a room is bounded by the time for a fetch-and-add, and a constant number of other standard instructions (i.e., reads, writes, arithmetic operations, and conditional jumps). Each user is therefore in a room for at most $t_r = O(\tau)$ time. Based on Theorem 1, the maximum time a user will wait to enter or exit a room is $O(m(\tau + t_r))$. Since $m = 2$ the total time to enter, process, and exit is $O(\tau)$. □

3.2. *Shared FIFO Queue*

Our implementation of a linearizable shared FIFO queue is given in Figure 4. The queue object contains `top`, which points to the top of the queue (i.e., the next location to insert an element), and `bot`, which points to the bottom of the queue (i.e., the first element to remove). The implementation properly checks for overflow and underflow (emptiness). In the case of overflow during an `enqueue`, the element is not inserted and the `top` pointer is not incremented (the implementation first increments it, but then resets it). Similarly in the case of an empty queue, the `bot` pointer is not incremented. Assuming the `int` type is of fixed precision then `bot` and `top` can both overflow. In our code and proof we assume there is no overflow, but it is not hard to fix the code to handle overflow without affecting the correctness or time bounds. This can either be done by taking advantage of wraparound when an integer overflows, or by using exit code (see the end of the previous section) to reset the counter when it is close to overflowing. In

```
 1  int ENQUEUEROOM = 0;
 2  int DEQUEUEROOM = 1;

 3  int SUCCESS = 0;
 4  int OVERFLOW = 1;

 5  struct queue {
 6     val *A;
 7     unsigned size;
 8     unsigned int top, bot;
 9     Rooms_t *r;
10  } Queue_t;

11  Queue_t *newQueue(int mysize) {
12    Queue_t *q = (Queue_t *)
13        malloc(sizeof(Queue_t));
14    q->A = (val *) malloc(
15         mysize * sizeof(val));
16    q->size = mysize;
17    q->top = q->bot = 0;
18    q->r = createRooms(2);
19    return q;
20  }

21  int enqueue(val y, Queue_t *q) {
22    int status = SUCCESS;
23    enterRoom(q->r, ENQUEUEROOM);
24    int j = fetchAdd(&q->top,1);
25    if (j >= q->bot + q->size) {
26      q->top = q->bot + q->size;
27      status = OVERFLOW;
28    }
29    else q->A[j % q->size] = y;
30    exitRoom(q->r);
31    return status;
32  }

33  val dequeue(Queue_t *q) {
34    val x;
35    enterRoom(q->r, DEQUEUEROOM);
36    int j = fetchAdd(&q->bot,1);
37    if (j >= q->top) {
38      q->bot = q->top;
39      x = EMPTY;
40    }
41    else x = q->A[j % q->size];
42    exitRoom(q->r);
43    return x;
44  }
```

Fig. 4. The code for a shared queue using room synchronizations.

the case of wraparound the range of `int` has to be greater than `2 * (q->size + p)` and a multiple of `q->size`.

Theorem 3. *The algorithm of Figure* 4 *implements a linearizable FIFO queue, such that both the enqueue and the dequeue operations take take $O(\tau)$ time, where τ is an upper bound on the time for any instruction* (*action*).

The proof is almost identical to the proof of Theorem 2.

3.3. *Dual Concurrent Writes*

Our next application is a dual concurrent write. The dual concurrent write operation takes a single value and writes it to two arbitrary locations, called a pair. We assume writes to the same pair from different users can overlap in time. However, concurrent writes can only involve the same location if they appear in the same ordered pair—i.e., (l_1, l_2) and (l_1, l_3) cannot be written concurrently. We are interested in linearizable solutions that take constant time. Linearizability implies that both locations of a pair that is written end up with the same value. This operation has an application in our real-time garbage collector. This application is discussed below. Because of the intended application in the garbage collector, we are also interested in solutions that do not require a synchronization location for each pair.

```
1  int COPYROOM = 0;

2  Rooms_t *r = createRooms(1);

3  void dualWrite(val y, val *l1, val *l2) {
4    *l1 = y;
5    enterRoom(r, COPYROOM);
6      val tmp = *l1;
7      *l2 = tmp;
8    exitRoom(r);
9  }
```

Fig. 5. The code for a dual concurrent write using room synchronizations.

Our solution is shown in Figure 5. An interesting aspect of this implementation is that it only involves a single room, and requires property P5 of rooms. We note that the implementation requires no additional memory beyond the room structure, which is constant size, and independent of the number of pairs.

Theorem 4. *The algorithm of Figure* 5 *implements a linearizable dual concurrent write that takes* $O(\tau)$ *time, where* τ *is an upper bound on the time for any instruction* (*action*).

Proof. We first show that the dual concurrent write is linearizable. As before, let σ be an arbitrary execution involving p users and agents. Consider the subsequence of the actions in σ comprised of the write actions generated by Step 4. We call these the *commit* actions. We argue that the ordering of these commit actions specifies a proper linearized order of the corresponding write operations.

For each dual write to a pair of locations (l_1, l_2), we call the interval in σ between the commit action and the ExitRoomReq action an (l_1, l_2) *write interval*. We call each maximal interval that is covered by (l_1, l_2) write intervals, an (l_1, l_2) *active interval*. We call the dual write associated with the last commit to l_1 within an (l_1, l_2) active interval, the *last write*, and the associated value, the *last value*. We call each maximal interval in which the COPYROOM is open a *copy interval*.

