Verification of a Lock-Free Implementation of Multiword LL/SC Object

Gao, Hui; Fu, Yan; Hesselink, Wim H.

Published in:
EPRINTS-BOOK-TITLE

2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing

Verification of a Lock-Free Implementation of Multiword LL/SC Object

Hui Gao, Yan Fu
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
Email: huigao@uestc.edu.cn, fuyan@ustec.edu.cn

Wim H. Hesselink
Dept. of Mathematics and Computing Science
University of Groningen
Groningen, The Netherlands
Email: w.h.hesselink@rug.nl

***Abstract*—On shared memory multiprocessors, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. By avoiding locks, the significant benefit of lock (or wait)-freedom for real-time systems is that the potentials for deadlock and priority inversion are avoided. The lock-free algorithms often require the use of special atomic processor primitives such as *CAS* (Compare And Swap) or *LL/SC* (Load Linked/Store Conditional). However, many machine architectures support either *CAS* or *LL/SC*, but not both. In this paper, we present a lock-free implementation of the ideal semantics of *LL/SC* using only pointer-size *CAS*, and show how to use refinement mapping to prove the correctness of the algorithm.**

I. Introduction

We are interested in designing efficient data structures and algorithms on shared-memory multiprocessors. On such machines, processes often need to coordinate with each other via shared data structures. In order to prevent the corruption of these concurrent objects, processes need a mechanism for synchronizing their access. The traditional approach is to explicitly synchronize access to shared data by different processes to ensure correct behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as long delays, convoying, priority inversion and deadlock. Using locks also involves a trade-off between coarse-grained locking which can significantly reduce opportunities for parallelism, and fine-grained locking which requires more careful design and is more prone to bugs.

Over the past two decades the research community has developed a body of knowledge concerning "Lock-Free" and "Wait-Free" algorithms and data structures. In contrast to algorithms that protect access to shared data with locks, lock-free and wait-free algorithms are specially designed to allow multiple threads to read and write shared data concurrently without corrupting it. The significant benefit of lock (or wait)-freedom for real-time systems is that by avoiding locks the potentials for deadlock and priority inversion are avoided.

It was shown in the 1980s that all algorithms can be implemented wait-free. However, the resulting performance does not in general match even naive blocking designs. It has also been shown [13] that the widely-available atomic conditional primitives, *CAS* and *LL/SC* cannot provide starvation-free implementations of many common data structures without memory costs growing linearly in the number of threads. Wait-free algorithms are therefore rare, both in research and in practice, and we are most interested in designing lock-free implementations.

A number of researchers[3], [5], [7], [9], [15] have proposed techniques for designing lock-free implementations. The lock-free algorithms often require the use of special atomic processor instructions such as *CAS* (compare and swap) or *LL/SC* (load linked/store conditional). However, Current mainstream architectures support either *CAS* or *LL/SC* with restricted semantics (but not both), which are susceptible to the *ABA* problem [14].

The ideal semantics of the atomic primitives *LL/SC* are inherently immune to *ABA* problem. However, for practical architectural reasons, no processor architecture supports the ideal semantics of *LL/SC*. Designing efficient algorithms to bridge the gap has been the subject of many researchers' interest. However, most of the research is focused on implementing only small LL/SC objects, whose value fits in a single machine [4], [8], [9], [11].

In this paper, using only pointer-size *CAS* we present a practical lock-free implementation of the ideal semantics of *LL/SC* Multiword objects (whose value does not have to fit in a single machine word) without causing ABA problem.

A true problem of lock-free algorithms is that they are hard to design correctly, even when apparently straightforward. To ensure our implementation is not flawed, we used the higher-order interactive theorem prover *PVS* [6] for mechanical support. All invariants as well as the simulation relation have been completely verified with *PVS*.

Overview. In Section 2 we present we present preliminary material which we require throughout this paper. In Section 3, we give a lock-free implementation of the ideal semantics of *LL/SC* Multiword objects. In Section 4, we provide an overview of the proof and a description of the role of the proof assistant PVS in it. In Section 5,we draw some conclusions.

978-0-7695-3929-4/09 $26.00 © 2009 IEEE
DOI 10.1109/DASC.2009.25

II. PRELIMINARY

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space in a heap. There can be several processes running on a single processor. Variables in shared context are visible to all processes running in associated parallel. Variables in private context are hidden from other processes.

