IEICE Electronics Express, Vol.2, No.19, 488–494

An arbitration algorithm for multiport memory systems

Alex A. Aravind [a)]
Department of Computer Science,
University of Northern British Columbia,
Prince George, British Columbia, Canada
a) *csalex@unbc.ca*

Abstract: Multiport memories are increasingly used in smart-phones, multimode handsets, multiprocessor systems, network processors, graphics chips, and other high performance electronic devices [1, 2, 4, 8]. This paper presents a fully distributed software solution to the arbitration problem in multiport memory systems. Our solution is simple, efficient, and assures *LRU* fairness.

Keywords: multiport memory, arbitration, shared memory
Classification: Storage technology

References

[1] C. Springer, "Enabling Multimode Handsets," *EE Times*, Oct. 2004.
[2] L. E. Frenzel, "Dual-Port SRAM Accelerates Smart-Phone Development," *Electronic Design*, Feb. 2004.
[3] G. Taubenfeld, "The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms," *Lecture Notes in Computer Science*, vol. 3274, pp. 56–70, 2004.
[4] C.-W. Wang, K.-L. Cheng, C.-T. Huang, and C.-W. Wu, "Test and Diagnosis of Word-Oriented Multiport Memories," *Proc. of the 21st IEEE VLSI Test Symposium*, pp. 248–253, 2003.
[5] J. H. Anderson, Y.-J. Kim, and T. Herman, "Shared-memory Mutual Exclusion: Major Research Trends Since 1986," *Distributed Computing*, no. 16, pp. 75–110, 2003.
[6] J. H. Anderson and Y.-J. Kim, "Nonatomic Mutual Exclusion with Local Spinning," *Proc. of the ACM Sym. on PODC*, pp. 3–12, 2002.
[7] T. Raineault, "Semaphores Aid Multiprocessor Designs," *Embedded Edge*, pp. 14–20, Oct. 2001.
[8] R. Stodieck, "The IDT FourPort SRAM Facilitates Multiprocessor Design," *IDT Application Note An-43*, March 2000.
[9] M. Raynal, *Algorithms for Mutual Exclusion Problem*, The MIT Press, Cambridge, Massachusetts, 1986.
[10] L. Lamport, "The Mutual Exclusion Problem Part II: Statement and Solutions," *J. ACM*, vol. 33, no. 2, pp. 323–326, 1986.
[11] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Comm. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

1 Introduction

Multiport memories are increasingly used in smart-phones, multimode handsets, multiprocessor systems, network processors, graphics chips, and other high performance electronic devices [1, 2, 4, 8]. Multiport memory allows concurrent accesses to memory words through multiple ports.

Resolving conflicting accesses to shared memory by concurrent processors, also called memory arbitration or mutual exclusion, is a fundamental problem in concurrent computing. Distributed solution to this problem is generally complex and many arbitration algorithms are available for single port memory systems [9, 5]. The problem becomes more complex for multiport memory systems, due to the possibility of concurrent accesses to individual memory words through many ports. Among the solutions proposed for single port memory systems, the algorithms presented in [11, 10, 6] can be used to solve the arbitration problem in multiport memory systems. We briefly review them here.

Lamport's Bakery algorithm presented in [11] is the simplest and popularly known arbitration algorithm. It is based on the idea of using token numbers to resolve the conflict among the competing processors. This algorithm has a practical limitation that the token numbers can grow unboundedly, if always some processor is in need of using the shared memory. There are many recent attempts to bound the token values [3] and all these attempts require exclusive access to the shared variables (memory words). Therefore, they cannot be used in multiport memory systems. Another algorithm, by Lamport, with many nice properties is incrementally developed and presented in [10]. However, this algorithm is conceptually difficult for an average designer to implement. Recently, an algorithm for distributed shared memory systems - where the memory access is non-uniform - is presented in [6]. Also, after mentioning hardware based solutions such as hardware interrupt masking, hardware semaphores, and stalling processors with busy logic, a software based Master/Slave control protocol for a multiport memory systems is presented in [8].