We argue that the state immediately following any (l_1, l_2) active interval will contain its last value in both l_1 and l_2. Because of property P5 of rooms, the copy interval used by the last write must start after it commits its value to l_1 (in σ). Consider all writes to (l_1, l_2) that join the copy interval used by the last write, or any later copy intervals within the active interval. Since the copies must all start after the last value is written to l_1, they all copy this same value into l_2 (the interleaving of their reads and writes does not matter). Since there is at least one such copy (the one associated with the last write), the last value will be properly copied into l_2 in the state immediately following the (l_1, l_2) active interval. Having the last value written into both l_1 and l_2 is consistent with a linearized order of the writes.

With regard to time, the time spent in the room only involves a read and a write, so by Theorem 1, a user will only have to wait $O(\tau)$ time, and the total time for the write is also $O(\tau)$. □

Our parallel real-time garbage collector uses a version of this dual concurrent write. To achieve real-time bounds, the collector copies the memory graph while the program is still running. This means that every location could have two copies, called the primary (the one accessed by the program) and the replica (the copy that is being made). When the program writes to a variable it needs to be sure that both copies of the variable are updated consistently. The protocol is somewhat more complicated than the dual concurrent write in that in addition to needing to support concurrent writes, there is a process that is copying elements from the primary to the replica. With the implementation we gave for dual concurrent writes, however, this is trivial to implement—the copy routines simply executes lines 5–8. In practice it is extremely important that every memory location does not require a separate synchronization variable—this could double the memory size.

In our actual implementation, we execute the initial write to the primary copy ($l1$) immediately on a write but at that point only store away the fact that this location needs to be copied to the replica ($l2$). This allows us to make no changes to how a write is compiled—keeping track of the writes is required anyway to maintain cross pointers between *generations* properly. The copying of values to the replica is then batched in groups, meaning that multiple copies are done while a user is in the COPYROOM. This amortizes the cost of entering and exiting.

3.4. *Shared Dynamic Stack*

We now consider implementing a linearizable shared dynamic stack. In a dynamic stack we assume the size of the stack is not known ahead of time and hence the space allocated for the stack must be capable of growing dynamically. In practice such dynamic stacks are quite important. If an application uses a collection of stacks that share the same pool of memory, it is crucial to minimize the space needed by each stack (i.e., allocating the maximum that each might possibly need is impractical). Our implementation is given in Figure 6. It makes use of the `assignExitCode` and `changeRoom` functions, and requires property P5 of rooms. We assume that `INITSIZE` is greater than the maximum number of concurrent users and is an even number. The `pushRoomExit` routine is assigned as the exit code to the `PUSHROOM` and runs whenever the `PUSHROOM` is exited, including when exiting through the `changeRoom` function.

Each time the stack grows, the allocated space is doubled and the old stack `s->B` is copied to the new larger one `s->A`. This copying is executed incrementally—each push on the top half of `s->A` copies one element from `s->B` to the bottom half of `s->A`. Once the top half of `s->A` is full, `s->B` is fully copied into the bottom half of `s->A`. There are two active arrays, `s->A` and `s->B`, at all times—when a new `s->A` is allocated the old `s->B` is freed and the old `s->A` becomes the new `s->B` (`s->B` is always half as large as `s->A`). The `pushRoomExit` code is responsible for checking if the stack has overflowed (indicated by the variable `s->start`) and allocating a new `s->A` if it has.

The `push` code works as follows. It tries to reserve a slot using the first `fetchAdd`. If this causes an overflow, a flag `s->start` is set to indicate to the `pushRoomExit` code that the size of the stack needs to be doubled. As with the static stack, `s->top` is then reset to `s->size` to undo the effect of the failed reservation. The `changeRoom` forces the user to wait until all users exit the room, at which point the `pushRoomExit` code is executed by a single user before the room is re-entered. If the reservation failed

```
int PUSHROOM = 0;
int POPROOM = 1;

struct stack {
   val *A;    /* current stack */
   val *B;    /* prev stack */
   int start;
   long size, top;
   Rooms_t *r;
} Stack_t;

void pushRoomExit(void *arg) {
   Stack_t *s = (Stack_t *) arg;
   if (s->start) {
     s->start = 0;
     s->size = 2*s->size;
     free(s->B);
     s->B = s->A;
     s->A = (val *) malloc(
        s->size * sizeof(val));
   }
}

Stack_t *newStack() {
   Stack_t *s = (Stack_t *)
           malloc(sizeof(Stack_t));
   s->A = (val *) malloc(
           INITSIZE * sizeof(val));
   s->B = (val *) malloc(
           INITSIZE/2 * sizeof(val));
   s->top = s->start = 0;
   s->size = INITSIZE;
   s->r = createRooms(2);
   assignExitCode(s->r, PUSHROOM,
                  pushRoomExit, s);
   return s;
}
```

```
void push(val y, Stack_t *s) {
   enterRoom(s->r, PUSHROOM);
start:
   int j = fetchAdd(&s->top,1);
   if (j >= s->size) {
      s->start = 1;
      s->top = s->size;
      changeRoom(s->r, PUSHROOM);
      goto start;
   }
   if (j >= s->size/2) {
      s->A[j - s->size/2] =
         s->B[j - s->size/2];
      s->A[j] = y;
   }
   else s->B[j] = y;
   exitRoom(s->r);
}

val pop(Stack_t *s) {
   val x;
   enterRoom(s->r, POPROOM);
   int j = fetchAdd(&s->top,-1);
   if (j < 0) {
     s->top = 0;
     x = EMPTY;
   }
   else if (j < s->size/2)
      x = s->B[j-1];
   else x = s->A[j-1];
   exitRoom(s->r);
   return x;
}
```

