Room Synchronizations*

Guy E. Blelloch Perry Cheng
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{guyb,pscheng}@cs.cmu.edu

Phillip B. Gibbons
Information Sciences Research Center
Bell Laboratories
Murray Hill, NJ 07974
gibbons@research.bell-labs.com

ABSTRACT

We present a class of synchronization called *room synchronizations* and show how this class can be used to implement asynchronous parallel queues and stacks with constant time access (assuming a fetch-and-add operation). The room synchronization problem involves supporting a set of m mutually exclusive "rooms" where any number of users can execute code simultaneously in any one of the rooms, but no two users can simultaneously execute code in separate rooms. Users asynchronously request permission to enter specified rooms, and neither the arrival time nor the arrival order nor the desired room of such requests are known ahead of time. We describe an algorithm for room synchronizations, and prove it satisfies a number of desirable properties. We have implemented our algorithm on a Sun UltraEnterprise 10000 multiprocessor. We present experimental results comparing an implementation of a parallel stack using room synchronizations to one using locks, demonstrating a significant scalability advantage for room synchronizations.

1. INTRODUCTION

There has been a long history of developing data structures for handling asynchronous parallel accesses—i.e., accesses for which neither the arrival times nor the number of 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, but they can fully sequentialize the access. Furthermore locks have the problem that if the process with the lock is blocked (e.g., swapped out by the OS or dies), then all processes can become blocked.

To avoid this problem many non-blocking (or lock-free) implementations of data-structures have been suggested [1, 2, 9, 10, 17, 18, 24, 25]. As with the versions that use locks, however, these implementations still sequentialize the access. For example the implementations of non-blocking queues [13, 17, 24] and stacks [22] sequentialize the inserts and deletes. Furthermore, most of these implementations have other problems such as requiring an atomic double compare-and-swap operation or requiring unbounded memory.

An asynchronous implementation of queues that does not sequentialize access was developed by Gottlieb et al. [8]. Their implementation uses a parallel fetch-and-add operation, and it is not non-blocking. Unfortunately, it does not implement a *linearizable*[1] queue (we give a counterexample in Section 5). It also seems unlikely that the technique can be extended to other data structures such as stacks.

Our goal is to develop data structures for handling asynchronous parallel accesses that (a) are linearizable, and (b) do not sequentialize access. The problem arose in the context of a real-time parallel garbage collector [3, 5]. In such contexts, whenever one of the program threads allocates memory, it also participates in the garbage collection. Thus the threads access the collector data structures asynchronously, in parallel, and somewhat unpredictably. Linearizability is needed for correctness. Avoiding sequential access is needed to guarantee real-time bounds, and more generally to insure good performance.

To achieve our goals, in this paper we introduce a class of synchronizations called room synchronizations. The *room synchronization* problem involves a set of m mutually exclusive rooms in which any number of users can execute code simultaneously in any one of the rooms, but one or more users can not be simultaneously in more than one room. Users enter and exit a room by executing enter and exit routines. We describe an algorithm that supports room synchronizations. For a constant number of rooms, and as long as each user only runs for constant time within a room, the implementation guarantees that a user will never wait more than constant time to enter a room (more specifically, time proportional to a fetch-and-add).

We show how linearizable stacks and queues can be easily implemented using room synchronizations, all with constant time access regardless of the number of processors. We also present experimental performance results for a shared work stack. We compare our version using room synchronizations with a locking version of a stack, on up to 30 processors. Although, not surprisingly, the lock based implementation

*This work was supported in part by the National Science Foundation under grants CCR-9706572 and CCR-0085982.

[1] A linearizable data structure [14] has the highly desirable property that the high-level data structure operations (such as push, pop, enqueue, dequeue) can be viewed as atomic.

SPAA 2001 Crete Island Greece

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

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

(a)

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

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

(b)

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

degrades linearly with the number of processors (because accesses are sequentialized), our version scales almost perfectly, beyond an initial cost for going from one to two processors. The room synchronizations introduced in this paper have been incorporated into our real-time garbage collector code [5], where they are used extensively.

One disadvantage of room synchronizations is that if a user fails or stops while inside a room, the user will block other users from entering another room. This property is clearly not desirable under all conditions, but in many conditions, including the garbage collector, it is not a problem. We discuss this issue in Section 5.

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)` 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 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 the full paper [4] we show that using our room synchronization algorithm (a) asynchronous accesses (pushes or pops) can arrive at any time, (b) every access is serviced in time proportional to the time of a fetch-and-add[2], and (c) that this implements a linearizable stack. 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 formally defines room synchronizations, and presents an algorithm that provably implements them. Section 3 shows how room synchronizations can be effectively used to implement shared queues and dynamic shared stacks. Section 4 presents our experimental results. Section 5 discusses further issues and related work, and Section 6 concludes.

