A Hardware Implementation of the MCAS Synchronization Primitive

Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, Smruti R. Sarangi
Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India.
emails: srishtypatel13@gmail.com, rajshekar.kalayappan@gmail.com, ishanimahajan7@gmail.com, srsarangi@cse.iitd.ac.in

***Abstract*—Lock-based parallel programs are easy to write. However, they are inherently slow as the synchronization is blocking in nature. Non-blocking lock-free programs, which use atomic instructions such as compare-and-set (CAS), are significantly faster. However, lock-free programs are notoriously difficult to design and debug. This can be greatly eased if the primitives work on multiple memory locations instead of one. We propose MCAS, a hardware implementation of a multi-word compare-and-set primitive. Ease of programming aside, MCAS-based programs are 13.8X and 4X faster on an average than lock-based and traditional lock-free programs respectively. The area overhead, in a 32-core 400mm^2 chip, is a mere 0.046%.**

I. Introduction

Lock-based parallel programs are easy to design, debug and reason about in terms of performance. The fastest possible implementations of parallel programs however are typically non-blocking ones where the algorithms do not use locks. They instead rely on built-in hardware synchronization primitives that allow conditional updates such as atomic compare-and-set (CAS). Such non-blocking algorithms have a rich literature and are typically several orders of magnitude faster than comparable implementations that use locks especially if there is high contention. They have also been shown to be much faster than implementations with transactional memory[1].

Sadly, designing and debugging such algorithms, and guaranteeing their performance, are fairly hard and complicated, even for experienced programmers. Consider the example of a queue. Below is a lock-free implementation of the enqueue operation implemented using only CAS instructions (as in the popular Boost C++ library).

```
while( true ) {
    last = tail ;          // ' tail ' is the queue's tail
    next = last→next;
    if ( last == tail )
        if (next == NULL)
            if (CAS( &(last→next), next, newNode) == next) {
                CAS( &tail, last , newNode);
                return ;
            }
        else
            CAS( &tail, last , next);
}
```

As can be seen, atomic updates to a shared data structure is quite intricate. A myriad of bugs may be introduced, whose manifestations may not be deterministic. The problem is even harder for more complicated structures such as doubly-linked lists and trees. Greenwald et al. [2] and Doherty et al. [3] observed that if we provide a single instruction called *MCAS* in hardware, the job of writing such non-blocking programs will become much easier. An *MCAS* instruction is a generalization of the CAS instruction. It compares the contents of k variables with k memory locations (pairwise), and if all the pairs match, then it atomically overwrites the k memory locations with k new values. We will refer to k as the arity of the MCAS instruction. Returning to our example, implementing the enqueue operation with MCASs is much simpler.

```
do {
    last = tail ;
    next = last→next;
    result = MCAS( 2, &(last→next), &tail, next, last , newNode,
          newNode); // first argument (=2) is the arity
} while( result == false );
```

We present implementations of six commonly used data structures using all three approaches: lock-based, CAS-based lock-free and MCAS-based lock-free, in the online appendix [4]. The MCAS-based approach reduces the lines of code by 51% on average as compared to the CAS-based one. Lines of code is a popular metric to quantify the ease of programming, debugging and maintaining software.

Although there have been works providing hardware support for the barrier synchronization primitive [5] [6] [7], to the best of our knowledge, there has been no work done to provide the implementation of an MCAS synchronization primitive in hardware. This paper proposes one such implementation, which needs minimal changes in the toolchain (compiler + linker), and has a hardware overhead of a mere 0.046%. We show improvements of 13.8X and 4X over lock-based and CAS-based lock-free implementations respectively.

II. MCAS Primitive Implementation: MCAS-BASE

The MCAS primitive is realized through *two-phase locking*. We first acquire *locks* on *all* concerned memory locations, and then perform the comparisons. If the values at *all* the locations are equal to the recorded old values, we update *all* the locations to the new values. We set the zero-flag (ZF) to 1 to indicate that the MCAS succeeded. If any of the comparisons fail, we set ZF to 0 to indicate failure. We then release *all* the locks.

A. ISA Augmentation

The programmer's interface for using the MCAS primitive consists of two instructions: (i) `MTS` and (ii) `MCAS`.