Fig. 6. The code for a shared dynamic stack.

on the first try, it will succeed on the second try since we assume the maximum number of users is bounded by the increase in size of the stack, which is at least `INITSIZE`. Note that the use of a `changeRoom` is critical for this to be the case since it limits to one the number of reservations any other user can make in the interim. Using an `exitRoom` followed by an `enterRoom` could allow another user to enter and exit many times before the attempt by the failed reservation is retried. When the reservation succeeds, and if it is in the top half of the stack, the push writes the data to the new array (`s->A`) and also copies an element from the old array (`s->B`) to the new one. If the reservation succeeds but it is in the bottom half, the push only writes the value to the old array. A later push in the top half will copy it to the new array.

The `pop` code is similar to the static stack except that it looks for the value in the old array (`s->B`) if the index is in the bottom half and the new array (`s->A`) if it is in the top half.

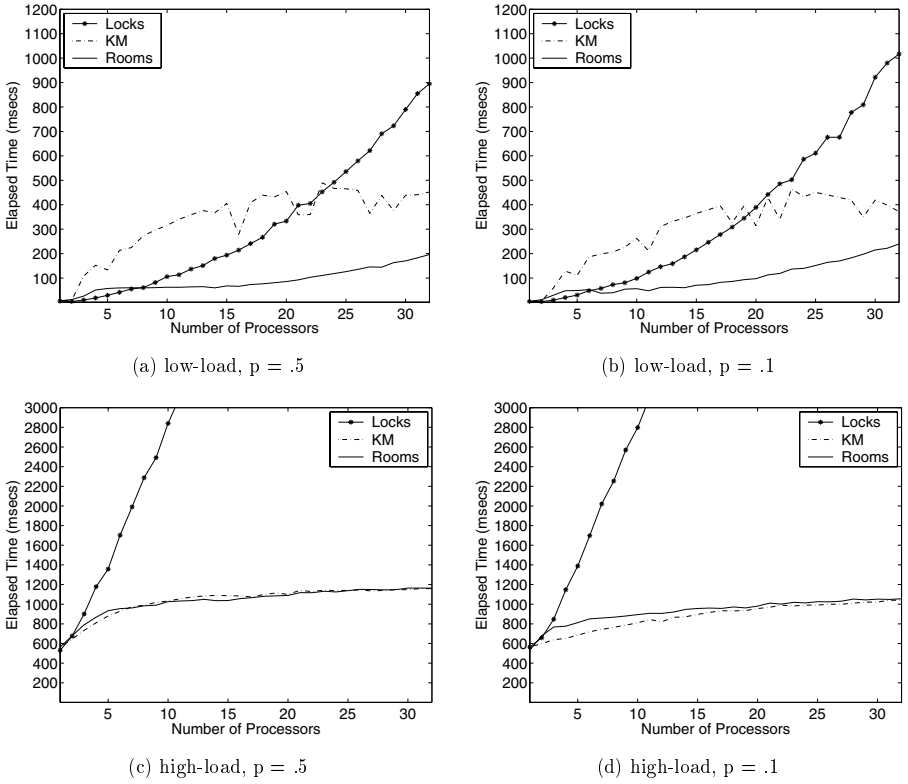
We claim that the algorithm of Figure 6 properly implements a linearizable stack of dynamic size, and that each push and pop will take time at most proportional to the cost of a `malloc` or fetch-and-add. Since we did not formalized the `assignExitCode` or `changeRoom` functions, we do not state a formal theorem.

4. Experimental Results

In this section we describe the results of two sets of tests. The first set compares the performance of group mutual exclusion using our algorithm and Keane and Moir's algorithm [20], henceforth the KM algorithm. The second set gives timings for an implementation of a shared work stack. The experiments were performed on a Sun UltraEnterprise 10000 with 64 250 MHz UltraSparc-II processors. We only ran our experiments on up to 32 processors because that was all we were able to access consistently. The UltraEnterprise is a shared-memory machine with a compare-and-swap instruction, but no fetch-and-add. The fetch-and-add is therefore simulated using the compare-and-swap. Using a compare-and-swap is not scalable since the fetch-and-add itself is sequentialized, but as the experiments show, in most cases we still outperform the KM algorithm.

Group Mutual Exclusion. Our first set of experiments compare our algorithm for group mutual exclusion with the KM algorithm and to locks. We implemented the KM algorithm using MCS locks [22] as suggested by Keane and Moir and used in their experiments. For all experiments we assume two rooms. In the experiments each processor loops for n rounds. Each round randomly selects one of the two rooms with a certain probability (a parameter of the experiment), enters the room, does some work (inside work), exits the room, and does some work (outside work). In the case of simple locks, only one processor can be in a room at a time.