2. ROOM SYNCHRONIZATIONS

The room synchronization problem involves a set of m rooms, and a set of p independent users who wish to access the rooms. A user wishing access to a room calls an Enter Room primitive, which returns once the user is granted permission to enter the room. The user is then said to be *inside* the room. When done with the room, the user exits the room by calling an Exit Room primitive. The rooms synchronization construct must ensure that at any point in time, there is at most one "open" room with users inside it. However, any number of users can be inside the open room.

In this section, we provide details on the room synchronization problem, discuss necessary and desirable properties for protocols implementing room synchronization, present a protocol that implements room synchronization, and show that this protocol satisfies all these properties.

2.1 Primitives

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

[2] This is under certain assumptions about all processors making progress.

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. In addition to these four basic primitives, there is a Change Room primitive, which is equivalent to an Exit Room followed by an Enter Room, but with the guarantee of immediate entry into the requested room if it is the next room to be opened. There is also an Assign Exit Code primitive, discussed at the end of Section 2.3.

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.

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 will 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 Formalization

We can formalize the room synchronization problem using the I/O Automaton model [16]. Our terminology and formal model are an adaptation of those used in [16] 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.[3]

[3]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.

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. We will consider various fairness metrics for executions and for room accesses. 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).

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:

- $\mathrm{EnterRoomReq}_j(i)$: the action of user j signalling to its agent p_j a desire to enter room i.
- $\mathrm{EnterRoomGrant}_j(i)$: the action of agent p_j signalling to user j that its Enter Room request has been granted.
- $\mathrm{ExitRoomReq}_j$: the action of user j signalling to its agent p_j a desire to exit its current room.
- $\mathrm{ExitRoomGrant}_j$: 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 the room synchronization problem and protocols that provide room synchronizations. 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.

Necessary 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:

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

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 $\mathrm{EnterRoomReq}_j$ or $\mathrm{ExitRoomReq}_j$ 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, $\mathrm{EnterRoomReq}_j(i)$ transitions user j from *outside* all rooms to *preparing to enter* room i, 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*.

We can now state formally what it means for a protocol to implement room synchronization. A protocol automaton A *solves the room synchronization 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.[4] Equivalently, in any execution, between any 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 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. **Global progress:** At any point in a weakly-fair execution: (1) If there is a user j preparing to enter room i (i.e., its most recent external action is an EnterRoomReq$_j(i)$), and there are no rooms with users inside, then at some later point some user is inside some room (i.e., there is a EnterRoomGrant action). (2) If there is a user j preparing to exit a room (i.e., its most recent external action is an ExitRoomReq$_j$), then at some later point user j is outside the room (i.e., there is an ExitRoomGrant$_j$ action).

Note that properties P1–P4 do not guarantee any fairness among rooms or among users, the delays in entering or exiting a room, etc. These are considered next.

Desirable properties. Properties P5–P11 define additional desirable properties for a protocol automaton A solving the room synchronization problem. It is useful to analyze protocol performance (such as in P6, P7, P9, and P10 below) assuming an upper bound δ on the (wall clock) time between successive actions by an agent.[5]

P5. **No room starvation:** In any weakly-fair execution: If all users inside a room eventually prepare to exit the room, then any closed room requested by at least one user is eventually opened.

P6. **Bounded delay for rooms:** In any execution with m rooms and p users: If each user is inside a room for at most time α, and the time between successive actions of each agent preparing to enter or preparing to exit is at most δ, then any closed room requested by at least one user is open within time $T_R = T_R(\alpha, \delta, m, p)$.

P7. **Constant delay for rooms:** This is a stronger version of P6 in which T_R is independent of p.

P8. **Lockout-freedom** (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. Note that this is a stronger property than P5.

P9. **Bounded time to enter and exit:** In any execution, any user preparing to enter a room is inside the room within time $T_1 = T_1(\alpha, \delta, m, p)$, and any user preparing to exit a room is outside the room within time $T_2 = T_2(\delta, m, p)$, where α, δ, m and p are as defined in P6.

P10. **Constant time access:** This is a stronger version of P9 in which T_1 and T_2 are independent of p. Note that this implies unbounded concurrent access to rooms (otherwise the time bound would necessarily depend on p).

P11. **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.

2.3 A room synchronization protocol

We now present a protocol (an algorithm) for solving the room synchronization problem that satisfies all the properties P1–P11. The protocol assumes a linearizable shared memory [14] supporting atomic reads, writes, fetch-and-adds, and compare-and-swaps on single words of memory.[6] In our actual implementation of this protocol, we do not assume a linearizable shared memory, and instead explicitly insert Memory Barriers into the code (details in the full paper [4]).

Consider a rooms object with m rooms. It 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. The special value `-1` 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.

The protocol is depicted in Figure 2.[7] For conciseness and readability, we present C code instead of a full I/O Automaton specification. Agents 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).

