Abstract

The Load-Link/Store-Conditional (LL/SC) primitive is considered the most suitable for implementing lock-free algorithms and data structures. However, the full semantics of LL/SC are not supported by any modern machine, so there has been a significant amount of work on simulations of LL/SC using Compare and Swap (CAS), a synchronization primitive that enjoys widespread hardware support. However, all of the algorithms so far that are constant time either use unbounded sequence numbers (and thus base objects of unbounded size), or require $\Omega(MP)$ space for $M$ LL/SC object (where $P$ is the number of processes).

We present a constant time implementation of $M$ LL/SC objects using only $\Theta(M + P^2)$ space and requiring only pointer-sized CAS objects. Our implementation can also be used to implement $L$-word LL/SC objects in $\Theta(L)$ time (for both LL and SC) and $\Theta((M + P^2)L)$ space. We focus on the setting where each process can have at most one LL/SC pair at a time. To support $k$ overlapping LL/SC pairs per process, our algorithms incur an extra factor of $k$ in their space usage.

To achieve these bounds, we begin by implementing a new primitive called Single-Writer Copy which takes a pointer to a word sized memory location and atomically copies its contents into another memory location. The only restriction is that the destination of the copy must be single-writer, which means that only one process is allowed to write/copy into it. We believe this primitive will be very useful in designing other concurrent algorithms as well.
can be arbitrary, but the location being written to has to be a special \textbf{Destination} object. A \textbf{Destination} object supports three operation, \texttt{read}, \texttt{write}, and \texttt{swcopy} and it allows any process to \texttt{read} from it, but only a single process to \texttt{write} or \texttt{swcopy} into it. We expect this primitive to be very useful in concurrent algorithms that use announcement arrays as it allows the algorithm to atomically read a memory location and announce the value that it read. We will see an example of this in Section 5.

In this work, we focus on lock-free and wait-free solutions. Roughly speaking, a \textit{lock-free} algorithm ensures that \textit{some} process is always making progress regardless of how the processes are scheduled. In particular, this means lock-free algorithms do not suffer problems such as deadlock and livelock. However, lock-freedom still allows processes to be starved from ever making progress, which motivates the definition of wait-freedom. Roughly speaking, \textit{wait-freedom} ensure that \textit{all} processes are making progress regardless of how they are scheduled. All algorithms in this paper take in $O(1)$ or $O(L)$ time (where $L$ is the number of words spanned by the implemented object), which is stronger than wait-freedom. The correctness condition we consider is \textit{linearizability}, which intuitively means that all operations appear to take effect at a single point.

In our results below, the time complexity of an operation is the number of instructions that it executes (both local and shared) and space complexity of an object is the number of words that it uses (both local and shared). There as been a significant amount of prior work on implementing LL/SC from CAS \cite{3,15,8,10,14} and we discuss them in more detail in Section 2. In this paper, we focus on the case where each process can have up to one outstanding LL/SC pair. The algorithms we present can also be extended to handle $k$ outstanding LL/SC pairs per process with an extra $k$ factor in the space complexity.

\textbf{Result 1 (Load-Link/Store-Conditional):} A collection of $M$ LL/SC objects operating on $L$-word values shared by $P$ processes can be implemented with:

1. $O(L)$ worst-case time for both LL and SC,
2. $O((M + P^2)L)$ space,
3. single word (pointer-width) read, write, CAS.

\textbf{Result 2 (Single-Writer Copy):} A collection of $M$ \textbf{Destination} objects shared by $P$ processes can be implemented with:

1. $O(1)$ worst-case time for read, write, and \texttt{swcopy}
2. $O(M + P^2)$ space
3. single word (pointer-width) read, write, CAS.

To help with implementing Single-Writer Copy, we implement a weaker version of LL/SC with the bounds below. Our version of weak LL/SC is even less restrictive than the weak LL/SC studied by prior work \cite{2,8,15}. We compare the two in more detail in Section 2.

\textbf{Result 3 (Weak Load-Link/Store Conditional):} A collection of $M$ weak LL/SC objects operating on $L$-word values shared by $P$ processes can be implemented with:

1. $O(L)$ worst-case time for both LL and SC,
2. $O((M + P^2)L)$ space,
3. single word (pointer-width) read, write, CAS.
Our implementations of \texttt{swcopy} and LL/SC are closely related. We begin in Section 3 by implementing a weaker version of LL/SC (Result 3). Then, in Section 4 we use this weaker LL/SC to implement \texttt{swcopy} (Result 2), and finally, in Section 5 we use \texttt{swcopy} to implement the full semantics of LL/SC (Result 1). As we shall see, once we have \texttt{swcopy}, our algorithm for regular LL/SC is almost the same as our algorithm for weak LL/SC.