We varied the number of processors, the amount of work performed inside and outside the rooms, and the ratio of requests for the two rooms. All experiments were run on from 1 to 32 processors. For the inside and outside work we report on two settings. For the *low load* setting the processors do no work inside or outside the room—they simply enter and exit rooms. This is meant to test the overhead when the requests to the room are fine grained and frequent. For the *high load* setting the processors do significant, and about equal, work inside and outside the room. The work inside the room is selected based on a Gaussian distribution with mean equal to the outside work, which is fixed. The standard deviation is selected equal to half the outside work. We include some variance in the inside work since it is a more realistic scenario than all processors requesting the room for exactly the same amount of time. For the ratio of requests to the two rooms we also report on two settings. The first setting selects each room with $p = .5$, and the second setting selects one room with $p = .1$ (leaving $p = .9$ for the other). Having imbalance in requests allows us to study better the effect of allowing concurrent entering since if one room is only accessed infrequently, this will maximize the potential benefits of concurrent entering.



(a) low-load, p = .5 (b) low-load, p = .1

(c) high-load, p = .5 (d) high-load, p = .1

Fig. 7. Comparisons of our algorithm with KM and locks.

The results for the two loads (low and high) and two ratio of requests (.5 and .1) are plotted in Figure 7. We used $n = 1000$ rounds. The experiments show that our algorithm is faster than the KM algorithm for the low load—up to a factor of 5 faster in the range from 10 to 15 processors. For the low loads the cost of the locks is not very high since no work is being done inside the lock. Locks are actually faster than our algorithm on up to about 7 processors, and faster than KM on up to about 20 processors. For the high load our algorithm and KM perform similarly. The KM algorithm does slightly better for a small number of processors (about 2–15) for the $p = .1$ case. This is because KM allows concurrent entering. Since one room is only requested infrequently, the concurrent entering often allows a processor to enter while the room is in use. As the number of processors increases this benefit decreases since the likelihood increases that some processor requests the $p = .1$ room and prevents concurrent entering. As the graphs show, the cost of locks is significant for high load since they do not allow any sharing of work in the rooms.

Shared Stack. Our second set of experiments compare implementations of a shared work stack using our room synchronizations and using locks. This is meant as a more

"real-world" example. The benchmark is loosely structured on the parallel graph traversal of a garbage collector [6]. We have therefore picked parameters and characteristics that roughly match the needs of that application. The overall structure of the benchmark is that each processor takes some work from the shared stack, does some simulated local work on what it has grabbed, and then puts work back.

The shared stack is initialized with h elements each with a count of d (the count is described below). Each processor repeatedly pops k elements from the shared stack onto its local stack, "operates" on the elements, and then pushes the entire contents of its local stack back to the shared stack. The pop of k elements is executed within a single room (for the room synchronization version) or a single lock (for the locks version). Each element with a count of c (> 0) generates two new elements of count $c-1$, and a element with a zero count simply disappears. In other words, each of the original h elements will generate $1+2+4+2^c = 2^{c+1}-1$ elements over the lifetime of the benchmark (a total of $h(2^{c+1}-1)$ elements are pushed and popped from the stack before the stack becomes empty). In addition to processing the elements on the local stack, each processor waits for a random amount of time between popping and pushing. The random time is selected uniformly between 0 and $2t_k$, where t_k is a parameter of the experiment, and is meant to represent the work associated with processing a stack element. In the case of garbage collection, such additional work might include decoding objects, copying objects, and installing forwarding pointers. We note that although the linearizability of the stack is not necessary for correct garbage collection, it is critical for the purpose of properly detecting termination.

For our experiments we use $n = 16{,}000$, $c = 11$, and $k = 500$. We ran the experiments on 1–30 processors. In all cases the average of five times for the given experiment is reported. Figure 8 shows three graphs with varying wait times (the parameter t_k). The graphs correspond, from first to last, to applications where the time to process an item is 40%, 100%, or 600% of the time it takes to transfer the items from the shared stack to the local stack. In each graph the bottom line (widely spaced dots) represents the work for the uniprocessor case when no synchronization is performed and no wait time is introduced. The next line (dotted) adds a varying amount of work reflected in the distance from the bottom line. Finally, the solid and dashed curves represent the rooms and locks versions of the benchmark with synchronization and additional work. The vertical axis represents the total work performed, which is calculated as the product of the wall-clock time and the number of processors. In all cases perfect speedup corresponds to the flat thinly spaced dotted line.

The rooms synchronization has good performance, introducing an overhead approximately equal to basic stack transfer time (without synchronization or additional work). The overhead is mostly independent of the number of processors, indicating that the speedup achieved is (after including the overhead) linear. Additionally the magnitude of the overhead is also independent of the amount of additional work introduced. In contrast, at few processors, the locks version has almost no overhead. However, as the number of processors increase, the contention in acquiring locks increases, causing a rapid performance degradation. The point at which this transition occurs varies from immediately to 10 processors and 20 processors for the three wait times, respectively. This trend is expected as the introduction of additional work between popping and pushing means that the processors spend less time locking the push and pop code.