We assume a universal set $\mathcal{V}$ of typed variables, which is called the *vocabulary*. A state s is a type-consistent interpretation of $\mathcal{V}$, mapping variables $v \in \mathcal{V}$ to values $s[\![v]\!]$. We denote by Σ the set of all states. If $\mathcal{C}$ is a command, we denote by $\mathcal{C}_p$ the transition $\mathcal{C}$ executed by process p, and $s[\![\mathcal{C}_p]\!]t$ indicates that in state s process p can do a step $\mathcal{C}$ that establishes state t. When discussing the effect of a transition $\mathcal{C}_p$ from state s to state t on a variable v, we abbreviate $s[\![v]\!]$ to v and $t[\![v]\!]$ to v'. We use the abbreviation $Pres(V)$ for $\bigwedge_{v \in V}(v' = v)$ to denote that all variables in the set V are preserved by the transition.

A. The Semantics of Synchronization Primitives

Traditional multiprocessor architectures have included hardware support only for low level synchronization primitives such as *CAS* and *LL/SC*, while high level synchronization primitives such as locks, barriers, and condition variables have to be implemented in software.

CAS atomically compares the contents of a location with a value and, if they match, stores a new value at the location. The semantics of *CAS* is given by equivalent atomic statements below. We use angular brackets $\langle \ldots \rangle$ to indicate atomic execution of the enclosed specification command[1].

proc *CAS*(**ref** X : `Val`; **in** $old,\ new$: `Val`) : `Bool` =
$\langle$ **if** $X = old$ **then** $X := new$; **return** *true*
else return *false*; **fi** $\rangle$

LL and *SC* are a pair of instructions, closely related to the *CAS*, and together implement an atomic Read/Write cycle. Instruction *LL* first reads the content of a memory location, say X, and marks it as "reserved" (not "locked"). If no other processor changes the content of X in between, the subsequent *SC* operation of the same process succeeds and modifies the value stored; otherwise it fails. The semantics of LL and SC are given by equivalent atomic statements below, where *me* is the process identifier of the acting process.

proc *LL*(**in** X : `Val`) : `Val` =
$\langle$ $S.X := S.X \cup \{me\}$; **return** X; $\rangle$

proc *SC*(**ref** X : `Val`; **in** Y : `Val`) : `bool` =
$\langle$ **if** $me \in S.X$ **then**
$S.X := \emptyset$; $X := Y$; **return** *true*
else return *false*; **fi** $\rangle$

[1]Note that , this is allowed only in the specification of the algorithm.

B. Refinement mappings

In practice, the specification of systems is concerned rather with externally visible behavior than computational feasibility. We assume that all levels of specifications under consideration have the same observable state space Σ_0, and are interpreted by their observation functions $\Pi : \Sigma \to \Sigma_0$. Every specification can be modeled as a four-tuple $(\Sigma, \Pi, \Theta, \mathcal{N})$ where $(\Sigma, \Theta, \mathcal{N})$ is the *transition system* [2].

A *refinement mapping* from a lower-level specification $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c)$ to a higher-level specification $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a)$, written $\phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$, is a mapping $\phi : \Sigma_c \to \Sigma_a$ that satisfies:

1) ϕ preserves the externally visible state component: $\Pi_a \circ \phi = \Pi_c$.
2) ϕ is a *simulation*, denoted $\phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$:
   ① ϕ takes initial states into initial states: $\Theta_c \Rightarrow \Theta_a \circ \phi$.
   ② $\mathcal{N}_c$ is mapped by ϕ into a transition (possibly stuttering) allowed by $\mathcal{N}_a$:
   $\mathcal{Q} \wedge \mathcal{N}_c \ \Rightarrow\ \mathcal{N}_a \circ \phi$, where $\mathcal{Q}$ is an invariant of $\mathcal{S}_c$.

Below we need to exploit the fact that the simulation only quantifies over all reachable states of the lower-level system, not all states. We therefore explicitly allow an invariant $\mathcal{Q}$ in condition 2 ②. The following theorem is stated in [1].