2 Related Work

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Prior Work & Word Size (W) & Size of Implemented Object & Time & Space \\
\hline
Anderson and Moir [3], Figure 1 & pointer-width & $W - 2 \log P$ & $O(1)$ & $O(P^2 M)$ \\
Moir [15], Figure 4 & unbounded & $W - \text{tag\_size}$ & $O(1)$ & $O(P + M)$ \\
Moir [15], Figure 7 & $W \geq 3 \log P$ & $W - 3 \log P$ & $O(1)$ & $O(P^2 + PM)$ \\
Jayanti and Petrovic [8] & $W \geq 4 \log P$ & $W$ & $O(1)$ & $O(PM)$ \\
Michael [14] & unbounded & $LW$ & $O(L)$ & $O((P^2 + M)L)$ \\
Jayanti and Petrovic [10] & unbounded & $LW$ & $O(L)$ & $O((P^2 + M)L)$ \\
Jayanti and Petrovic [11] & unbounded & $W$ & $O(1)$ & $O(P^2 + PM)$ \\
Aghazadeh et al. [1] & $W \geq 2 \log M + 6 \log P$ & $LW$ & $O(L)$ & $O(MP^5 L)$ \\
Anderson and Moir [2], Figure 2 & pointer-width & $LW$ & $O(L)$ & $O(P^2 ML)$ \\
Jayanti and Petrovic [9] & pointer-width & $LW$ & $O(L)$ & $O(PML)$ \\
This Paper & pointer-width & $LW$ & $O(L)$ & $O((P^2 + M)L)$ \\
\hline
\end{tabular}
\caption{Cost of implementing \textit{M} LL/SC variables from CAS. Size is measured in number of bits.}
\end{table}

*uses unbounded sequence numbers

\textbf{LL/SC from CAS.} Results for implementing LL/SC from CAS are summarized in Table 1. The “Size of Implemented Object” column lists largest possible LL/SC object supported by each algorithm. For example, $W - 2 \log P$ means that the implemented LL/SC object can store at most $W - 2 \log P$ bits, and $LW$ means that the implemented object can be arbitrarily large. All the algorithm shown in the table are wait-free and have optimal time bounds.

So far, all previous algorithms suffer from one of three drawbacks. They either (1) are not wait-free constant time \cite{4, 7}, (2) use unbounded sequence numbers \cite{15, 14, 10, 11}, or (3) require $\Omega(MP)$ space \cite{3, 8, 12, 9, 15}. There are also some other desirable properties that an algorithm can satisfy. For example, the algorithms by Jayanti et al. \cite{11} and Doherty et al. \cite{4} do not require knowing the number of processes in the system. Also, some algorithms are capable of implementing multi-word LL/SC from single-word CAS, whereas others only work when LL/SC values are smaller than word size.

\textbf{Weak LL/SC from CAS.} A variant of \texttt{WeakLLSC} was introduced by \cite{2} and also studied in \cite{8, 15}. The version we consider is even less restrictive than theirs because they require a failed \texttt{wLL} operation to return the process id of the SC operation that caused it to fail whereas we don’t require failed \texttt{wLL} operations to return anything. While prior work is able to implement the stronger version of \texttt{wLL}, they either employ stronger primitives like LL/SC \cite{2}, use unbounded sequence numbers \cite{15}, require $O(MP)$ space for \textit{M} \texttt{WeakLLSC} objects \cite{2, 8}, or require storing $(4 \log P)$-bits in a single word \cite{8}. To match the bounds stated in Result 3 we
define and implement a version of weak LL/SC that is sufficient for our swcopy algorithm. Conveniently, the majority of our weak LL/SC algorithm from Section 3 can be reused when implementing full LL/SC in Section 5.