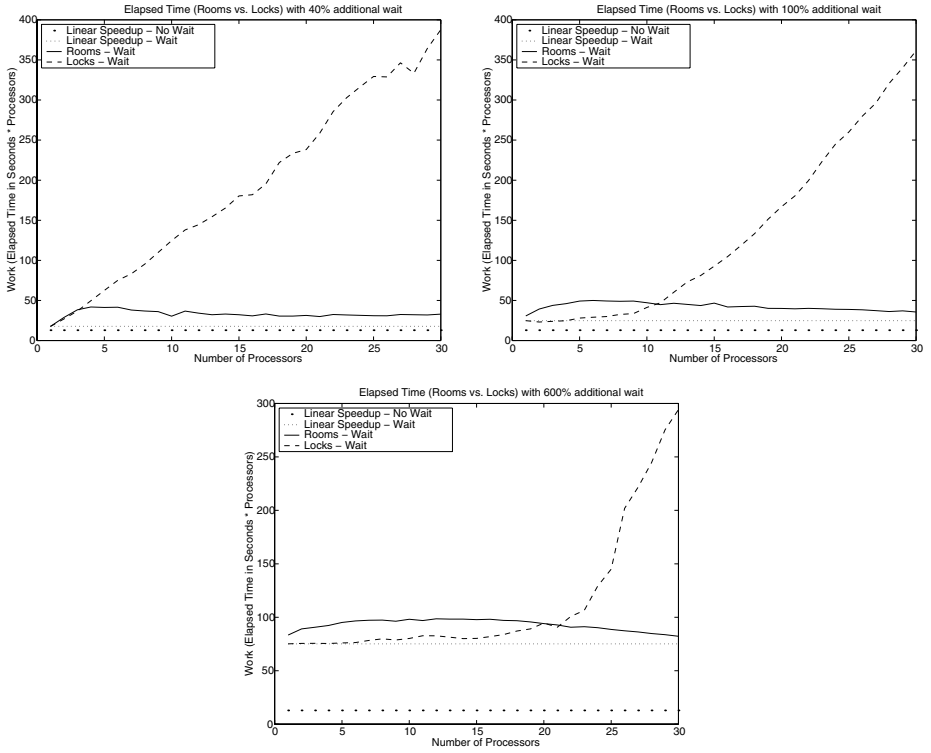


Fig. 8. Total work (elapsed time in seconds × number of processors) *versus* number of processors, for 40% additional wait time, 100% additional wait time, and 600% additional wait time.

We note that we also implemented the Treiber nonblocking stack [30], but the performance was so poor, due to a large overhead and the fact that it did not scale, that we did not include the results in this paper.

5. Related Work and Discussion

There is a long history of synchronization models and synchronization constructs for parallel and distributed computation. At the one end of the spectrum, there are synchronous models such as the PRAM, in which the processors execute in lock-step and there is no charge for synchronization. Shared data structure design is simplified by not having to deal with issues of asynchrony. Bulk-synchronous models such as the BSP [31] or the QSM [9] seek to retain the simplicity of synchronous models, while permitting the processors to run asynchronously between barrier synchronizations (typically) among all the processors. Algorithms designed for these models are *necessarily* blocking (due to the barrier synchronizations). For the loosely synchronous applications considered in this paper, there are significant overheads in implementing shared data structures using barrier synchronizations, because all the processors must coordinate/wait even if they

are not currently accessing the data structure. In many contexts, this is not practical. For example, in our parallel garbage collector, a process only needs to access the shared stack when the thread running on the process allocates memory. In general the allocation behavior of threads is completely unpredictable. It would be a major problem to suspend all threads on a regular basis so they can coordinate on a stack operation.

At the other end of the synchronization models spectrum are the fully asynchronous models, in which processors can be arbitrarily delayed or even fail, and shared data structures are designed to tolerate such delays and failures. Wait-free data structures [14] have the property that any user's request (e.g., a push or pop request) will complete in a bounded number of steps, regardless of the delays or failures at other processors. Because of the large overheads in wait-free data structures, there has been considerable work on nonblocking (or lock-free) data structures [14], which only require that *some* user's request will complete in a bounded number of steps (although any particular user can be delayed indefinitely). Examples of nonblocking data structures work includes [2], [3], [13], [14], [15], [23], [25], [32], and [33]. Most of these implementations still fully sequentialize access to the data structure. Moreover, they often require unbounded memory (because of the so-called ABA problem [32]), or the use of atomic operations on two or more words of memory (such as a double compare-and-swap or transactional memory [16], [27]). Such operations are significantly more difficult to implement in hardware than single word atomic operations. Thus, wait-free and nonblocking data structures are essential in contexts where the primary goal is making progress in highly asynchronous environments, but there is a significant cost to providing their guarantees.

Room synchronizations are designed for asynchronous settings more concerned with fast parallel access (and bounded memory) than with providing nonblocking properties. In other words, settings between those suitable for bulk-synchronous models and those suitable for fully asynchronous models. In the experimental results presented in this paper, as well as experiments with a parallel garbage collector, we have obtained good performance with room synchronizations on the Sun UltraEnterprise 10000, a 64 processor shared-memory machine. This gives some indication that room synchronizations are suitable for that machine. We expect similar performance on other shared-memory machines such as the SGI Power Challenge and the Compaq servers.