Among the various criteria of the arbitration algorithms, the fairness property is very crucial one. It decides which processor among the competing processors is allowed to succeed next. First In First Out ($FIFO$) and Least Recently Used (LRU) are two important fairness criteria widely used in many applications. LRU favors the infrequent users of the resource compared to the frequent users. There are many algorithms available in the literature [9, 5] to assure $FIFO$. To the best of our knowledge, no arbitration algorithm is presented for shared memory system with LRU fairness criterion.

This paper presents a fully distributed software solution to the arbitration problem in multiport memory systems. Our algorithm is simple, assures LRU fairness, and applicable for multiport memory systems.

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

2 System Model and Problem Statement

We consider a multiport shared memory system of n-processors with ids $1, 2, \ldots, n$. The processors can simultaneously access the same memory location, may be through independent ports. The execution speed of any processor is finite but unpredictable.

We assume R as the memory segment that requires *mutually exclusive access* among the processors. The memory arbitration problem is to design an algorithm that assures the following properties: (i) *at any time, at most one processor is allowed to access R (safety property)* and (ii) *when one or more processors interested in accessing R, one of them eventually succeeds in accessing R (liveness property).* In addition to these two properties the following is a desirable property: (iii) *any processor interested in accessing R will be able to do so in finite time (freedom from starvation property).* We also assume that a processor will neither accesses R continously forever nor fails when it is accessing R.

The *code segment* that a competing processor executes can be divided into two parts: the part which accesses the shared memory R (*Critical Section (CS)*) and the remaining part (*Noncritical Section (NCS)*). A solution to the arbitration problem has essentially two components:*Entry Section* and *Exit Section*. These components has to be designed and inserted appropriately in the codes that all the competing processors execute to ensure consistent access to R.

3 The Least Recently Used (LRU) Algorithm

3.1 Idea

The basic idea behind the algorithm is very simple that, among the competing processors, the processor who accessed the CS "least recently" succeeds to access the CS next. The popular way to implement this idea is by using the timestamp (clock value) of each processor's latest access to the CS. Unfortunately, this approach requires unbounded size shared variables, similar to token variables in bakery algorithm, to hold the timestamp values. In this paper, we introduce a different approach to implement the LRU idea. Instead of timestamps, our algorithm uses the order of most recent CS accesses of the processors to choose the least recently CS accessed processor. The appeal of our approach is that it uses only bounded size shared variables and works for multiport memory systems.

3.2 Algorithm Design

We use an integer array called pos of size n to hold relative positions of the recent CS accesses (we refer as LRU positions) of the processors. The cell $pos[1]$ holds the id of "least recently" CS accessed processor, $pos[2]$ holds the id of next least recently CS accessed processor, etc., and $pos[n]$ holds the id of "most recently" CS accessed processor. That is, between any two processors p and q with respective LRU position values i and j, if $i > j$ then the processor q has higher priority than p in accessing the CS. Each

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

processor uses a local variable k to keep track of its LRU position in pos. A boolean array $competing$ of size n is used to indicate processors' interest in accessing the CS. Initially, for each p, $pos[p]$ is set to p, $competing[p]$ is set to $false$, and k is set to n. If k is not the current LRU position of a processor p, then it can find it by scanning the pos array downwards from location k as follows:

```
while($pos[k] \neq p$) $k := k - 1$;                    (a)
```

In the entry section, a *filter* mechanism is used to block the lower priority processors. A blocked processor, say p, can cross the filter only after all the higher priority processors complete their CS executions. The filter essentially has two components: (i) *checking for the competition of higher priority processors* and (ii) *waiting for the higher priority processors to complete their CS accesses and leave the competition.*

If there exists a higher priority processor q with LRU position less than k is competing for the CS, then p waits until q completes the CS access and leaves the competition. Since each higher priority processor changes the LRU positions of all the lower priority processors before it exits, a lower priority processor p can simply wait for its position change before it checking for any other higher priority processors. Thus, the filter for a processor p is designed as follows:

```
while($\exists j, (1 \leq j < k) \wedge (competing[pos[j]] = true)$)
    { wait until($pos[k] \neq p$); while($pos[k] \neq p$) $k := k - 1$; }
```