**Atomic Copy.** A similar primitive called memory-to-memory move was studied in Herlihy’s seminal Consensus Hierarchy paper [5]. The primitive allows atomic reads and writes to any memory location and supports a move instruction which atomically copies the value at one memory location into another. Herlihy showed that this primitive has consensus number infinity. Our swcopy is a little different because it allows arbitrary atomic operations (e.g. Fetch-and-Add, Compare-and-Swap, Write, etc) on the source memory location as long as the source object supports an atomic read. Another difference is that we restrict the destination of the copy to be single-writer.

## 3 Weak LL/SC from CAS

As a subroutine, our swcopy operation makes use of a weaker version of LL/SC. This weaker version supports two operations wLL and SC, and works the same way as regular LL/SC except that wLL is allowed to not return anything if the subsequent SC is guaranteed to fail. We call a wLL operation successful if it returns a value. Otherwise, we call it unsuccessful. Note that a wLL operation can only be unsuccessful if it is concurrent with a successful SC.

This is similar to the version of weak LL/SC defined by Anderson and Moir in [2] except that they require an unsuccessful wLL operation to also return the process id of the SC operation that caused it to be unsuccessful.

In Section 3.1, we present a constant time algorithm for Weak LL/SC.

### 3.1 Implementation of Weak LL/SC

In this section, we show how to implement $M$ WeakLLSC objects, each spanning $L$-words, in wait-free constant time and $O((M + P^2)L)$ space. The high level idea is to use a layer of indirection and use an algorithm similar to Hazard Pointers [13] to upper bound the memory usage. Each WeakLLSC object is represented using a pointer, $buf$, to an $L$-word buffer storing the current value of the object. To perform an SC, the process first allocates a new $L$-word buffer, writes the new value in it, and then tries to write a pointer to this buffer into $buf$ with a CAS. A wLL operation simply reads $buf$ and returns the value that it points to. The problem with this algorithm is that it uses an unbounded amount of space. Our goal is to recycle buffer objects so that we use at most $O(M + P^2)$ of them. This high level approach has been used in many previous algorithms [10, 12]. However, since we are only interested in implementing WeakLLSC, we are able to avoid using unbounded sequence numbers and provide better time/space complexities.

We recycle buffers with a variant of Hazard Pointers that is worst-case constant time rather than expected constant time. Before accessing a buffer, a wLL operation has to first protect it by writing its address to an announcement array. To make sure that it’s announcement happened “in time”, the wLL operation re-reads $buf$ and makes sure it is the same as what was announced. If $buf$ has changed, then the wLL operation can return empty because it must have been concurrent with a successful SC and it can linearize immediately before the linearization point of the SC. If $buf$ is equal to the announced pointer, then the buffer has been protected and the wLL operation can safely read from it.

For the purpose of the SC operation, each process maintains two lists of buffers: a free list (flist) and a retired list (rlist). In a SC operation, the process allocates by popping a buffer off its local free list. If the CAS instruction performed by the SC is successful, it adds the old value of the CAS to its retired list. Each process’s free list starts off with $2P$ buffers and we maintain the invariant that the free list and retired list always add up to $2P$ buffers. When the free list becomes empty and the retired list hits $2P$ buffers, the process moves some buffers from the retired list to the free list. To decide which buffers are safe to reuse, the process scans the announcement array (the scan doesn’t have to be atomic) and moves a buffer from the retired list to the free list
shared variables:
Buffer* A[P]; // announcement array

local variables:
list<Buffer*> flist;
list<Buffer*> rlist;
// initial size of flist is 2P
// rlist is initially empty

struct Buffer {
    // Member Variables
    Value[L] val;
    int pid;
    bool seen;
    void init(Value[L] initial_val) {
        copy initial_val into val
        buf_pid = -1; seen = 0;
    } 
}

struct WeakLLSC {
    // Member Variables
    Buffer* buf;
    // Constructor
    WeakLLSC(Value[L] initial_val) {
        buf = new Buffer();
        buf->init(initial_val);
    }

    optional<Value[L]> wLL() {
        Buffer* tmp = buf;
        A[pid].write(tmp);
        if (*X == tmp) return tmp->val;
        else return empty;
    }

    bool SC(Value[L] new_val) {
        Buffer* old = A[pid].read();
        Buffer* new = flist.pop();
        buf->init(new_val);
        bool succ = CAS(&buf, old, new);
        if (succ) retire(old);
        else flist.add(new);
        A[pid].write(NULL);
        return succ;
    }

    void retire(Buffer* old) {
        rlist.add(old);
        if (rlist.size() == 2*P) {
            list<Buffer*> reserved = [];
            for (int i = 0; i < P; i++)
                reserved.add(A[i].read());
            flist.add(rlist \ reserved);
            rlist = rlist \ reserved;
        }
    }
};