We note that our experiments are run in an environment in which each process is mapped to one processor. This means that it is unlikely that a process will be swapped out (context switched) by the operating system while inside a room. There are a couple of potential mechanisms to deal with the case where the operating system could swap out a process while inside a room. First, the interrupt for a context switch might be delayed until the `exitRoom`. This can be achieved for many situations on most processors by temporarily disabling certain kinds of interrupts while inside a room. A second potential solution is to have a special interrupt handler code that restores the state of the process to a point in which it is safe to exit the room, and then exit the room before submitting to the context switch. This would only be applicable under certain conditions.

The algorithm by Gottlieb et al. for parallel queues [12] (mentioned in the Introduction) has characteristics which our similar to ours. Like ours, the algorithm works with unpredictable arrival times or requests, is based on the fetch-and-add operation, and can fully parallelize access. Also like ours, it is not nonblocking. It, however, has some important disadvantages compared with our algorithm. Firstly, it is not linearizable—the

following can occur on two processors:

```
        P1                      P2
enqueue(v1)             enqueue(v2)
                        v1 <- dequeue()
                        EMPTY <- dequeue()
```

Secondly, the algorithm requires a lock (or counter) for every element of the queue. This requires both extra memory for each element, and manipulating this lock for every insert and delete. In our solution it is easy to batch the inserts or deletes, as was done in our experiments. Thirdly, the technique does not appear to generalize to other data structures such as stacks. The technique does have an advantage, which is that the blocking is at a finer granularity—per location rather than across the data structure.

There have been a number of papers describing techniques for reducing the contention in accessing shared data structures (e.g., [28], [26], and [29]). The diffracting trees of Shavit and Zemach [28] are not linearizable. The work of Shavit and Zemach on combining funnels leads to a linearizable and scalable implementation of stacks [29]. The idea is that pushes and pops can combine if they collide in a software combining tree. Pushes cannot efficiently combine with each other, but if two equal-size combining trees, one consisting of pushes and one consisting of pops, try to combine, then all the push requests can be combined with the pop requests. A time bound for the algorithm is not given, but experimental results on a simulator show that the technique scales well. It would be interesting to compare this technique with an implementation of linearizable stacks using our technique.

The group mutual exclusion algorithms of Joung [19] and Keane and Moir [20] sequentialize entry and exit to the critical regions. The algorithm of Joung sequentializes access by requiring every entry and exit from a critical region to loop through an array of length p. This means that even when only one user is requesting a critical region, the access takes p time. The algorithm, however, requires no synchronization primitives beyond atomic reads and writes. The algorithm of Keane and Moir [20] uses a locks (mutual exclusion) to access the key data structures needed to enter and exit a critical region. If all users concurrently request a critical region, they must sequentially access the code inside the lock. However, if only one user requests a critical region, its request can be serviced in constant time. The particular primitives they require depend on how the mutual exclusion is implemented. Their best variant, experimentally, is based on Mellor-Crummey and Scott's mutual exclusion algorithm [22] and hence requires a compare-and-swap operation. We note that it is not completely fair to compare the performance of our algorithms with theirs because we use a significantly more powerful synchronization primitive—the fetch-and-add. In effect our results are based on reducing group mutual exclusion to a fetch-and-add, and therefore potentially taking advantage of scalable implementations of the fetch-and-add.

We note that there is a difference in the semantics supported by our algorithm and those of Joung and of Keane and Moir. In particular, we do not support the *concurrent entering* requirement, and instead support the contradictory *no late entry* requirement (property P5). Either of these requirements may prove more useful depending on the context.

6. Conclusions

In this paper we presented a scalable solution to the group mutual exclusion problem. Our techniques are likely to be useful in a parallelism context that lies between highly synchronous models such as the PRAM or BSP model, and highly asynchronous models where it is assumed processors can stall, fail, or become disconnected. In particular, our algorithms can handle requests that come in at arbitrary times, and from arbitrary subsets of the processors. They, however, are blocking and hence if a processor fails in certain critical regions of the code, other processors can become blocked. Based on our room synchronization solution, we presented simple and efficient implementations of shared stacks and queues. These data structures are linearizable, handle asynchronous requests, and allow for constant-time access (assuming a constant-time fetch-and-add).

Acknowledgments

Thanks to Toshio Endo and the Yonezawa Laboratory for use of their 64-way Sun UltraEnterprise 10000, to Faith Fich for first pointing us to the previous work on group mutual exclusion, and to Nir Shavit and the anonymous referees for providing many helpful comments that improved the content and presentation of this paper.

Appendix. Bounded Counters

In this appendix we discuss the issues of bounded counters in the room synchronization protocol. The difficulty in adapting our protocol to use only bounded counters is that it employs *inequality* tests in order to admit multiple waiting users at once. If not careful, inequality tests can be foiled by the wrap around arising with bounded counters. This section contains the following results:

- We prove that the (unbounded) variables in each of the two inequality tests in the protocol never differ by more than the number of users p.
- Using this, we show how to modify the protocol to use only bounded variables, with maximum value at most B, for any $B > \max\{2p, m, r\}$, where m is the number of rooms and r is the largest value of a pointer variable in the protocol.
- We prove that the modified protocol satisfies properties P1–P7.
- We argue why, in practice, our modifications are unnecessary, because the code in Figure 2 is correct as shown (thanks to twos-complement arithmetic).