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

[5] The time bound only applies if the second action is enabled no later than when the first action occurs: an agent blocked waiting for, say, a request from its user can be arbitrarily delayed. Also, note that in the absence of a positive lower bound on the time between successive actions, we have not restricted the relative speeds of the agents. Finally, note that the time bound applies only to the analysis of the time performance, and not to any correctness (safety) properties.

[6] We have explicitly avoided atomic operations on two or more words of memory. We use only the weaker fetch-and-increment form of fetch-and-add. Also note that we could in principle implement rooms using only reads and writes, following ideas used, e.g., in the Bakery Algorithm [15] for mutual exclusion, at the cost of far greater delays.

[7] We show only the code for enterRoom and exitRoom. The complete version of the code is available at: `www.cs.cmu.edu/afs/cs/project/pscico/www/rooms.html`

```
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 (r->activeRoom == -1) {                          // while waiting, if no active room
5        if (compareSwap(&r->activeRoom,-1,i) == -1) {     // then make i the active room
6          r->grant[i] = r->wait[i];                       // enable all with tickets to enter room i
7          break;
8        }
9      }
10   }
11 }

12 int 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 thru the rooms
18       if (r->wait[newAr] - r->grant[newAr] > 0) {       // if ticketed waiters
19         r->activeRoom = newAr;                          // make it the active room
20         r->grant[newAr] = r->wait[newAr];               // enable all with tickets to enter room newAr
21         return 1;
22       }
23     }
24     r->activeRoom = -1;                                 // no waiters found, so no active room
25     return 1;                                           // 1 indicates was last to be done
26   }
27   return 0;                                             // 0 indicates was not the last
28 }
```

Figure 2: C code for Enter Room and Exit Room.

Agents exit a room by incrementing the `done` counter (Step 14). Once the `done` counter matches the `grant` counter (Step 15), then all agents granted access to the room have exited the room. The unique agent to increment the `done` counter up to the `grant` counter (the "last done") does the work of selecting the next active room (Steps 16–23). The `grant` counter of that active room is set to be equal to its current `wait` counter (Step 20), thereby granting all waiting tickets for that room. If the last done agent fails to discover a room with waiting tickets, it resets `activeRoom` to -1 (Step 24). Whenever `activeRoom` $= -1$ (Step 4), a ticketed agent can self-select its requested room as the next active room (Step 5), and grant all waiting tickets for that room (Step 6).

THEOREM 1. *The room synchronization protocol in Figure 2 satisfies properties P1–P11.*

PROOF. Due to page constraints, we only present the proofs of property P1 (trace behaved) and property P2 (mutual exclusion among rooms). Other properties are discussed briefly, with the details left to the full paper [4]. We also restrict our attention to the case where `wait`, `grant`, and `done` are unbounded counters, and hence they are monotonically nondecreasing. For ease of description, we view each step of the C code as an atomic action. This is done without loss of generality for all steps that access at most one shared memory location. For the other steps (Steps 6, 18, 20), the proof is readily extended to have separate atomic actions for each shared memory access. Finally, 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).

Property P1. 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 P2. To prove mutual exclusion, we will need the following definitions and lemmas. 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 Step 6, 16, 17, 18, 19, 20, or 24 is enabled, or Step 15 is enabled with a successful conditional test.

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.

LEMMA 2. *In any execution σ (with behaved traces), if a user j is inside room i after σ, then j is an unblocked user with a ticket for room i.*

PROOF. 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 by user j that modifies `myTicket` at j: its value is the same in all states in σ_2. Because `grant[`i`]` is a nondecreasing counter, user j's `myTicket` $>$ `grant[`i`]` in all states in σ_2. Thus j would never 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.

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

LEMMA 3. *In any execution (with behaved traces):*

1. *For all rooms i,* `wait[`i`]` $-$ `done[`i`]` *is the number of users with tickets for room i, and* `wait[`i`]` $-$ `grant[`i`]` *is the number of blocked users with tickets for room i. (Thus* `grant[`i`]` $-$ `done[`i`]` *is the number of unblocked users with tickets for room i.) All users with tickets for room i have* `myTicket` $\leq$ `wait[`i`]`.

2. *At most one user is in the advance room region. If a user is in the advance room region, then* `activeRoom` $\neq -1$ *and for all rooms i,* `grant[`i`]` $=$ `done[`i`]`.

3. *If there exists an unblocked user with a ticket for room i then* `activeRoom` $= i$. *If Step 6 (14, 20) is enabled at a user j, then* `activeRoom` $= i$ (`ar`, `newAr`, *respectively) at j and* `activeRoom` $\neq -1$.

PROOF. The proof is by induction on the number of actions in the execution. Initially, `wait[`i`]` $=$ `grant[`i`]` $=$ `done[`i`]` $= 0$, and all three invariants hold for the start state. Assume that all three 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. W.l.o.g., assume that α is an action by user j or its agent. Let s_1 be the last state in σ and s_2 be the updated state after α occurs.