//Destructor
~WeakLLSC() { free(buf); }

optional<Value[L]> wLL() {
    Buffer* tmp = buf;
    A[pid].write(tmp);
    if (*X == tmp) return tmp->val;
    else return empty;
}

bool SC(Value[L] new_val) {
    Buffer* old = A[pid].read();
    Buffer* new = flist.pop();
    buf->init(new_val);
    bool succ = CAS(&buf, old, new);
    if (succ) retire(old);
    else flist.add(new);
    A[pid].write(NULL);
    return succ;
}

void retire(Buffer* old) {
    rlist.add(old);
    if (rlist.size() == 2*P) {
        list<Buffer*> reserved = [];
        for (int i = 0; i < P; i++)
            reserved.add(A[i].read());
        flist.add(rlist \ reserved);
        rlist = rlist \ reserved;
    }
};

Figure 1: Amortized constant time implementation of L-word Weak LL/SC from CAS. Code for process with id pid.

if it does not appear in the array. Since there are at most $P$ different buffers pointed to by the announcement array, the free list’s size is guaranteed to be at least $P$ after this step. In a later paragraph, we show how this step can be performed in worst-case $O(P)$ time, which amortizes over the number of free buffers found.

Pseudo-code is shown in Figure [1]. In the pseudo-code, we use $A[i].read$ and $A[i].write$ to read from and write to the announcement array $A$. Since each element of the announcement array is a pointer type, $read$ and $write$ are trivially implemented using the corresponding atomic instruction. We wrap these instructions in function calls so that the code can be reused in Section 5.1.

The operations $rlist \ reserved$ and $rlist \ reserved$ on lines 53 and 54 represent set difference and set intersection, respectively. What makes Hazard Pointers expected rather than worst-case constant time is that they use a hash table to perform these two steps. Instead, we add some space for meta-data in each $Buffer$ object so that it can store a process id, $pid$, and a bit, $seen$. To perform the set difference $rlist \ reserved$, the process first visits each buffer $B$ in $rlist$ and prepares the buffer by setting $B.pid$ to its own process id and setting $B.seen$ to false. Then, the process loops through the reserved and for each buffer, it sets $seen$ to true if $pid$ equals its own process id. Next, the process loops through $rlist$ again and constructs a list of buffers that have not been seen. This list is the result of the set intersection. Set intersection can be performed similarly. Finally, the process has to reset everything by setting $B.pid$ to $\perp$ for each $B$ in $rlist$.

So far, the algorithm we have described is only amortized constant time. To deamortize it, each process can maintain two sets of retired list and free lists. Each time the process pops from one free list, it performs a constant amount of work towards populating the other. The algorithm uses $P$ shared space for the announcement array, $O(P^2)$ local space for all the retired and free lists, and $M + O(P^2)$ shared space for all the buffers and
WeakLLSC objects. Therefore, it’s total space usage is $O((M + P^2)L)$. In addition, it only uses pointer-width read, write, CAS as atomic operations, so it fulfills the claims in Result [3].

4 Single-Writer Atomic Copy

The copy primitive, \texttt{swcopy}, can be used to atomically read a value from some source memory location and write it into a \texttt{Destination} object. It is similar to the memory-to-memory move primitive that was studied in [5], except that our \texttt{Destination} objects are single-writer and we allow the source memory location to be modified by any instruction (e.g. write, fetch-and-add, swap, CAS, etc). The sequential specifications of \texttt{swcopy} and \texttt{Destination} objects are given below.

\textbf{Definition 4.1.} A \texttt{Destination} object supports 3 operations \texttt{read}, \texttt{write} and \texttt{swcopy} with the following sequential specifications:

\begin{itemize}
  \item \texttt{read()}: returns the current value in the \texttt{Destination} object (initially $\perp$).
  \item \texttt{write(Value v)}: sets $v$ as the current value of the \texttt{Destination} object.
  \item \texttt{swcopy(Value* addr)}: reads the value pointed to by $addr$ and sets it as the current value of the \texttt{Destination} object.
\end{itemize}

Any number of processes can perform \texttt{read} operations, but only one process is allowed to \texttt{write} or \texttt{swcopy} into a particular \texttt{Destination} object.