A.1. *Additional Properties of the Unbounded Protocol*

We first prove that the (unbounded) variables in each of the two inequality tests in the protocol (Step 3 of `enterRoom` and Step 19 of `exitRoom`) never differ by more than p. Step 3 compares `myTicket` at j and `grant[`i`]`; this is addressed by Lemma 3. Step 19 compares `currWait` at j and `grant[newAr]`; this is addressed by Lemma 4.

Lemma 3. *In any execution* (*with behaved traces*) *of the room synchronization protocol* (*with unbounded counters*), *if there is a user* j *with a ticket for room* i, *then* $|$`myTicket` *at* $j -$ `grant[`i`]`$| \leq p$.

Proof. We use the invariants of Lemma 2. By invariant 2, `myTicket`$-$`grant[`i`]` $\leq p$. In the bulk of the proof we show that `grant[`i`]` $-$ `myTicket` $< p$. For purposes of contradiction, consider the shortest execution σ which ends in a state such that user j has a ticket for room i and `grant[`i`]` $-$ `myTicket` $\geq p$. Let $\sigma = \sigma_1\alpha_1\sigma_2$, where α_1 is the last occurrence of Step 2 by user j, σ_1 is the prefix of σ prior to α_1, and σ_2 is the suffix of σ after α_1. Let s_1 be the last state in σ_1 and let s_2 be the updated state after α_1 occurs. By invariant 2, `wait[`i`]` $\geq$ `grant[`i`]` in s_1, and hence `myTicket` $>$ `grant[`i`]` in s_2, and user j is blocked. Thus unless σ_2 contains a step that increases `grant[`i`]` and unblocks user j, we have a contradiction.

Accordingly, let $\sigma_2 = \sigma_3\alpha_2\sigma_4$, where α_2 is the first occurrence in σ_2 of a Step 7 or 21 that increases `grant[`i`]` and unblocks user j. Let s_3 be the last state in σ_3 and let s_4 be the updated state after α_2 occurs. Assume σ_2 is a Step 7 (the case for Step 21 is similar). Step 7 updates only `grant[`i`]`. Let g_{old} (g_{new}) be the value of `grant[`i`]` in s_3 (s_4, respectively). Because user j is blocked in s_3, `myTicket` $> g_{\text{old}}$. By invariant 2, `wait[`i`]` $- g_{\text{old}} \leq p$. Because Step 7 is enabled in s_3, there is a user k in the advance room region, and hence, by invariant 2, `currWait` at $k \leq$ `wait[`i`]`. By Step 7, we have $g_{\text{new}} =$ `currWait` at k. Therefore, $g_{\text{new}} -$ `myTicket` $< g_{\text{new}} - g_{\text{old}} =$ `currWait` at $k - g_{\text{old}} \leq$ `wait[`i`]` $- g_{\text{old}} \leq p$.

Moreover, it follows from invariant 2 that `grant[`i`]` never decreases. Thus user j is unblocked in all states in σ_4. It follows from invariants 2 and 2 that in all states in σ_4, `grant[`i`]` $>$ `done[`i`]`, no user is in the advance room region, and `grant[`i`]` $= g_{\text{new}}$. Thus `grant[`i`]` $-$ `myTicket` $< p$ in the last state in σ, a contradiction. □

Lemma 4. *In any execution* (*with behaved traces*) *of the room synchronization protocol* (*with unbounded counters*), *if there is a user j with Step* 19 *enabled, then* $0 \leq$ `currWait` *at* $j -$ `grant[newAr]` $\leq p$.

Proof. By invariant 2 of Lemma 2, when Step 19 is enabled, `currWait` at $j -$ `grant[newAr]` ≥ 0. By invariants 2 and 2, `currWait` at $j -$ `grant[newAr]` $\leq$ `wait[newAr]` $-$ `grant[newAr]` $\leq p$. □

A.2. Protocol Modifications and Correctness Proof

Next, we show how to modify the protocol to use only bounded variables. Let p be the number of users, and let B be an integer greater than $2p$. We make the following changes to the code in Figure 2:

- Let any `fetchAdd` on a shared counter increment that counter modulo B. Similarly, the addition by 1 in Steps 2 and 14 are performed modulo B.
- Define a function `greater()` as follows:

```
int greater(int a, int b) {
     return ((a > b && a - b <= p) || (a < b && b - a > p));
}
```

- Replace `myTicket - r->grant[i] > 0` in Step 3 with `greater (myTicket, r->grant[i])`, and `currWait - r->grant[newAr] > 0` in Step 19 with `greater(currWait, r->grant[newAr])`.

We call this the *room synchronization protocol with bounded counters*. Note that `a` and `b` are passed by value, so we retain the property that each step contains at most one shared memory access.

The rationale behind the function `greater` is summarized by the following lemma.

Lemma 5. *Let B and p be positive integers such that $B > 2p$. For all nonnegative integers x and y such that $|x - y| \leq p$,*

$$x > y \qquad \textit{if and only if} \quad \texttt{greater}(x\%B,\ y\%B).$$

Proof. Let x_1 and x_2 be nonnegative integers such that $x_1 = x\%B$ and $x = x_1 + x_2 \cdot B$. Let y_1 and y_2 be nonnegative integers such that $y_1 = y\%B$ and $y = y_1 + y_2 \cdot B$. Because $|x - y| \leq p < B$, $|x_2 - y_2| \leq 1$. Note that $\texttt{greater}(x\%B, y\%B) = \texttt{greater}(x_1, y_1)$.