Theorem 1 *If there exists a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, then $\mathcal{S}_c$ implements $\mathcal{S}_a$.*

Refinement mappings give us the ability to reduce an implementation by reducing its components in relative isolation, and then gluing the *reductions* together with the same structure as the implementation.

III. THE LOCK-FREE IMPLEMENTATION OF *LL/SC*

Let us assume there are P ($\geq$ 1) concurrently executing sequential processes. To distinguish private persistent variables of different processes, every persistent private variable name can be extended with the suffix "." + "*process identifier*". In particular, $pc.q$ is the program location of process q, it ranges over all defined integer labels.

The specification $\mathcal{S}_a$ of *LL/SC* can then be given as shown in Fig. 1. In the specification, we model the `Node` as an array of the N shared variables in the heap under consideration, which can be of any type (e.g. `Val`). The indices of the `Node` are the addresses (or the pointers) to shared variables. We can thus simply regard the shared variable X (under consideration) as a synonym of an index of the `Node`, and its value is stored in `Node[`x`]`. As before, the action enclosed by angular brackets $\langle \ldots \rangle$ is defined as atomic statement.

We now turn our attention to the lock-free implementation using only pointer-size *CAS*, which is given by the

Constant
$P =$ `number of processes`;
$N =$ `number of shared variables`;
Shared variable
`Node`: **array** $[1 \dots N]$ **of** `Val`;
`S`: **array** $[1 \dots N]$ **of** `Set`;
Private variable
pc: $\{a_1;\ a_2\}$;
me: `ProcID`;
proc LL(**in** $x : 1 \dots N$) : `Val` =
a_1: $\langle S[x] := S[x] \cup \{me\}$; **return** `Node`$[x]$; $\rangle$
proc SC(**in** $x : 1 \dots N;\ Y :$ `Val`) : `Bool` =
a_2: $\langle$**if** $me \in S[x]$ **then**
$S[x] := \emptyset;\ Node[x] := Y;$ **return** *true*
else return *false*; **fi** $\rangle$
Initial conditions
Θ_a: $\forall p$: $1 \dots P$: $pc.p = a_1 \vee pc.p = a_2$

Figure 1. The Specification $\mathcal{S}_a$ of *LL/SC*

algorithm shown in Fig. 2. This lock-free implementation is inspired by our previous work [12].

In the lock-free implementation, the shared variable `indir`$[x]$ acts as pointers to the shared node x under consideration(i.e., the shared variable), while `node`$[mp_p]$ is taken as a "private" node of process p though it is declared publicly: other processes can read it but cannot modify it.

IV. Correctness

In this section we will prove that the concrete system $\mathcal{S}_c$ implements the abstract system $\mathcal{S}_a$. Formally, like we did in [10], [14], we define

$\Sigma_a \triangleq$ (`Node`$[1 \dots N]$, `S`) $\times$ (pc, me, x, Y)^P
$\Sigma_c \triangleq$ (`Node`$[1 \dots K]$, `indir`$[1 \dots N]$,
`prot`$[1 \dots K]$) $\times$ (pc, x, Y, mp, m, $mybuf$)^P
$\Pi_a(\Sigma_a) \triangleq$ `Node`$[1 \dots N]$
$\Pi_c(\Sigma_c) \triangleq$ `node`[`indir`$[1 \dots N]$]
$\mathcal{N}_a \triangleq \mathcal{N}_{a_0} \vee \mathcal{N}_{a_1} \vee \mathcal{N}_{a_2}$
$\mathcal{N}_c \triangleq \bigvee_{10 \le i \le 34} \mathcal{N}_{c_i}$.

The transitions of the abstract system can be described: $\forall s, t : \Sigma_a,\ p : 1 \dots P$:

