International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:12, 2009

A Technique for Execution of Written Values on Shared Variables

Parvinder S. Sandhu, Vijay K. Banga, Prateek Gupta, Amit Verma

Abstract**—**The current paper conceptualizes the technique of release consistency indispensable with the concept of synchronization that is user-defined. Programming model concreted with object and class is illustrated and demonstrated. The essence of the paper is phases, events and parallel computing execution .The technique by which the values are visible on shared variables is implemented. The second part of the paper consist of user defined high level synchronization primitives implementation and system architecture with memory protocols. There is a proposition of techniques which are core in deciding the validating and invalidating a stall page .

Keywords**—** synchronization objects, barrier, phases and events, shared memory

I. Introduction

RELEASE consistency[4] with user-definable high level synchronization primitives(RCHS) provides a pattern in which users can define their own synchronization primitives, called synchronization classes. RCHS also constrains the execution of synchronization primitives but any synchronization primitive that follows the paradigm can be used in this memory model. The paradigm provides

- Better interface for generating algorithms
- Improves the performance of subsequent applications.

Primitives (RCHS) are designed for a software distributed shared memory system in a high latency network, where the cost of traditional atomic operations, for example *fetch & add* and busy waiting for the purpose of synchronization is considerable, and has an adverse effect on the performance[6] of the computation.

Dr. Parvinder S. Sandhu is Professor with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 INDIA (Phone: +91-98555-32004; (Email: parvinder.sandhu@gmail.com).

Er. Prateek Gupta is Lecturer with Electronics & Communication Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 INDIA

Prof. Vijay K. Banga is Professor with Electronics & Communication Engineering Department, Amritsar College Of Engineering and Technology, Amritsar (Punjab)- INDIA

Er. Amit Verma is Assistant Professor with Electronics & Communication Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 INDIA

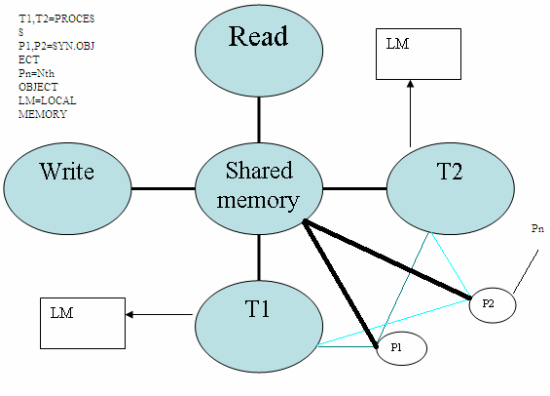

Fig. 1. Programming Model
(for two object can extended for n object)

There are two types of objects those communicate with in RCHS

- Shared memory
- Synchronization objects (see Fig. 1).

The shared memory consists of a pattern in memory as array (like) in virtual memory. Two types of operations are allowed in shared memory, read and write. Anything a process writes in shared memory may be visible to other processes. In addition to shared memory, processes can also communicate with each other via synchronization objects. Each process accesses synchronization objects only by calling operations defined in synchronization classes[6]. Synchronization objects can not access other shared objects. Each synchronization operation [7] may be annotated with one of following attributes, A: attributes(diagram 1)

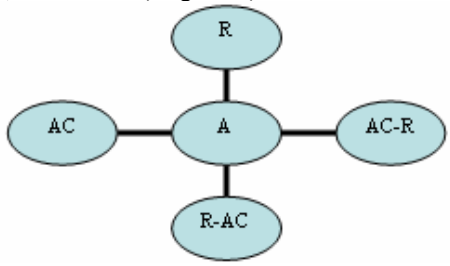

Fig. 2. Synchronization operation attributes

R: *release(*puts)
AC: *acquire(*gets)
AC-R: *acquire release*
R-AC: *release acquire*

These attributes are used to define the visibility of the values in shared memory. Informally, the annotation *release* may be thought of as meaning that the process “puts" its

visible shared memory writes onto the synchronization object, and AC may be thought of as meaning that the process "gets" the visible writes from the synchronization object. For example,

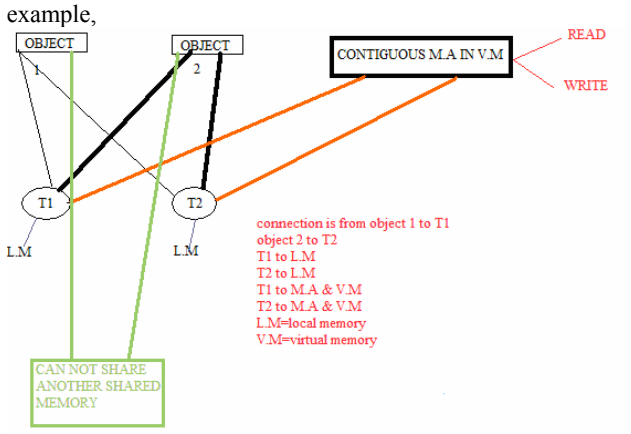

Fig. 3. Example with shared memory

Suppose that process *p* writes to shared variable *X* and then executes an operation provided by synchronization object *S* with attribute R. *S* can see the newly written value of *X* (figure 3,4). If process *q* subsequently executes an operation of *S* with attribute *acquire*, *q* obtains what *S* can see. So *q* can read the new value of *X* which *p* has written. Similarly, AC-R and R-AC may be thought of as a combination of the two. In the context of following discussion, *release operations* are operations with attribute *release*, *acquire release* or *release acquire*. *Acquire operations* are operations with attribute *acquire*, *release acquire*, or *acquire release*.