To show invariant 1 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, 6, 14, and 20. Step 2 of enterRoom(i) increments both `wait[`i`]` and the number of (blocked) users with tickets for room i (by Lemma 1). In addition, each `myTicket` is at most `wait[`i`]` in s_2. Thus invariant 1 is maintained. Step 6 of enterRoom(i) sets `wait[`i`]` $-$ `grant[`i`]` to zero. It follows inductively by invariant 1 that any user with a ticket for room i in s_1 must have `myTicket` $\leq$ `wait[`i`]` $=$ `grant[`i`]` in s_2, and hence cannot be blocked. Likewise, Step 20 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 that in the proof of Lemma 2, j is not blocked. Thus inductively by invariant 3, `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 1 holds in s_2.

To show invariant 2 holds in s_2, we again consider each relevant case for α, namely, Steps 5–6, 14–20, and 24. If α is a Step 5 that succeeds in enabling Step 6 in s_2, then `activeRoom` $= -1$ in s_1 (otherwise, the compareSwap would not return -1). Inductively by invariants 1 and 3, `grant[`i`]` $=$ `done[`i`]` for all rooms i in s_1, and hence in s_2. Inductively by invariant 2, there are no users in the advance room region in s_1. Thus the invariant is maintained. If α is a Step 6, 15 (with a successful conditional), 16–20, or 24, then user j is in the advance room region in s_1. Inductively by invariant 2, j is the only such user in s_1. If α is a Step 6, 20, or 24, then there are no users in the advance room region in s_2, and the invariant is maintained. On the other hand, if α is a Step 15–19, then inductively by invariant 2 and the fact that none of these steps add a user to the advance room region, set `activeRoom` $= -1$, update `grant`, or update `done`, 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 invariants 1 and 3, `grant[`i`]` $=$ `done[`i`]` in s_1 and hence s_2 for all rooms $i \neq$ `ar`, 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. Hence, in all cases, invariant 2 holds in s_2.

Finally, to show invariant 3 holds in s_2, we consider each relevant case for α, namely, Steps 2, 5, 6, 13, 19, 20, and 24. If α is a Step 2, then by invariant 1 applied to both s_1 and s_2, the number of unblocked users with tickets for room i is unchanged. Moreover, `activeRoom` is unchanged, so the invariant is maintained. If α is a Step 5 that succeeds in changing `activeRoom`, then `activeRoom` $= -1$ in s_1 and $i \neq -1$ in s_2. Thus, inductively by invariant 3 and the fact that Step 5 does not create an unblocked ticketed user, the invariant is maintained. If α is a Step 6, then inductively by invariant 3, `activeRoom` $= i$ in s_1 and hence in s_2. Step 6 can only unblock users with tickets for room i, so the invariant is maintained. The case for Step 20 is symmetric. If α is a Step 13, then `activeRoom` is 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 `ar` $\neq -1$. Step 13 neither creates a new unblocked ticketed user nor changes `activeRoom`, so inductively by invariant 3, the invariant is maintained. If α is a Step 19, then user j is in the advance room region in s_1, and hence inductively by invariant 2, there is no other user in the advance room region in s_1. Thus inductively by invariants 1 and 2, 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, Step 6 and Step 20 are enabled at some user j' only if j' is in the advance room region. Thus neither step is enabled in s_1, and hence in s_2, with the exception of Step 20 being enabled at user j in s_2. But α sets `activeRoom` $=$ `newAr` $\neq -1$, as is required. Hence, the invariant is maintained. The case for Step 24 is the same as for Step 19, except that Step 20 will not be enabled at user j. Hence, in all cases, invariant 3 holds in s_2.

This concludes the proof of Lemma 3.

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 Lemma 2, j (j') is an unblocked user with a ticket for room i (i', respectively). Thus by invariant 3 of Lemma 3, `activeRoom` equals both i and i', a contradiction.

Other properties. Let m be the number of rooms and p be the number of users. Intuitively, the remaining properties hold due to the following observations. (1) A user desiring a ticket will get a ticket within a constant number of its 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. (3) A room with ticketed users gets its turn within m turns or less. (4) Each ticketed user in the enterRoom while loop will get inside the room within a constant number of its actions after access is granted to the room. (5) If we place an upper bound α on the time a user is inside a room, and an upper bound δ on the time between successive actions of an agent preparing to enter or exit a room, then we can compute the time to enter or exit a room. Each turn for a room takes at most $\alpha + O(\delta)$ time, thus in any execution, any user preparing to enter a room is inside the room in time $T_1 \leq (\alpha + O(\delta)) \cdot m$. Moreover, in any execution, any user preparing to exit a room is outside the room in time $T_2 = O(\delta \cdot m)$ for the *last done* and $O(\delta)$ for all others.

In the above time analysis, we are implicitly assuming that δ 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 $T_1 \leq (\alpha + 2 \cdot t_f + O(\delta)) \cdot m$ and $T_2 \leq t_f + O(\delta \cdot m)$.