$s[\![(\mathcal{N}_{a_0})_p]\!]t \triangleq\ s = t$ (to allow stuttering)
$s[\![(\mathcal{N}_{a_1})_p]\!]t \triangleq\ pc.p = a_1\ \wedge\ pc'.p = a_2$
$\wedge$ `S'`$[x.p] =$ (`S`$[x.p]\ \cup me$)
$\wedge\ Pres(\mathcal{V} - \{pc.p,$ `S`$[x.p]\})$
$s[\![(\mathcal{N}_{a_2})_p]\!]t \triangleq\ pc.p = a_2\ \wedge\ pc'.p = a_1$
$\wedge$ (($me \in$ `S`$[x.p]\ \wedge$ `S'`$[x.p] = \emptyset\ \wedge$ `Node'`$[x.p] = Y$
$\wedge\ Pres(\mathcal{V} - \{pc.p,$ `Node`$[x.p],$ `S`$[x.p]\}))$
$\vee\ (me \notin$ `S`$[x.p]\ \wedge\ Pres(\mathcal{V} - \{pc.p\}))))$

The transitions of the concrete system can be described in the same way. Here we only provide the description of

Constant
$P =$ `number of processes`;
$N =$ `number of shared variables`;
$K = N + 2P$;
Shared variable
`Node`: **array** $[1 \dots K]$ **of** `Val`;
`indir`: **array** $[1 \dots N]$ **of** $1 \dots K$;
`prot`: **array** $[1 \dots K]$ **of** $0 \dots K$;
Private persistent variable
pc: $[c_{10} \dots c_{34}]$;
mp: $1 \dots K$;
proc LL(**in** $x : 1 \dots N$) : `Val` =
loop
c_{10}: $m :=$ `indir`$[x]$;
c_{12}: $mybuf :=$ `Node`$[m]$;
c_{14}: `prot[m]` $++$;
c_{16}: **if** $m =$ `indir`$[x]$ **then**
`return` $mybuf$;
else
c_{18}: `prot`$[m] --$;
fi;
end;
proc SC(**in** $x : 1 \dots N;\ Y :$ `Val`) : `Bool` =
c_{20}: `Node`$[mp] := Y$
loop
c_{22}: $m :=$ `indir`$[x]$;
c_{24}: **if** CAS(`indir`$[x],\ m,\ mp$) **then**
c_{26}: `prot[m]` $--$;
c_{28}: **if** `prot`$[m] = 1$ **then**
$mp := m$;
else
c_{30}: `prot`$[m] --$;
repeat
choose mp **from** $1 \dots K$
c_{32}: **until** CAS(`prot`$[mp],\ 0,\ 1$)
fi;
`return` *true*;
else
c_{34}: `prot`$[m] --$;
`return` *false*;
fi;
end.
Initial conditions
Θ_c: $(\forall p$: $1 \dots P$: $(pc.p = c_{10} \vee pc.p = c_{20})$
$\wedge\ mybuf_p = N{+}p)$
$\wedge\ (\forall i$: $1 \dots N$: `indir`$[i] = i)$
$\wedge\ (\forall i$: $1 \dots K$: `prot`$[i] = (i \le N{+}P\ ?\ 1 : 0))$

Figure 2. The Lock-free implementation $\mathcal{S}_c$ of *LL/SC*

concrete transitions c_{16}: $\forall s, t : \Sigma_c,\ p : 1 \dots P$:

$s[\![(\mathcal{N}_{c_{16}})_p]\!]t \triangleq\ pc.p = c_{16}$

$$\wedge\ ((m.p = \texttt{indir}[x.p]\ \wedge\ pc'.p = c_{20}) \vee (m.p \neq \texttt{indir}[x.p]\ \wedge\ pc'.p = c_{18}))$$
$$\wedge\ Pres(\mathcal{V} - \{pc.p\})$$

To prove that $\mathcal{S}_c$ implements $\mathcal{S}_a$, we define the state mapping ϕ: $\Sigma_c \rightarrow \Sigma_a$ by showing how each component of Σ_a is generated from components in Σ_c:

$$\forall i\colon 1 \ldots N\colon \texttt{Node}_a[i] = \texttt{Node}_c[\texttt{indir}_c[i]]$$
$$\forall i\colon 1 \ldots N\colon \texttt{S}_a[i] = \{p\colon 1 \ldots P | pc_c.p \notin \{c_{10}; c_{20}; c_{22}\} \wedge\ x_c.p = i\ \wedge\ m_c.p = \texttt{indir}_c[x_c.p]\}$$
$$\forall p\colon 1 \ldots P\colon pc_a.p = (pc_c.p \in [c_{10} \ldots c_{18}]\ ?\ a_1 :\ a_2)$$

where the subscript indicates the concrete or abstract system a variable belongs to, and the remaining variables in Σ_a are identical to the variables occurring in Σ_c.