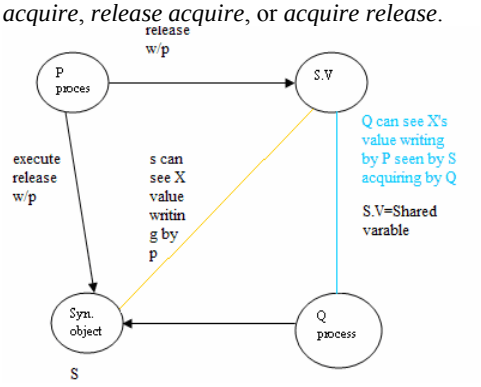

Fig. 4. Operation with processes

We consider an example of a simplified version of the producer and consumer problem [1]. Products produced by producers are stored in shared memory. Consumers read products from the shared memory. We assume there is infinite memory [3, 6] to store products. We use a synchronization object called buffer to coordinate producers and consumers. The definition of the synchronization object is shown in appendix 1.The synchronization object, buffer, works as a server which keeps pointers of available products for consumers. After a producer writes its product to shared memory, it calls the method PutItemPtr and passes the pointer of the product in shared memory to buffer. Since the product in shared memory needs to be visible to other consumers, PutItemPtr is annotated with attribute *release*. A consumer acquires the pointer of a ready product by calling GetItemPtr. In order to read the part of the shared memory written by producers, GetItemPtr is annotated with attribute *acquire*.

II. Synchronization Class

Our system provides two basic synchronization classes,

- Semaphores
- Barriers [3]

Semaphores have two operations, P(k) and V(k), where k is the number to increase or decrease the counter of the semaphore. The P& V operations of binary Semaphores usually correspond to operation on locks.

When a process, lets ∏ get lock by P, then the acquire notation specifies that previous lock halters writes become visible to the process(as read operation) Wait for barrier is a release acquire operation (attribute),calling process putting its writes (i.e making them visible)to the syn. object and will be getting other writes (i.e collecting currently visible writes) from the syn. object when the call returns.

III. Synchronization Classes And Synchronization Objects

Synchronization classes are similar to the classes in the C++ programming language. But

1) Start as syn.classes
2) Don't have inheritance property means can not put into other classes and retain the value

syn classes can be of two types these are

- Public
- Private

A. Public

The operation declared in public section can only be called by process. each operation can have only one parameter.

Public section toggles with syn. attribute these are R,A,RA,AR (*release*, *acquire*, *release acquire*, or *acquire release)* used to define the visibility of process write to share memory.

B. Private section

Used to define procedure and data structure in syn objects (act as server, servicing RPC)

IV. Execution Of Program

To specify the execution of a program, a directed acyclic graph (DAG) is used in the following discussion.

A. Phases & Events of a synchronization object

The synchronization operation is an event of a process, which in turn gets executed as a sequence of phases. There are two notable events associated with a synchronization object viz.

- Receiving a request from a process and
- Replying to a process

The computation between two events, including the ending event, is a *phase*. Two important phases associated with a synchronization object are as follows:

- *Receiving phase*: This phase of a synchronization object starts with a requesting event from a process, even though the requesting event is not in the phase.
- *Replying phase*: This phase of a synchronization object ends with a replying event.

Each phase of a process is identified by a unique time stamp. The time stamp of the initial phase is assumed with value 1 and the following phases attain a time stamp value incremented by one each time the next phase comes, i.e. time stamp gets values as say, a,a+1,a+2,.and so on.

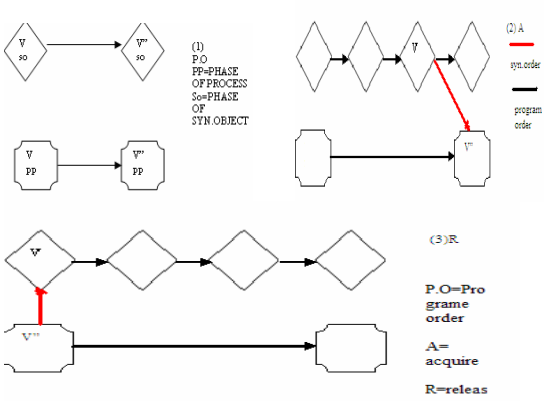

Fig. 5. Directed Acyclic Graph

The execution of a parallel program can be represented by a directed acyclic graph, $G = (V;E)$, where V is the set of phases. The visibility of a written value in shared memory is defined by G. There is an edge *evv0* from

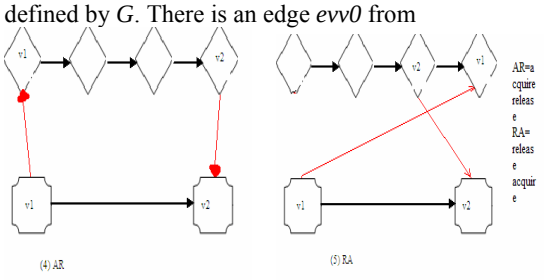

Fig. 6. Directed acyclic graph with computing (line notations are same as above fig)

vertex *v* to *v0(in above figure V1& V2)* if and only if one of following conditions holds.