`MTS` (MCAS Table Store) instructions are used to set up the parameters of the MCAS primitive, one instruction per

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

memory location. The `MTS` instruction allows the programmer to specify the address of the memory location, the recorded old value against which the comparison must be made, and the new value to be written if the comparison evaluates to equal.

An `MCAS` instruction is used to direct the hardware to perform the MCAS primitive, based on the parameters already set up using `MTS` instructions. If all the comparisons evaluate to equal, the writes are performed and the *zero-flag* is set to 1; else it is set to 0. The `MCAS` instruction is also interpreted as a memory fence instruction in the pipeline. Therefore, there can be at most one active `MCAS` instruction per core.

B. Hardware Augmentation

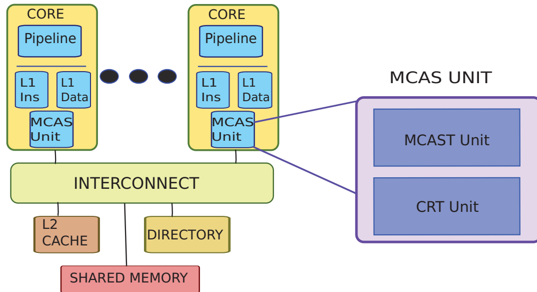


Fig. 1: Hardware Architecture

The additional hardware required is one *MCAS unit* per core (see Figure 1). Each MCAS unit consists of two sub-units: an MCAS Table (MCAST) Unit (Figure 2a) and a Cache Line Request Table (CRT) Unit (Figure 2b).

The MCAS Table stores the parameters of the MCAS instruction (note that a core can have at most one outstanding MCAS instruction). Each entry in the table corresponds to one memory location associated with the MCAS primitive. An entry in the table consists of the location's address, the recorded old value of the word, and the new value to be written. We assume an arity of 4 to draw area estimates of our proposed units, as arities > 4 are rarely useful. Our proposed protocols process the entries in the MCAS Table in ascending order of the address field (see Section II-C1) to avoid deadlocks between threads. To achieve a sorted access order, we store the indices of the entries in an auxiliary table called "Sorted Indices". These indices are maintained in the sorted order of the address fields of the MCAS Table entries they point to. The MCAST Unit additionally contains three registers: "Number of Addresses Added" (NAA), "Address to Lock Next" (ALN) and "Address to Write Next" (AWN).

Each entry in the CRT is composed of a cache line address (64 bit) and its corresponding *Cache Line Request List*. This is a list of cores that have requested and are waiting for this cache line (detailed in Section II-C2). Each entry in this list has two fields: (i) core ID (5 bit) and (ii) request type (read or write) (1 bit). The maximum number of entries in the list is 31, assuming a 32-core processor. If a core is to process a miss in its private caches, it first checks if there is space in its CRT. If there is no space, it stalls until there is. In our experiments, we observed that with a table size of 8, operations on standard data structures suffered no stalls.

C. Operation

1) Execution of an `MCAS` *Instruction:* The execution of MCAS instructions consists of four phases: (i) Setup, (ii) Acquire Lock and Compare, (iii) Write and (iv) Exit.

Setup Phase: On encountering an `MTS`, the pipeline signals the MCAST Unit to make an entry (address, old value, new value) in the MCAS Table. The NAA maintains how many entries are currently in the MCAS Table, and the new entry is made accordingly. The index of the new entry is inserted in the Sorted Indices Table at the appropriate position.

```
for nextToLock = 0; nextToLock <MCAS_Table.length; nextToLock++
do
    line = getCacheLine(memoryLocation);
    if cache miss then
        wait for the cache line to be loaded;
    end
    lock(line);
    if state(line) == Shared OR state(line) == Exclusive then
        state(line) = Modified;
    end
    if *memory_location!=old_value then
        ZF=0; // MCAS fail
        go to MCAS Exit phase;
    end
end
go to MCAS Write phase; //All memory locations have been locked
```