Note that all these time bounds and other properties hold even in the presence of arbitrarily fast agents who do their best to starve other agents. The reader is referred to the full paper [4] for the proofs. □

Remarks. The technique of using a `wait` counter and a `grant` counter is used in mutual exclusion protocols such as TicketME [6, 16]. Mutual exclusion protocols are simpler because only one agent is granted access at a time. Thus the `grant` counter can double as the `done` counter. Also, the test for granting access is simply whether your ticket *equals* the `grant` counter, so modulo p counters are used for p users. With room synchronization, we need an inequality test between the ticket and the `grant` counter in order to admit multiple waiting users at once, so modulo p counting does not suffice. Instead, and to avoid unbounded counters, we rely on the fact that when integers are represented in twos-complement (and p is less than the maximum integer), then letting the counters wrap around to a negative number (ignoring the overflow) gives the desired result for any inequality test in the protocol. This is the reason for using `myTicket - r->grant[i] > 0` instead of `myTicket > r->grant[i]` in the enterRoom code.

Note that `activeRoom` may be set to -1 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 properties P9 or P10, because within time $t_f + O(\delta)$ the slow agent will grab a ticket, and from there will proceed in constant time (a function of α, δ, 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 protocol has the property that any user requesting to enter an already open room cannot enter the room until all users inside the room have exited.

Variants. 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 21, 25, and 27.

We implemented a version of the Exit Room primitive that includes special *exit code*. Exit code is assigned to a room using an Assign Exit Code primitive 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. 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 4 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.

3. APPLICATIONS

In this section we describe two additional applications of room synchronizations: a shared queue and a dynamic shared stack.

FIFO queue. The implementation of a linearizable FIFO queue is given in Figure 3. The `newQueue` routine creates a new queue object, including allocating an array of a fixed size `mysize` and calling createRooms to create two rooms associated with the queue (one for enqueuing and one for dequeuing). 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).

```
1  struct queue {
2    val *A;
3    unsigned size;
4    int top, bot;
5    Rooms_t *r;
6  } Queue_t;

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

17 int ENQUEUEROOM = 0;
18 int DEQUEUEROOM = 1;

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

31 val dequeue(Queue_t *q) {
32   val x;
33   enterRoom(q->r, DEQUEUEROOM);
34   int j = fetchAdd(&q->bot,1);
35   if (q->top - j <= 0) {
36     fetchAdd(&q->bot,-1);
37     x = EMPTY;
38   }
39   else x = q->A[j % q->size];
40   exitRoom(q->r);
41   return x;
42 }
```

Figure 3: The code for a parallel queue using room synchronizations.

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 goes back and decrements 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 the proof below we assume there is no overflow. The code, however, works correctly even with overflow as long as the range `int` is greater than `2 * (q->size + P)` for P processes, and as long as the range of `int` is a multiple of `q->size`.[8] Alternatively we could use exit code (see the end of the previous section) to reset the counter when it is close to overflowing.

THEOREM 2. *The algorithm of Figure 3 implements a linearizable FIFO queue, such that both the enqueue and the dequeue operations take $O(t_f+\delta)$ time regardless of the number of concurrent users, where t_f is an upper bound on the time for a fetch-and-add, and δ is an upper bound on the time for any other instruction.*

Proof: We first show the queue is linearizable. Consider a collection of p users executing enqueue and dequeue operations on a queue, and p agents executing enterRoom and exitRoom operations in response to user requests, as indicated in the code. Let σ be the execution comprising the actions by both users and agents and the complete state after each action. Consider the subsequence of the actions in σ comprised of the fetch-and-add actions generated by Step 22 of the enqueue operation and Step 34 of the dequeue operation. We call these the *commit* actions. We argue that the ordering of these commit actions specifies a proper linearized order of the corresponding queue operations.

Consider the enqueue. We call any region between a commit action of an enqueue and the next EnterRoom-Grant(`DEQUEUEROOM`) action a *safe enqueue region.* Because of property P2 of the rooms, this region will contain no actions from the body of the dequeue. In particular `q->bot` will not change, and no dequeue will commit. Consider an enqueue operation. In the case that the queue is not full (the else branch is taken) the committing fetch-and-add effectively reserves a location to put the item to enqueue. Any commits from other enqueues coming after but within the safe enqueue region will reserve higher, and hence later in "time", locations in the queue (because of the properties of fetch-and-add). Also, all writes to the reserved locations in Step 27 will complete while still in the safe enqueue region (because of property P2 of the rooms). Hence these enqueues will have the proper linear order. If an enqueue operation does return `OVERFLOW` (the if branch is taken), then any other enqueues coming after but within the safe enqueue region will also return `OVERFLOW`. This is because only an enqueue that increments `q->top` such that `q->top - q->bot > q->size` will take the if branch. By decrementing `q->top` by one in Step 24 it cannot make it the case that `q->top - q->bot < q->size`. Since no dequeue can commit in a safe enqueue region, having all future enqueues within the region return `OVERFLOW` is the expected results of the specified linear order. The argument for the proper linearized order of the dequeues is similar to that for enqueues.