We restricted this interface to be single-writer because it was sufficient for the use cases we consider. We find that single-writer \texttt{Destination} objects are very useful in announcement array based algorithms where it is beneficial for the read and the announcement to happen atomically. It’s possible to generalize this interface to support atomic copies that concurrently write to the same destination object. However, we are not sure what the desired behaviour should be in this case. One option would be to give atomic copy ‘store’ semantics where the value of the \texttt{Destination} object is determined by the last \texttt{write} or \texttt{copy} to that location. Another option would be to give atomic copy ‘CAS’ semantics where the \texttt{copy} is only successful if the \texttt{Destination} object stores the expected value. The right choice of definition will likely depend on the application.

In Section 4.1, we present our implementation of \texttt{swcopy}.

4.1 Algorithm for Single-Writer Atomic Copy

In this section, we show how to implement \texttt{Destination} objects that support \texttt{read}, \texttt{write}, and \texttt{swcopy} in $O(1)$ time and $O(M + P^2)$ space (where $M$ is the number of \texttt{Destination} objects).

We represent a \texttt{Destination} object $D$ internally using a triplet, $D.val$, $D.ptr$, and $D.old$. When there is no \texttt{swcopy} in progress, $D.val$ stores the current value of the \texttt{Destination} object. When there is a copy in progress, $D.ptr$ stores a pointer to the location that is being copied from. Finally, $D.old$ stores the previous value of the \texttt{Destination} object. The variables $D.val$ and $D.ptr$ are stored together in a WeakLLSC object (defined in Section 3). This allows us to read from and write to them atomically as well as prevent any potential ABA problems. The downside is that the only way to read $D.val$ or $D.ptr$ is through a \texttt{wLL} operation which can continue to fail due to concurrent SC operations. For this reason, we keep $D.old$ in a separate object, so that the readers can return $D.old$ if they fail too many times on \texttt{wLL}. Readers will only perform SC operations that change $D.ptr$ from not \texttt{NULL} to \texttt{NULL}. Therefore, the writer’s \texttt{wLL} will be successful whenever $D.ptr$ is \texttt{NULL}.

A \texttt{swcopy(Value* src)} on \texttt{Destination} object $D$ begins by backing up the current value from $D.val$ into $D.old$. At this point, $D.ptr$ is guaranteed to be \texttt{NULL}, so the writer can successfully read $D.val$ with a \texttt{wLL}. The \texttt{swcopy} proceeds by writing $src$ into $D.ptr$ with a SC. Finally, it reads the value $v$ pointed to by $src$ and tries to write $(v, \text{NULL})$ into $(D.val, D.ptr)$ with a SC. It’s not a problem if the SC fails because that means another process
struct Data {Value val; Value* ptr;};

struct Destination {
    // Member Variables
    WeakLLSC<Data> data;
    // data is initially (⊥, NULL)
    Value old;

    void swcopy(Value *src) {
        // This wLL() cannot fail
        old = data.wLL().val;
        data.SC((<empty>, src));
        Value val = *src;
        optional<Data> d = data.wLL();
        if (d != empty && d.ptr != NULL)
            data.SC((val, NULL));
    }

    void write(Value new_val) {
        data.wLL(); // cannot fail
        data.SC((new_val, NULL));
    }

    Value read() {
        optional<Data> d = data.wLL();
        if (d == empty) {
            d = data.wLL();
            if (d == empty) return old;
        }
        if (d.ptr == NULL)
            return d.val;
        value v = *(d.ptr);
        if (data.SC((val, NULL)))
            return v;
        d = data.wLL();
        if (d != empty && d.ptr == NULL)
            return d.val;
        return old;
    }
}

Figure 2: Atomic copy (single-writer). Code for process with id pid.

has helped complete the copy. This algorithm ensures that D.ptr is NULL as long as there is no ongoing swcopy operation. Furthermore, when D.ptr is NULL, it will ensure that D.val stores the current value of D.