Algorithm 1: Acquire Lock and Compare

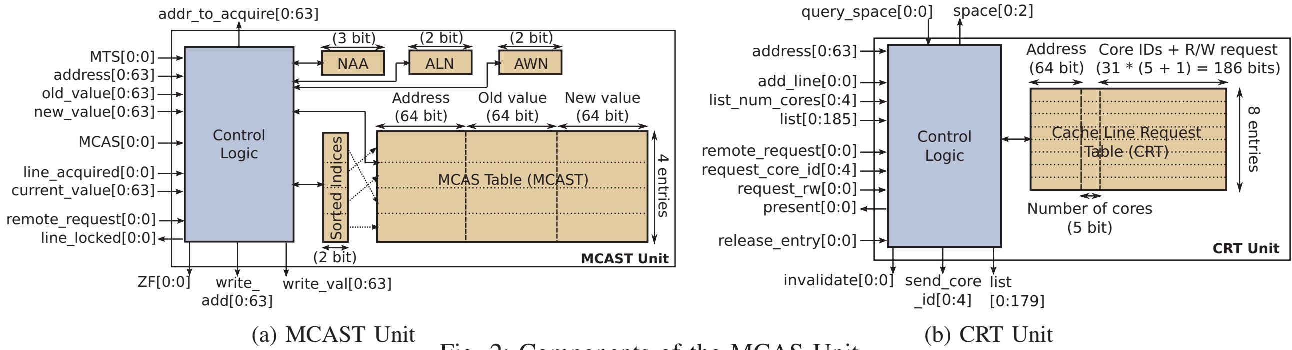
Acquire Lock and Compare Phase: Algorithm 1 describes the protocol. $nextToLock$ is maintained in the ALN register.

Write Phase: This phase is executed only when the MCAS is a success – that is, all entries in the MCAS Table have their lines locked. The MCAST Unit issues write requests to the cache to write the new values to the memory locations. The AWN maintains which line has to be written to next. ZF is set to 1. The exit phase is executed after all the writes.

Exit Phase: The entries of the MCAS Table are now processed in descending order of the addresses. If there is an entry in the CRT related to this address, it is processed (see Section II-C2). Once all the MCAS Table entries are processed, the MCAS Table contents are purged. The MCAS Unit signals the pipeline that the MCAS has *completed*, so that subsequent memory operations can be issued.

2) Staggered Cache Line Forwarding: When a cache line has been locked by a core, it possesses the cache line in the modified state. The cache coherence protocol ensures that no other core has this line. Now it must also be ensured that this locked line is not evicted until the MCAS is completed. For this, whenever a cache receives a request from another private cache to forward a cache line, the former checks if (i) that line is locked (by querying the MCAST Unit – the line is locked if the requested address is present in the MCAS Table and the position of its index in the Sorted Indices Table is less than ALN), or (ii) an entry corresponding to that line is present in the CRT. If either of these are true, then the cache line is not sent to the requester. Instead, this request is added to the end of the Cache Line Request List corresponding to that address.

The processing of the list is done on any of the three events: an MCAS, a read, or a write associated with the address completes. The processing of the list is as follows. If the list contains any write request, then the line in the cache



(a) MCAST Unit (b) CRT Unit

Fig. 2: Components of the MCAS Unit

is invalidated. Otherwise, the line in the cache is marked as *shared*. Next, the line is forwarded to the first core in the list. Along with the line, the rest of the list (having removed the first entry) is also sent. The receiving core adds the line to its cache and the list to its CRT. Since the list is forwarded along with the line, the latter arrives at all requesters in a first-come-first-serve order. *Starvation does not occur*.

The designs of the MCAST Unit and the CRT Unit were implemented in VHDL and synthesized using the Cadence Encounter RTL compiler and the 65nm technology standard cell library. Due scaling to 14nm was performed according to [8]. The area overhead for a 32-core, 400 mm^2 chip is 0.0456%. The units can operate at 3.4 GHz, while (i) servicing MTS requests, (ii) issuing lock acquire requests, (iii) issuing cache write requests, and (iv) processing Cache Line Request Lists, all at the rate of 1 per cycle.

III. Optimized Implementation: MCAS-OPT

In this section, we improve upon our base implementation MCAS-BASE with an optimized implementation MCAS-OPT. In the former, the `MTS` instructions to populate the MCAS Table were executed just before the `MCAS` instruction. However, analyzing common concurrent data structures, we observed that the addresses of the MCAS-related memory locations are known much earlier in the program. Thus, at runtime, it is possible to make the MCAS Table entry much earlier. This is extremely useful because in the time interval between when the entry was made and the `MCAS` instruction, through tracking of the cache coherence messages, we can know if the concerned memory location was written to by some other core. If so, then it is highly likely that this MCAS will fail when executed. We can thus perform an *early back-off*.