In regards to time, we note that the time that a process is in a room is bounded by the time for two fetch-and-adds (on Steps 22 and 24 of enqueue, for example), 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 $\alpha = 2t_f + O(\delta)$ time. Based on Theorem 1, the maximum time a processor will wait to enter or exit a room is proportional to t_f, α and δ (e.g., the time to enter a room is bounded by $m \cdot (\alpha + 2t_f + O(\delta))$). Therefore the total time to enter, process, and exit is $O(t_f + \delta)$. □

Dynamic stack. We now consider the implementation of a linearizable 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. Each time it grows, we double the allocated space, and the old stack is copied to the new larger one. In practice such dynamic stacks are very 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 shown in Figure 4. The copying from a smaller to a larger stack is implemented incrementally. We assume that `INITSIZE` is greater than the maximum number of concurrent users. The `pushRoomExit` routine is assigned as the exit code to

[8] As with the rooms code, this takes advantage of twos-complement arithmetic and is sensitive to the particular way the comparisons are made in Steps 22 and 34.

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

struct stack {
   val *A;    /* current stack */
   val *B;    /* prev stack */
   int copy;  /* being copied? */
   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->copy = 1;
     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 = NULL;
   s->top = s->copy = s->start = 0;
   s->size = INITSIZE;
   s->r = createRooms(2);
   assignExitCode(s->r, PUSHROOM,
                  pushRoomExit, s);
   return s;
}

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

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

Figure 4: The code for a parallel dynamic stack.

the PUSHROOM. For simplicity, we have left out some details, such as handling possible failures of malloc.

THEOREM 3. *The algorithm of Figure 4 implements a linearizable stack of dynamic size, such that both the push and the pop operations take $O(t_f + t_a + \delta)$ time regardless of the number of concurrent users, where t_f is an upper bound on the time for a fetch-and-add, t_a is the upper bound on the time for a call to* malloc *or* free *and δ is an upper bound on the time for any other instruction.*

We prove this in the full paper [4]. We note that since the blocks of memory are powers of two, allocating from a shared pool should be quite cheap using a buddy system (i.e., t_a should be small).

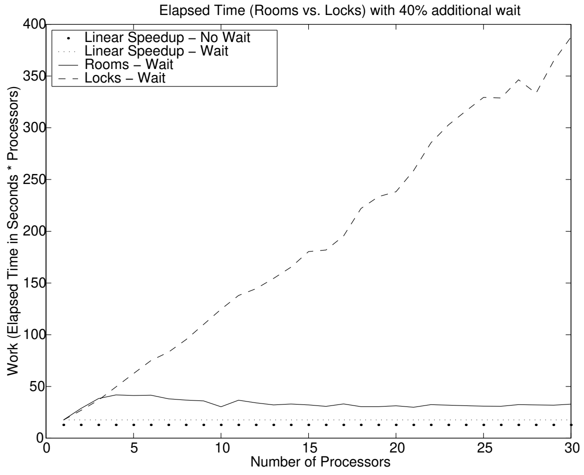
4. EXPERIMENTAL RESULTS

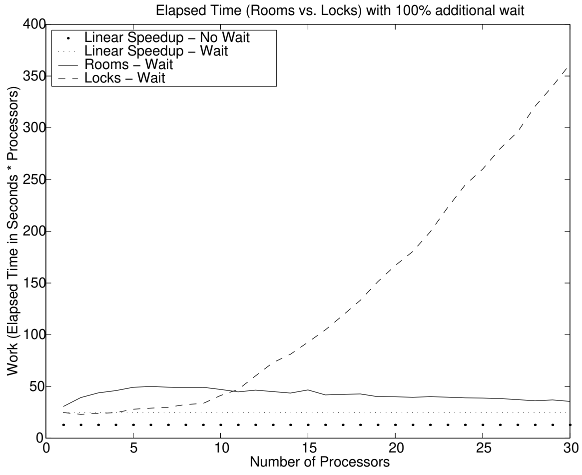
In this section we describe the results of a benchmark that uses a shared stack to distribute work across processors. The experiments were performed on an Sun UltraEnterprise 10000 with 64 250Mhz UltraSparc-II processors. This is a shared-memory machine with a compare-and-swap instruction, but no fetch-and-add. The fetch-and-add is therefore simulated using the compare-and-swap. We demonstrate the effectiveness of room synchronizations by comparing the performance of the shared stack using rooms and using locks. We also implemented the Treiber non-blocking stack [21]. However, the performance of the non-blocking stack was so poor due to a large overhead and general unscalability that we did not include the results in this paper.