1. *v0* and *v* are phases from the execution of the same process *v0* immediately follows *v* in program manner. See Fig. 5.
2. A process invokes a synchronization operation with the *A(A=acquire)* attribute. *v* is the answering phase of the syn object, and *v0* is the phase after the process invokes the synchronization operation. See Fig. 5.
3. A process invokes a synchronization operation with the R(R=release) attribute.
   v = phase ending
   v0= receiving phase (See Fig 5.)
4. A process invokes a synchronization operation with the RA(RA=*release acquire)* attribute. This case involves two edges.
(a) *v* = phase ending
   v0= receiving phase (See Fig 5.) (see edge *ev*1*v*01)
(b) *v* = replying phase
   v0 =phase after the process invokes operation (see edge *ev*2*v*02 in Fig 6.)
5. A process invokes a synchronization operation with the AR(AR=*acquire release)* attribute. In this case there are two edges:
(a) *v*= phase of the process ending
   v0=phase immediately after the replying phase (see edge *ev*1*v*01 in Fig 6).
(b) *v*= replying phase
   v0=phase after the process invokes the synchronization operation (see edge *ev*2*v*02 in Fig 6).

Where v,v0= synchronization operation,

A phase *Q0* is reachable from *Q*, denoted by *Q - Q0*, if there is a path from *Q* to *Q0*. It means Q reaches Q0 .Two phases are concurrent if there is no path between them. *Competing accesses* are two operations accessing the same shared variable in concurrent phases and one of the operations is write.

Assume two operations *o* and *o0* are executed in two phases, *Q* and *Q0* respectively. *o0* is reachable from *o* If any of following conditions is satisfied :
(1) *Q* = *Q0* and *o* is executed before *o0*,
(2) *Q - Q0*.

V. Visibility Of Written Values On Shared Variables

A written value of a write operation, *w*, on shared variable *x* in phase *Q* is visible to a read operation on *x* in phase *Q0* if and only if one of following situations holds

1. if *Q* = *Q0*, the write operation is the last write operation on *x* before the read operation.
2. if *Q* =! *Q0(not equal)* and *Q0* is reachable from *Q*, *w* is the last write on *x* in phase
Q and there is no other phase on the path from *Q* to *Q0* which has write operation on *x*.
3. *Q* and *Q0* are concurrent.

The set of written values visible to the read operation *op* on *X* is called the *visible set* of *X*.

From the programmer's point of view, an *acquire* action takes place when a synchronization object replies to a requesting process. All the updates in shared memory that are visible to the synchronization object before the synchronization object replies are also visible to the process after the process receives the response from the synchronization object. The time instance when R(R=*release)* acts depends on how conservatively the updates in shared memory are expected to be propagated. If the attribute RA(RA=*release acquire)* or R(R=*release)* is used, the updates of the requesting process are visible to the synchronization object when the object receives the request. If the operation is annotated with AR(AR=*acquire release)*, the updates are made visible to the synchronization object after the object replies to the process. The attribute AR(AR=*acquire release)* can be used for atomic updates to the synchronization object. For example, in the consumer producer problem , if the queue buffer in the synchronization object buffer is full, the producer needs to be suspended until some products are taken

by consumers. The updates in shared memory by the producer do not have to be seen by others until the products are stored in the queue buffer. The function PutItemPtr may be annotated with the attribute AR(AR=*acquire release)*. A synchronization operation can have no attribute. The updates of the process are not visible to such a synchronization object when it executes the synchronization operation

REFERENCES

[1] A. S. Tanenbaum. Modern Operating Systems, chapter 2. Pren-tice Hall, Vol.2,1992
[2] M. Dubois and C. Scheurich., "Memory access dependencies in shared-memory multiprocessors" IEEE Transaction on Software Engineering, June 1990.
[3] O. Babaoglu and K. Marzzullo "Distributed Systems", chapter 4.Addison-Wesley, second edition, 1993.
[4] J. B. Carter. "Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency". PhD thesis, Rice University, 1993.
[5] www.springer.com/computer/communications/book/978-0-387-21509-9 - 36k
[6] http://www.stormingmedia.us/62/6230/A623044.html
[7] Coulouris, George F.; Dollimore , "Distributed Systems: "Concepts and Design(international computer system series)"Second Edition Published by Addison-Wesley", 1994.
[8] C.George ,Jean Dollimore, Tim Kindberg "Distributed. Systems Concepts and Design" ,third edition, published August 7, 2000 672 pages