A. Proving the invariants with PVS

When we started to investigate the algorithm, it soon became apparent that we could use PVS as a proof assistant. In PVS, we defined the state space in terms of the shared and private variables, like the following:

```
N, P: posnat
K: posnat = N+P*2
Process: TYPE = range[P]
Index: TYPE = range[N]
Val: TYPE
State : TYPE = [#
  % shared variables
     Node : [ range(K) -> Val ],
     indir : [ Index -> range(K) ],
     prot : [ range(K) -> nat ],
     ...
  % private variables:
     pc : [ Process -> nat ],
     mp : [ Process -> range(K) ],
     ...
  % local variables of procedures:
     m : [ Process -> range(K) ],
     x : [ Process -> Index ],
     mybuf : [ Process -> Val ],
     ...
#]
```

The code of Section 3 can be easily transformed into a transition system. For example, using s and t of type state and p of type Process, line c_{16} is represented by the definition:

```
step16(p, s, t): bool =
  pc(s)(p) = 16 AND
  if m(s)(p) = indir(s)(x(s)(p)) then
     t = s WITH [ (pc)(p) := 20 ]
  else
     t = s WITH [ (pc)(p) := 18 ]
```

Since our algorithm is concurrent, the $step$ is defined as the disjoint of all atomic actions.

```
% transition steps
step(p,s,t) : bool =
  step10(p,s,t) or step12(p,s,t) or ...
  step20(p,s,t) or step22(p,s,t) or ...
  ...
```

We then started to guess and prove several invariants as described in the next sections. This improved our understanding and our confidence in the correctness of the algorithm. Finding invariants in an algorithm one does not really understand requires a good intuition, but is mainly a lot of work. The notion of stability for an proposed invariant can be proved by their corresponding $Theorem$ or $Lemma$ in PVS like:

```
% Theorem about the stability of invariant I1
IV_I1: THEOREM
  forall (s,t : state, p : Process ) :
     step(p,s,t) AND I1(s) AND I4(s) AND I5(s)
     => I1(t)
```

To ensure that all proposed invarinats be proved stable, we construct a global invariant `INV` by conjoining all proposed invariants, and discharge a particular proof for its stability.

```
% global invariant
INV(s:state) : bool =
  I1(s) and I2(s) and I3(s) and ...
  ...

% Theorem about the stability of the global invariant
IV_INV: THEOREM
  forall (s,t : state, p : Process ) :
     step(p,s,t) AND INV(s) => INV(t)
```

After the stabilities of all proposed invarinats have been checked separately, we define $Init$ as an initial condition that must be satisfied by all proposed invariants.

```
% initial state
Init: { s : state |
     (forall (p: Process):
        pc(s)(p)=10 or pc(s)(p)=20) and
     (forall (i: Index):
        indir(s)(i)=i) and
     ...
     }


% The initial condition can be satisfied
IV_Init: THEOREM
  INV(Init)
```

The role of PVS was plain verification. We ourselves invented the invariants. In the more difficult proofs of

preservation of some invariants, we also had to guide the choices of case distinctions.

B. Invariants

We establish some invariants for the concrete system $\mathcal{S}_c$, that will aid us in proving the refinement.

$$\begin{array}{ll} \textit{I1:} & p \neq q \ \wedge \ \mathit{pc}.p \notin [c_{26} \ldots c_{32}] \ \wedge \ \mathit{pc}.q \notin [c_{26} \ldots c_{32}] \\ & \quad \Rightarrow \ \mathit{mp}.p \neq \mathit{mp}.q \\ \textit{I2:} & \mathit{pc}.p \notin [c_{26} \ldots c_{32}] \ \Rightarrow \ \mathtt{indir}[x] \neq \mathit{mp}.p \\ \textit{I3:} & x \neq y \ \Rightarrow \ \mathtt{indir}[x] \neq \mathtt{indir}[y] \end{array}$$