The benchmark is loosely structured after the parallel graph traversal of a garbage collector [5]. When the benchmark first executes, the shared stack is initialized with 16,000 nodes with a count of 11. Each processor repeatedly pops 500 nodes from the shared stack onto its local stack, operates on the nodes, and then pushes the entire contents of its local stack back to the shared stack. The pop of 500 nodes, and push of the local stack are each done within a single room (for the room synchronization version) or a single lock (for the locked version). A node with a count of $k(>0)$ turns into two nodes of count $k-1$, and a node with a zero count simply disappears. In other words, each of the original 16,000 nodes is the root node of a fully balanced binary tree of depth 11. In addition to processing the nodes 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.

An interesting problem posed by the shared stack benchmark (and also present in the garbage collector) is detecting termination, which should occur only when there are no nodes left for any processor to work on. One can correctly test whether the *shared stack* is empty by using suitably designed exit code in the push room. However, detecting that the shared stack is empty ignores the work on the local stacks of each processor in its work section (i.e., each processor not trying to enter the pop room and not having just left the push room). To solve this, we centralize the emptiness condition by including in the shared stack a counter indicating how many local stacks have borrowed items from the shared stack and are (possibly) non-empty. This counter is updated when a processor performs a push or pop. Termination is triggered when the shared stack is empty and the counter is simultaneously zero.

For each setting of the parameters (locks vs. rooms, number of processors (1 to 30), and an amount of additional work), the experiments were run 5 times and the averages





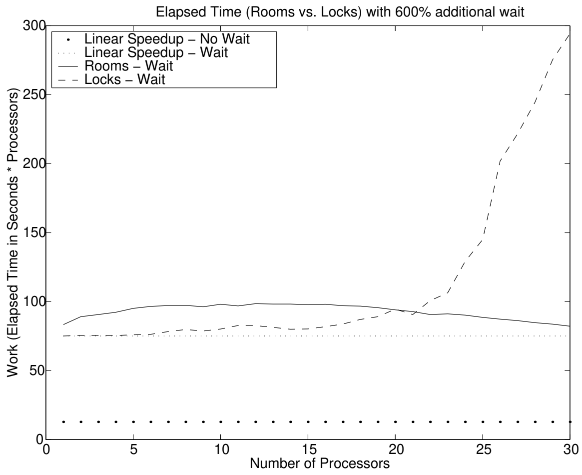


Figure 5: Total work (elapsed time in seconds $*$ number of processors) *vs.* number of processors, for 40% additional wait time, 100% additional wait time, and 600% additional wait time.

are reported. Figure 5 shows three graphs with differing amounts of additional wait time (the parameter t_k). The graphs correspond, from left to right, 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 wall-clock time and 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.

Given that we simulate the fetch-and-add with a compare-and-swap, which sequentializes the fetch-and-add, one might wonder why the room version scales so well. Recall, however, that we need only execute the fetch-and-add a small constant number of times per room access: once when entering, once when leaving and twice in each of the push and pop codes. This compares with the 500 elements that have to be copied off the stack during a pop, and from 0 to 1000 that need to be copied back onto it during a push. The proportion of instructions spent in code that might be sequentialized is therefore small. Asymptotically our code would eventually cause the work to increase linearly with the number of processors, but the slope would be much less than that of the locked version. This is a subtle and crucial point about room synchronization and a reason why our heavy use of fetch-and-adds does not preclude good performance even on machines that do not support a parallel fetch-and-add.

5. RELATED WORK AND DISCUSSION

Synchronization. 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 [23] or the QSM [7] 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. To reduce the number of barriers, coordination primitives such as parallel prefix or fetch-and-add are still needed to enable processors to make parallel updates to the data structure. In loosely synchronous applications having multiple classes of operations that must be separated (e.g., the push class and the pop class) and unpredictable arrival times for operation requests, it is desirable to have flexibility in selecting which class is permitted during the next phase. The application would be forced to implement this flexibility, and reason about fairness and other issues, whereas this flexibility is already encapsulated in room synchronizations.

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 [10] 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 non-blocking (or lock-free) data structures [10], 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 non-blocking data structures work includes [1, 2, 9, 10, 11, 17, 18, 24, 25]. Most of these implementations still fully sequentialize access to the data structure. Moreover, they often require unbounded memory[9], or the use of atomic operations on two or more words of memory (such as a double compare-and-swap or transactional memory [12, 20]). Such operations are significantly more difficult to implement in hardware than single word atomic operations. Thus, wait-free and non-blocking 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 non-blocking properties. In other words, settings somewhat in between those suitable for bulk-synchronous models and those suitable for fully asynchronous models. In the experimental results in this paper, as well as experiments with our parallel garbage collector, we have obtained good performance with room synchronizations on the Sun UltraEnterprise 10000, a 64 processor shared-memory machine, indicating 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 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 on most processors by temporarily disabling certain kinds of interrupts while inside a room. A second potential solution is to have 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.