If $x_2 = y_2 + 1$, then $x > y$ and $x - y = x_1 + B - y_1 \leq p < B/2$. Thus,

$$x_1 < y_1 - B/2 \quad \text{and} \quad y_1 - x_1 > B/2 > p. \tag{A.1}$$

Suppose $x > y$. If $x_2 = y_2$, then $x_1 > y_1$ and $x_1 - y_1 \leq p$, so $\texttt{greater}(x_1, y_1)$ is true. If $x_2 = y_2 + 1$, then, by (A.1) above, $x_1 < y_1$ and $y_1 - x_1 > p$, so $\texttt{greater}(x_1, y_1)$ is true.

Suppose $x = y$. Then $x_1 = y_1$, so $\texttt{greater}(x_1, y_1)$ is false.

Finally, suppose $x < y$. If $y_2 = x_2$, then $x_1 < y_1$ and $y_1 - x_1 \leq p$, so $\texttt{greater}(x_1, y_1)$ is false. If $y_2 = x_2 + 1$, then, by (A.1), $x_1 > y_1$ and $x_1 - y_1 > p$, so $\texttt{greater}(x_1, y_1)$ is false. □

We now have the components in place to prove the correctness of the modified protocol.

Theorem A.1. *The room synchronization protocol with bounded counters satisfies properties* P1–P7.

Proof. Let A_b (A_u) be the automaton for the protocol with bounded (unbounded, respectively) counters. Let A_b^* be the automaton A_b augmented with history variables that keep track of the unbounded counters corresponding to each bounded counter.

By Lemmas 3 and 4, the two inequality tests in A_u are only applied for (unbounded) variables that differ by at most p. Thus based on Lemma 5, there is a straightforward correspondence between the states and actions in A_b^* and those in A_u, such that any execution of A_b^* can be mimicked by A_u (formally, there is a *simulation relation* from A_b^* to A_u [21]). It follows that any trace of A_b is a trace of A_b^* is a trace of A_u, and hence the safety properties of A_u carry over to A_b (i.e., properties P1, P2, P5, and P6). The proofs of the remaining properties follow the proofs of the same properties for Theorem 1. □

A.3. *Correctness of the Original Code*

Finally, a key observation is that in practice, equivalent arithmetic and comparisons occur automatically, so that *the code in Figure* 2 *is correct as shown*. This is because when

integers are represented in twos-complement, then letting the counters wrap around to a negative number (ignoring the overflow) gives the desired result for any inequality test in the protocol. More specifically, consider an inequality test between two numbers x and y. The inequality test $x - y > 0$, when x and y are represented as twos-complement integers with a bounded number of bits, gives the same result as the inequality test $x > y$ when x and y are represented as unbounded integers, as long as x and y differ by less than the maximum integer representable by the given number of bits. By Lemmas 3 and 4, the counters differ by at most the number of users p, which we assume is less than the maximum integer. Note that it is important to use `|myTicket - r->grant[i] > 0|` instead of `|myTicket > r->grant[i]|` in the `enterRoom` code (Step 3, see also Step 19), in order for the (bounded variable) comparison to evaluate as desired for correctness.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] O. Agesen, D. L. Detlefs, C. H. Flood, A. T. Garthwaite, P. A. Martin, N. N. Shavit, and G. L. Steele, Jr. DCAS-based concurrent deques. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, July 2000.

[3] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[4] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Languages Design and Implementation*, pages 104–117, May 1999.

[5] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronizations. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 122–133, July 2001.

[6] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Languages Design and Implementation*, June 2001.

[7] I. Dimitrovsky. A Group Lock Algorithm with Applications. Technical Report, Courant Institute, New York Univeristy, Nov. 1986.

[8] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, Jan. 1989.

[9] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? *Theory of Computing Systems*, 32(3):327–359, 1999.

[10] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Apr. 1989.

[11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, 32(2):175–189, Feb. 1993.

[12] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, Apr. 1983.

[13] M. Greenwald. Non-Blocking Synchronization and System Design. Ph.D. thesis, Stanford University, Palo Alto, CA, 1999. Technical Report STAN-CS-TR-99-1624.

[14] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, Jan. 1991.

[15] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[16] M. P. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the* 20*th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[17] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the* 14*th ACM Symposium on Principles of Programming Languages*, pages 13–26, Jan. 1987.

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

[19] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[20] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Trasactions on Parallel and Distributed Computing*, 12(7):673–685, 2001.

[21] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[22] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[23] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the* 15*th ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[24] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[25] M. C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, Nov. 1999.

[26] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30(6):645–670, Nov. 1997.

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

[28] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[29] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, Nov. 2000.

[30] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[31] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(9):103–111, Sept. 1990.

[32] J. D. Valois. Implementing lock-free queues. In *Proceedings of the* 7*th International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Oct. 1994.

[33] J. D. Valois. Lock-Free Data Structures. Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY, 1995.

[34] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, Apr. 1987.

Received January 16, 2002, *and in revised form March* 6, 2003, *and in final form May* 13, 2003.
Online publication August 8, 2003.