A. Placement of `MTS` Instructions

We design a compiler pass that places each `MTS` instruction as soon as the corresponding address is unambiguously known. The proposed method is a Worklist algorithm for iterative forward data-flow analysis. Due to space constraints, we discuss the algorithm details in the online appendix [4].

B. Early Back-Off

Each MCAS Table is augmented with a 1-bit flag *MCAS Invalid Flag* (MIF). When a cache receives an *invalidate* message from the directory, it checks its MCAS Table to see if it has an entry corresponding to the same address. If so, it sets the MCAS Invalid Flag (MIF) to 1. Now, at any point in time when the Lock Acquire and Compare phase is in progress, if the MIF is found to be 1, the phase is aborted. The MCAS is deemed to be failed, and the Exit phase is executed.

TABLE I: Parameters of the simulated architecture

Parameter	Value	Parameter	Value
Number of cores	32	Frequency	3.4GHz
Microarchitecture	Intel Sandybridge based		
Shared L2			
Size	8 MB	Latency	32 cycles
Associativity	8	Block size	64 Bytes
Write mode	Write back		
NOC			
Topology	2-D Torus	Routing Algorithm	dyn X-Y
Flit size	32 Byte	Hop latency	2 cycles
Main memory		Latency	200 cycles
MCAS Table (one per core)		Entries	4
Cache Line Request Table (one per core)		Entries	8

The hardware overhead of MCAS-OPT is 0.0466%.

IV. Evaluation

We use the Java-based multicore architectural simulator Tejas [9] (cycle accurate and verified against hardware) for evaluating our proposals. Table I lists the processor configuration. The concurrent data structures used for our evaluation are: Binary Search Tree (BST), Sorted List (LIST), Doubly Linked List (DLL), HashSet (HASH), Queue and Stack. We do so because these data structures are widely used in many real-world applications, and hence provide credibility to the proposal. Secondly, each of these benchmarks requires MCASs of different arity. This helps us gain a better insight into the behavior of such a primitive. Each benchmark is composed of 32 threads. Each thread makes a total of 300 operations (empirically found to reach steady state) on the shared data structure – alternate insertions and deletions of random elements to the data structure. We use three versions of benchmarks (most optimal implementations known in literature [10] [11]): with locks, CAS-based lock-free, and MCAS-based lock-free (henceforth referred to as "Lock", "Lock-Free" and "MCAS" respectively). The benchmarks are implemented in C++, and compiled with GCC version 4.7.2, with the -std=c++0x option. The source code can be found at [4].

A. MCAS v/s Lock

The MCAS benchmarks (both MCAS-BASE and MCAS-OPT) are up to 30X (13.8X on average) faster than the Lock ones (see Figure 3). This is expected since Lock benchmarks are blocking. Each thread remains blocked for long periods while waiting to acquire the lock, as shown in Table II.

TABLE III: Instruction mix of the different benchmarks (sync instructions: lock, CAS (in Lock-Free), MCAS)

Benchmark	Total Instructions (relative to Lock)				Percentage of sync instructions			
	Lock	**Lock-Free**	**MCAS-BASE**	**MCAS-OPT**	**Lock**	**Lock-Free**	**MCAS-BASE**	**MCAS-OPT**
BST	1.00	2.90	0.98	0.98	0.29	0.22	0.34	0.34
DOUBLYLL	1.00	7.76	2.06	2.07	0.12	0.11	0.07	0.08
HASHSET	1.00	0.61	0.64	0.64	0.22	1.40	0.40	0.40
LIST	1.00	3.56	0.57	0.57	0.21	0.29	0.42	0.42
QUEUE	1.00	2.49	0.93	0.91	0.33	1.84	0.87	0.80
STACK	1.00	0.91	1.45	0.90	0.31	0.91	4.31	0.87

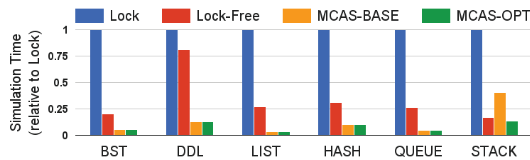


Fig. 3: Simulation Time Comparison

TABLE II: Average waiting time per lock (in cycles)