Next, we design a *safety-net* to assure exclusive access to the CS. Suppose a processor p starts its competition, determines that no higher priority processor is competing, and therefore proceeds further to enter the CS. Now, a higher priority processor q starts its competition and determines a lower priority processor p is competing. From q's point of view, p could be either blocked in the filter or in the CS (that is outside the filter). To avoid this dilemma, we introduce a boolean array called in_cs of size n to indicate whether the processors crossed the filter or not. The safety net will allow a processor to cross it only when in_cs of all other processors are $false$. ($in\_cs[p]$ is initialized to $false$, for all p). The safety-net for p is given next.

```
repeat {
    $in\_cs[p] := false$; $filter$; $in\_cs[p] := true$;
}until ($\forall j \neq p,\ in\_cs[j] = false$)
```

Finally, after completing the CS access, the processor p adjusts the LRU positions as follows:

```
while($pos[k] \neq p$) $k := k - 1$;
for($j := k$ to $n - 1$) $pos[j] := pos[j + 1]$;
$pos[n] := p$;                                           (c)
```

The LRU positions adjustment basically promotes the processors between location $k + 1$ and n by shifting one position left and places the id of p at location n. Since each position adjustment (by (c)) is done exclusively after every CS access completion, it is easy to see that it preserves the ids of all processors in pos. Also, it is obvious that a processor can find its id (by (a)) during no LRU position shift. Suppose p is reading at $pos[j]$ while q is writing

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

the value of $pos[j+1]$ on $pos[j]$ for a position shift, then it is easy to see that $pos[j+1]$ cannot be equal to p (otherwise, since p is scanning from right to left, it must have find its id at $pos[j+1]$ and therefore blocked at $pos[j+1]$ (in the filter). If the value of $pos[j]$ is p, before q writes on it, then the current value of $pos[j-1]$ must be p (due to shift). Therefore, even if p reads $pos[j]$ value as p during a shift and blocks at $pos[j]$, it will eventually read the correct value of $pos[j]$ and proceed further to identify its LRU position. That is, the possibility of inconsistent read due to multiport memory will not affect the correct reading of the LRU positions. The complete algorithm for the processor p is presented in Figure 1.

```
1.   $competing[p] := true; k := n;$
2.   while($pos[k] \neq p$) $k := k - 1;$
3.   repeat {
4.       $in\_cs[p] := false;$
5.       while($\exists j, (1 \leq j < k) \wedge (competing[pos[j]] = true)$)
6.           { wait until($pos[k] \neq p$); while($pos[k] \neq p$) $k := k - 1;$ }
7.       $in\_cs[p] := true;$
8.   }until ($\forall j \neq p,\ in\_cs[j] = false$)
9.   CS;
10.  while($pos[k] \neq p$) $k := k - 1;$
11.  for($j := k$ to $n - 1$) $pos[j] := pos[j + 1];$
12.  $pos[n] := p;$
13.  $competing[p] := false; in\_cs[p] := false;$
```

Fig. 1. *LRU* Algorithm

4 Correctness Proofs

First we introduce some terminology. We denote: (i) $r_i(x)$ and $w_i(x)$, respectively, as the *read* and *write* operations of processor i on the variable x; (ii) an event e_i *occurred before* another event e_j as $e_i \rightarrow e_j$. We say $e_i \rightarrow e_j$ as e_i *precedes* e_j in time; (iii) an event e_i *does not occurred before* another event e_j as $e_i \not\rightarrow e_j$; (iv) occurrences of two events e_1 and e_2 that *overlap* in time as $e_1 || e_2$; and (v) occurrences of two events e_1 and e_2 that *do not overlap* in time as $e_1 \not|| e_2$.

Theorem 4.1 *The LRU algorithm assures mutual exclusion.*

Proof : Proof by contradiction. Suppose there are more than one processors simultaneously accessing the CS. In particular, we consider two such processors i and j that access the CS at same time. Each of these two processors must have observed the other's in_cs value as $false$, to cross the line 8 before entering the CS. Let $w_i(in\_cs[i])$ and $r_i(in\_cs[j])$, respectively, be the latest write on $in\_cs[i]$ at line 7 and the latest read on $in\_cs[j]$ at line 8 by the processor i before it entered the CS. Similarly, let $w_j(in\_cs[j])$ and $r_j(in\_cs[i])$,