To read from D, a process begins by trying to read the pair (D.val, D.ptr) with a wLL. If it fails on this wLL twice, then it is safe to return D.old because the value of D has been updated at least once during this read. Now we focus on the case were (D.val, D.ptr) is successfully read into local variables (val, ptr). If ptr is NULL, then val stores a consistent value, so the read returns it. If ptr is not NULL, then there is a concurrent swcopy operation and the read tries to help by reading the value v referenced by ptr and writing (v, NULL) into (D.val, D.ptr) with a SC. If the SC is successful, then the read returns v. Otherwise, the process performs one last wLL. If it is successful and sees that D.ptr is NULL, then it returns D.val. Otherwise, it is safe to return D.old.

The write operation is the most straightforward to implement. Since each Destination object only has a single writer, a write operation simply uses a wLL and a SC to store the new value into D.val. There cannot be any successful SC operations concurrent with the wLL because a non-writer processes can only succeed on a SC during a swcopy operation. Therefore, the wLL in write always succeeds.

In our algorithm, we assumed that values fit in a single word so that they can be atomically read from and written to. However, this assumption is not necessary. The algorithm can be generalized to work for larger objects as long as they support an atomic read operation.

Pseudo-code is showing in Figure 2. From the pseudo-code, we can see that each operation takes constant time. To implement M Destination objects, it uses M double-word-wide WeakLLSC objects and O(M) pointer-width read, write, CAS objects. Using the algorithm from Result 3 to implement the WeakLLSC objects, we get an overall space usage of O(M + P^2), which satisfies the conditions in Result 2.

5 LL/SC from CAS

Now we have all the tools we need to implement LL/SC (Result 1). Our algorithm is presented in Section 5.1.

5.1 Implementation of LL/SC from CAS

This algorithm is almost identical to our algorithm for weak LL/SC from CAS (Section 3.1). To ensure that the LL operation always succeeds, we use swcopy to atomically read and announce the current buffer (lines 32 and 33 of Figure 1). This means that the announcement array needs to be an array of Destination objects (from
shared variables:
Destination<Buffer*> A[P];

struct LLSC {
    Buffer* buf;
    ...
    value_t[L] LL() {
        A[pid].swcopy(buf);
        return A[pid].read()->val;
    }
    ...
};

Figure 3: Amortized constant time implementation of $L$-word LL/SC from CAS. The algorithm is exactly the same as Algorithm 1 except for the parts that are shown. Code for process with id $p_{id}$.

Section 4.1 rather than raw pointers. Other than that, the algorithm remains the same. Figure 5 shows the difference between this algorithm and the weak LL/SC algorithm from Figure 1.

This algorithm uses $O((M + P^2)L)$ pointer-width read, write, CAS objects just like in Figure 3 but it also uses $P$ Destination objects for the announcement array. From Result 2 we know that $P$ Destination objects can be implemented in constant time and $O(P^2)$ space, so this algorithm achieves the bounds in Result 1.

References

[1] Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings, pages 442–457, 2016.

[2] James H Anderson and Mark Moir. Universal constructions for large objects. In International Workshop on Distributed Algorithms, pages 168–182. Springer, 1995.

[3] James H Anderson and Mark Moir. Universal constructions for multi-object operations. In PODC, volume 95, pages 184–193. Citeseer, 1995.

[4] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pages 31–39. ACM, 2004.

[5] Maurice Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1):124–149, 1991.

[6] IBM. IBM System/370 Extended Architecture, Principles of Operation. Technical report, Publication No. SA22-7085. 1983.

[7] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, pages 151–160. ACM, 1994.

[8] Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of ll/sc variables. In Proceedings of the twenty-second annual symposium on Principles of distributed computing, pages 285–294. ACM, 2003.
[9] Prasad Jayanti and Srdjan Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In 25th IEEE International Conference on Distributed Computing Systems (ICDCS’05), pages 59–68. IEEE, 2005.

[10] Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of ll/sc objects. In International Conference On Principles Of Distributed Systems, pages 17–31. Springer, 2005.

[11] Prasad Jayanti and Srdjan Petrovic. Efficiently implementing ll/sc objects shared by an unknown number of processes. In International Workshop on Distributed Computing, pages 45–56. Springer, 2005.

[12] Maged M Michael. Aba prevention using single-word instructions. IBM Research Division, RC23089 (W0401-136), Tech. Rep, 2004.

[13] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems, 15(6):491–504, 2004.

[14] Maged M Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In International Symposium on Distributed Computing, pages 144–158. Springer, 2004.

[15] Mark Moir. Practical implementations of non-blocking synchronization primitives. In PODC, volume 97, pages 219–228. Citeseer, 1997.