Benchmark	BST	DLL	HASH	LIST	QUEUE	STACK
Waiting Time	40.64	82.56	108.8	163.2	182.4	345.6

B. MCAS v/s Lock-Free

The MCAS benchmarks are up to 7X faster (4X on average) than the Lock-Free ones (see Figure 3). The reason for this is twofold. Firstly, multiple CAS instructions are required in the Lock-Free benchmarks for each operation (Table III compares the percentage of synchronization operations in the benchmark). This results in greater time spent in synchronization, leading to greater contention and lower CAS success rates (see Table IV). Secondly, Lock-Free benchmarks employ inter-thread helping to achieve progress and atomicity, thereby increasing the number of instructions executed (see Table III).

The Lock-Free STACK benchmark performs better than the MCAS-BASE one. The reason for this is twofold: (i) the Lock-Free benchmark does not employ any inter-thread helping (Table III shows that the total number of instructions are not that high), and (ii) the Lock-Free benchmark requires a single CAS instruction per push operation, while our encoding of the MCAS benchmark employs an MCAS of arity 2. Although unnecessary, we chose to do this simply to exercise the MCAS hardware. Increasing the arity increases the contention between threads, thereby reducing performance. MCAS-OPT, however, reduces this contention and achieves a speed-up of 22% over the Lock-Free benchmark.

C. MCAS-BASE v/s MCAS-OPT

MCAS-OPT is advantageous (up to 4.9X faster than MCAS base) when the contention between threads is high – that is, in the QUEUE and STACK benchmarks. In these benchmarks, the different threads are trying to operate on the same set of addresses (*top* pointer in STACK, *head* and *tail* pointers in QUEUE). Consequently, the MCAS-BASE success rate is low. The optimization reduces the contention, thus improving the MCAS-OPT success rate, as shown in Table IV. This reduces the number of MCAS instructions executed, as well as the time spent on each MCAS, as shown in Tables III and V. This results in a net increase in performance.

TABLE IV: Success rates of different primitives

Benchmark	BST	DLL	HASH	LIST	QUEUE	STACK
Lock-Free	0.91	0.29	0.25	0.42	0.1	0.37
MCAS-BASE	0.866	0.746	0.849	0.888	0.408	0.049
MCAS-OPT	0.857	0.742	0.853	0.881	0.448	0.394

TABLE V: Average execution time of MCAS variants

Benchmark	BST	DLL	HASH	LIST	QUEUE	STACK
MCAS-BASE	26.15	98.08	22.27	23.74	25.09	107.57
MCAS-OPT	21.89	132.62	31.44	23.54	17.77	22.74

V. Conclusion

The effort of designing and verifying efficient concurrent data structures can be greatly reduced if a hardware-provided, multi-word synchronization primitive is available. We have presented a detailed design of a multi-word compare-and-set primitive (MCAS), which has a minimal area overhead of 0.046%. We also show that MCAS-based concurrent data structures are 13.8X and 4X faster on an average than lock-based and lock-free implementations respectively.

References

[1] P. Aggarwal and S. R. Sarangi, "Software transactional memory friendly slot schedulers," in *ICDCIT*, 2014.

[2] M. Greenwald and D. Cheriton, "The synergy between non-blocking synchronization and operating system structure," *SIGOPS*, 1996.

[3] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr, "Dcas is not a silver bullet for nonblocking algorithm design," in *SPAA*, 2004.

[4] "A hardware implementation of the mcas synchronization primitive: Appendix." [Online]. Available: http://www.cse.iitd.ac.in/~srsarangi/tr/mcas_appendix.pdf

[5] S. Chandran, E. Peter, P. R. Panda, and S. R. Sarangi, "A generic implementation of barriers using optical interconnects," in *VLSID*, 2016.

[6] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *Micro*, 2006.

[7] J. Sartori and R. Kumar, "Low-overhead, high-speed multi-core barrier synchronization," in *HiPEAC*, 2010.

[8] W. Huang, K. Rajamani, M. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *Micro*, 2011.

[9] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*, 2015.

[10] P. Martin, "Practical lock-free doubly-linked list," uS Patent 7,533,138. [Online]. Available: http://www.google.co.in/patents/US7533138

[11] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.