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

respectively, be the latest write on $in\_cs[j]$ at line 7 and the latest read on $in\_cs[i]$ at line 8 by the processor j before it entered the CS. The read and write operations of same processor cannot overlap in time. Therefore,
$w_i(in\_cs[i]) \rightarrow r_i(in\_cs[j])$ (1)
and
$w_j(in\_cs[j]) \rightarrow r_j(in\_cs[i])$ (2)

If $w_j(in\_cs[j]) \rightarrow r_i(in\_cs[j])$ then the processor i must have read the value of $in\_cs[j]$ as $true$ at line 8 before entering the CS, which is a contradiction. Similarly, if $w_i(in\_cs[i]) \rightarrow r_j(in\_cs[i])$ then the processor j must have read the value of $in\_cs[i]$ as $true$ at line 8 before entering the CS, which is also a contradiction.

Without loss of generality, assume that $w_j(in\_cs[j]) \not\rightarrow r_i(in\_cs[j])$ (3)
It is enough to prove that $w_i(in\_cs[i]) \rightarrow r_j(in\_cs[i])$. Relation (3) implies either $r_i(in\_cs[j]) \rightarrow w_j(in\_cs[j])$ or $r_i(in\_cs[j]) || w_j(in\_cs[j])$. If $r_i(in\_cs[j]) \rightarrow w_j(in\_cs[j])$, then by (1) and (2), we get $w_i(in\_cs[i]) \rightarrow r_j(in\_cs[i])$. If $r_i(in\_cs[j]) || w_j(in\_cs[j])$, then by (1), $w_j(in\_cs[j]) \not\rightarrow w_i(in\_cs[i])$. This means either $w_i(in\_cs[i]) \rightarrow w_j(in\_cs[j])$ or $w_j(in\_cs[j]) || w_i(in\_cs[i])$.

Case 1: $w_i(in\_cs[i]) \rightarrow w_j(in\_cs[j])$

$\Rightarrow w_i(in\_cs[i]) \rightarrow w_j(in\_cs[j]) \rightarrow r_j(in\_cs[i])$ (by (2))

$\Rightarrow w_i(in\_cs[i]) \rightarrow r_j(in\_cs[i])$

Case 2: $w_i(in\_cs[i]) || w_j(in\_cs[j])$

From the hypothesis, we get $w_j(in\_cs[j])$ overlaps with both $w_i(in\_cs[i])$ and $r_i(in\_cs[j])$. By (1) $w_i(in\_cs[i])$ must terminate before $w_j(in\_cs[j])$ terminates, for $r_i(in\_cs[j])$ to overlap with $w_j(in\_cs[j])$. This implies, and by (2), $w_i(in\_cs[i])$ must terminate before $r_j(in\_cs[i])$ starts. That is, $w_i(in\_cs[i]) \rightarrow r_j(in\_cs[i])$. Hence the proof.

Theorem 4.2 *The LRU algorithm assures liveness.*

Proof : If there is only one processor trying for its CS execution, then it can complete the lines 1 to 8 without blocking, and hence it can enter the CS in a finite time. If more than one processors are competing for the CS, then except the processor with the lowest *LRU* position, all other processors will be blocked at the filter in lines 5 and 6. This will eventually allow the processor with the lowest *LRU* position to cross the line 8 and enter the CS.

Theorem 4.3 *In LRU algorithm, the maximum number of overtakes possible over a processor to access the CS is* $n-1$.

Proof : A processor after completing its CS execution, shifts the *LRU* positions in lines 11 and 12 in such a way that it gets the lowest priority for the next turn. So, a processor cannot execute its CS consecutively when there are other processors currently competing for their CS executions. But, it is possible that a later processor can overtake an earlier arrived processor once due its current *LRU* position. Since the maximum number of processors in the system is n, the maximum number of overtakes possible over a processor is $n-1$.

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005

5 Conclusion

In this paper, after briefly reviewing the hardware and software solutions for the arbitration problem in multiport memory systems, we presented a new algorithm. Our algorithm is simple, efficient, and assures *LRU* fairness.

© IEICE 2005
DOI: 10.1587/elex.2.488
Received August 02, 2005
Accepted September 13, 2003
Published October 10, 2005