In the expression of invariants, free variables p and q range over $1 \ldots P$, and x and y range over $1 \ldots N$. Invariants $I1$ and $I2$ indicate that, for any process p, $\mathtt{node}[\mathit{mp}.p]$ can be treated as a "private" node of process p since only process p can modify that. Invariant $I3$ implies that all shared nodes are different. To prove the invariance of *I1* to *I3*, we postulate

$$\begin{array}{ll} \textit{I4:} & \forall i\colon 1 \ldots K\colon \mathtt{prot}[i] = \sharp(\{x\colon 1 \ldots N \mid \mathtt{indir}[x] = i\}) \\ & \quad +\sharp(\{p \mid (\mathit{pc}_p \notin [c_{26} \ldots c_{32}] \ \wedge \ \mathit{mp}_p = i) \\ & \qquad \vee \ (\mathit{pc}_p = c_{26} \ \wedge \ m_p = i)\}) \\ & \quad +\sharp(\{p \mid \mathit{pc}_p \in [c_{16} \ldots c_{34}] \ \wedge \ \mathit{pc}.p \neq c_{32} \ \wedge \ m_p = i\}) \\ \textit{I5:} & \mathit{pc}.p \in [c_{20} \ldots c_{34}] \ \wedge \ \mathit{pc}.p \neq c_{32} \ \wedge \ \mathit{mp}.q = m.p \\ & \quad \Rightarrow \ \mathit{pc}.q \in [c_{26} \ldots c_{32}] \end{array}$$

Invariant $I4$ precisely describe the counter $\mathtt{prot}[i]$ for each $i \in 1 \ldots K$. Invariant $I5$ implies that process p cannot read the "private" node of other process q.

Consequently, we have the main reduction theorem for the lock-free implementation using *CAS*:

Theorem 2 *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is implemented by the concrete system $\mathcal{S}_c$ defined in Fig. 2, that is, $\exists \phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$.*

V. Conclusion

We are interested in designing efficient data structures and algorithms on shared-memory multiprocessors. On such machines, lock-free algorithms offer significant reliability and performance advantages over conventional lock-based implementations. The lock-free algorithms often require the use of special atomic processor primitives such as *CAS* or *LL/SC*. However, many machine architectures support either *CAS* or *LL/SC* with restricted semantics.

In this paper, we first present a lock-free implementation of the ideal semantics of Multiword *LL/SC* object using only pointer-size *CAS* without causing ABA problem. Then to ensure our algorithm is not flawed, we use refinement mapping to prove the correctness of the algorithm, and the higher-order interactive theorem prover PVS for mechanical support.

Acknowledgment

The Project Sponsored by the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Theoretical Computer Science, 2(82), 1991.

[2] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer Verlag, 1992.

[3] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. 15(5):745-770, November 1993

[4] A. Israeli and L. Rappoport. Disjoint-Access-Parallel implementations of strong shared-memory primitives. In Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, pages 151-160, August 1994.

[5] J. D. Valois. Implementing lock-free queues. In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pages 64C69, Las Vegas, NV, 1994.

[6] F. Cassez, C. Jard, B. Rozoy, M. Dermot (Eds.): Modeling and Verification of Parallel Processes. 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000.

[7] M.P. Herlihy and V. Luchangco and M. Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structure. Proceedings of 16th International Symposium on Distributed Computing, pages 339-353. Springer-Verlag, October 2002.

[8] P. Jayanti and S. Petrovic. Efficient and practical constructionsof ll/sc variables. In Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, pages 285-294, July 2003.

[9] V. Luchangco and M. Moir and N. Shavit. Nonblocking k-compare-single-swap. Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, pages 314-323. ACM Press, 2003

[10] H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. Proceedings of the 16th Conference on Computer Aided Verification (CAV), July 2004.

[11] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In Proceedings of the 23rd annual ACM symposium on Principles of distributed computing, pages 31-39, July 2004.

[12] H. Gao, J.F. Groote and W.H. Hesselink. Lock-free Dynamic Hash Tables with Open Addressing. Distributed Computing 17 (2005) 21-42.

[13] R. Bencina. Survey "Some Notes on Lock-Free and Wait-Free Algorithms" at www.audiomulch.com/ rossb/code/lockfree/.

[14] H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. Information and Computation 205 (2007) 225-241.

[15] H. Gao, J.F. Groote and W.H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. Science of Computer Programming 64 (2007) 341-374.