[9] In such algorithms, memory can never be reused, because a delayed processor may still have a pointer to the old memory location. A discussion of this problem for a lock-free queue algorithm is in [24]. A compare-and-swap is used to insert or delete from the queue, so a delayed processor should fail in its compare-and-swap because the queue has changed in the meantime. However, if the memory is reused, then the compare-and-swap may succeed when it should fail.

Locks and other constructs. Traditional implementations of shared data structures use locks. Often locks are distinguished as either *shared* or *exclusive*. A shared lock permits other users to also be granted shared locks to the data structure, but forbids the granting of an exclusive lock. An exclusive lock forbids the granting of any other locks. Room synchronization can be viewed as providing a lock that is shared among those requesting the same room, but exclusive to the shared locks for other rooms. This greater sharing implies greater concurrency.

The closest algorithms to any of ours is an algorithm by Gottlieb, Lubachevsky and Rudolph for parallel queues [8]. 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 non-blocking. It, however, has some important disadvantages compared to ours. Firstly, it is not linearizable—the following can occur on two processors:

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

Secondly, the algorithms requires a lock (or counter) for every element of the queue. This both requires extra memory, and requires 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 one advantage, which is that the blocking is at a finer grain—at each location rather than across the data structure.

Gottlieb, Lubachevsky and Rudolph [8] also provide justification for why a parallel fetch-and-add operation can be implemented efficiently in hardware, and indeed, the Ultracomputer they subsequently built provided a parallel fetch-and-add. In addition to queues they showed how the fetch-and-add could be used for various other operations including a solution to the readers-writers problem, which allows a shared lock for the readers of a data structure, but an exclusive lock for the writers. Room synchronization can be viewed as extracting from these algorithms (and other previous algorithms using fetch-and-add) a synchronization construct useful for many problems.

In our earlier work on garbage collection we briefly described another weaker version of the room synchronizations [3]. That version had a couple important disadvantages over the version described in this paper. Firstly, it did not guarantee properties P5–P10, unless you assumed certain properties of the exact time taken by each instruction. In particular one process could be starved for an arbitrary amount of time if other processes kept going through the room very quickly (fast enough that the process missed doing a check of a flag for the brief time the room is open). This could even be true if all processors are running at the same rate. Secondly, the room synchronizations required that each processor once entering the first room had to also go through the second room. Thirdly, it only considered synchronization with two rooms. Finally, the conditions on what constitutes a room synchronization were not formalized. In our new implementation of the garbage collector [5], we use the implementation discussed in this paper.

As an abstraction, room synchronization is distinct from all previous abstractions known to us. It differs from mutual exclusion by introducing rooms that are mutually exclusive to each other, but allowing concurrent access to a given room. In more general abstractions, such as the resource allocation abstraction [16] of which the dining philosophers problem is a well-known specific instance, each user conflicts with a subset of the other users, and only nonconflicting sets of users can proceed in parallel. It is not clear how one would emulate rooms using this formalization.

Finally, there have been a number of papers describing techniques for reducing the contention in accessing shared data structures (e.g., [8, 19, 21]). These techniques are complementary to room synchronization, and perhaps can be exploited in the implementation of rooms by increasing the concurrency.

6. CONCLUSIONS

We presented a class of synchronizations called room synchronizations, that are likely to be useful in a 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 room synchronizations 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, the other processors can become blocked.

Based on room synchronizations we presented simple and efficient implementations of shared stacks and queues. To the best of our knowledge these are the first implementations of stacks and queues that are linearizable, handle asynchronous requests, and allow for constant-time access (assuming a constant-time fetch-and-add).

Acknowledgements

Thanks to Toshio Endo and the Yonezawa Laboratory for use of their 64-way Sun UltraEnterprise 10000.

7. REFERENCES

[1] 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 *Proc. 12th ACM Symp. on Parallel Algorithms and Architectures*, pages 137–146, July 2000.

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

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

[4] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronizations. Technical report, Carnegie Mellon University, Pittsburgh, PA, Apr. 2001.

[5] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proc. ACM SIGPLAN'01 Conf. on Programming Languages Design and Implementation*, June 2001.

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

[7] 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. Preliminary version appeared in SPAA'97.

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

[9] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, Palo Alto, CA, 1999. Tech. report STAN-CS-TR-99-1624.

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

[11] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 3(3):304–311, May 1992. Preliminary version in SPAA'91.

[12] M. P. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th International Symp. on Computer Architecture*, pages 289–300, May 1993.

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

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

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):435–455, Aug. 1974.

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

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

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

[19] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 54–63, June 1995.

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

[21] N. Shavit and A. Zemach. Combining funnels. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, pages 61–70, June-July 1998.

[22] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

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

[24] J. D. Valois. Implementing lock-free queues. In *Proc. 7th International Conf. on Parallel and Distributed Computing Systems*, pages 64–69, Oct. 1994.

[25] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1995.