(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 September 2012 (20.09.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/124995 A2

(51) International Patent Classification:
G06F 9/06 (2006.01) *G06F 15/76* (2006.01)
G06F 9/44 (2006.01)

(21) International Application Number:
PCT/KR2012/001880

(22) International Filing Date:
15 March 2012 (15.03.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
793/CHE/2011 15 March 2011 (15.03.2011) IN

(71) Applicant *(for all designated States except US)*: **SAMSUNG ELECTRONICS CO., LTD.** [KR/KR]; 129, Samsung-ro, Yeongtong-gu, Suwon-si, Gyeonggi-do, 443-742 (KR).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **KOLIPAKA, Parikshit** [IN/IN]; Bagmane Lakeview, Block 'B', No. 66/1, Bagmane Tech Park, C V Raman Nagar, Byrasandra, Bangalore 560093 (IN). **NAGPAL, Rahul** [IN/IN]; Bagmane Lakeview, Block 'B', No. 66/1, Bagmane Tech Park, C V Raman Nagar, Byrasandra, Bangalore 560093 (IN).

(74) Agent: **LEE, Keon-Joo**; Mihwa Bldg., 110-2, Myongryun-dong 4-ga, Chongro-gu, Seoul 110-524 (KR).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

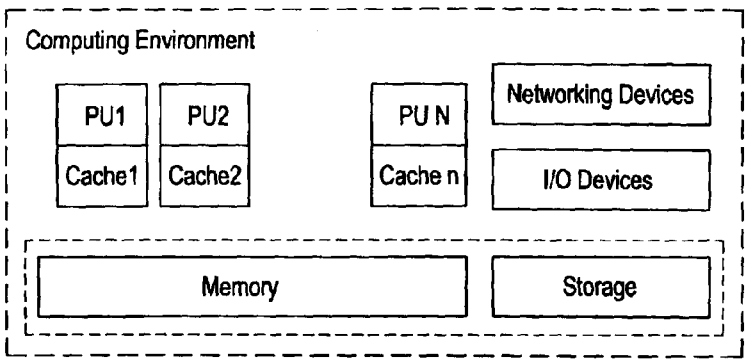
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) Title: METHOD AND SYSTEM FOR MAINTAINING VECTOR CLOCKS DURING SYNCHRONIZATION FOR DATA RACE DETECTION

[Fig. 9]



(57) Abstract: Method and system for maintaining vector clocks during synchronization for data race detection. Embodiments herein disclose methods to reduce overheads of maintaining and updating vector clock during synchronization in vector based dynamic data race detection systems. Embodiments herein enable improvement of vector based dynamic data race detection systems orthogonally without compromising with precision of the system by using opportunistic methods to reduce overheads during synchronization of threads.

WO 2012/124995 A2

Description

Title of Invention: METHOD AND SYSTEM FOR MAINTAINING VECTOR CLOCKS DURING SYNCHRONIZATION FOR DATA RACE DETECTION

Technical Field

[1] Embodiments herein relate to dynamic data race detection, and, more particularly, to reducing overheads in maintaining and updating vector clocks during synchronization.

Background Art

[2] Vector Clock based dynamic data race detector provides a general dynamic analysis framework based on vector clock mechanism for detecting data races in concurrent programs at run time more precisely (fewer false positive) than either static or any other dynamic approach, such as lock-set based approach. The major issue with the dynamic data race detector is space and time overheads of maintaining and updating vector clocks that is O(n) in general where n is number of threads. Increasing number of cores on chip and high degree of threading supported by cores and GPGPUS further exaggerate performance and space overheads associated with vector clocks.

[3] A data race condition occurs when two threads access same memory location at the same time without synchronization (or not ordered by happens before) and at least one of these memory access is a write access. Race conditions are inherently difficult to detect, reproduce and eliminate primarily because they occur rarely and only in certain rare executions and rare contexts. The major trade-offs between static and dynamic data race detectors is that of soundness vs. precisions. In contrast to static data race detector which do not actually run the program but never miss a data race if one exist in program (soundness), at the cost of being conservative and producing lots of false positive (less precise). But dynamic data race detectors actually run the program to gain in precision. Dynamic data race detector (DDRD) perform and scale better as compared to static race detectors, that bear the inherent curse of algorithmic overheads involved in deep program analysis. However, any overhead due to DDRD directly impacts the program execution time and not merely the compilation or analysis time as in static techniques. Improving the performance overheads of DDRD is further fueled by growing popularity of multi-core architecture and GPGPUs. Current state-of-the-art tools trade accuracy (preciseness) for the speed.

[4] The lockset based approach of DDRD is limited to detecting races in program that use most popular synchronization primitive i.e. locking discipline. The lockset based approach assumes that all the variables are guarded by all the locks in the beginning of program execution. As it processes the trace of the program, it iteratively refines the

locks associated with the shared variables and whenever it finds that a variable is not guarded by proper locks, it alarms a possible data race. This approach being limited to locking discipline leads to lots of false positives in the presence of other synchronization primitive such as fork-join, and wait-notify among others.

[5] The main idea behind a Purely Happens Before technique (PHB) is to monitor all the thread and their accesses to the shared memory locations in the current execution trace and deduce a partial order called happens before as imposed by the synchronization primitives. Formally, happens before relationship among program statements is defined as follows. Statement A happens before B (A < B) if any of the following is true: A executes before B in the same thread or A and B are operations on the same synchronization variable, between threads and events are ordered according to the properties of the synchronization objects they access (e.g., A releases a lock, and B subsequently acquires the same lock) or A < C and C < B then A < B, happens before is transitive. This partial order is used thereafter to find out the possibility of access to the same memory location by two different statement not related with happens before relation. If at least one of these is write, the race is detected.

[6] A specific mechanism to implement happens-before is vector clock. A vector clock is essentially defined as a mapping C: Tid -> N@id for all id $\in$ N where id represents the thread identification number. Some primitive operation on vector clocks is defined as follows: happens before relationship operation: $C_1 < C_2$ iff $C_1(t) < C_2(t)$ for each t $\in$ id, Join operation: $C_1 \sqcup C_2 = \max(C_1(t), C_2(t))$, for each t $\in$ id, Ov= 0. Oe=0@0, for each t $\in$ id (Ov is the minimal version epoch and Oe is minimal epoch) and INCt (C) = For all j $\in$ id if j == t then Ct(j)= Ct(j)+1 else Ct(j)= Ct(j).

[7] During the execution of the program, race detector need to maintain multiple vector clocks such as vector clocks to store the clock value of each of the thread by every other thread and storing last read and write by any thread for each shared variable among others. As the program executes these clocks are updated depending on the read and writes to shared variables of different threads and synchronization operation by different thread. Vector clocks, if not used efficiently leads to expensive O(n) operations and space overheads. Different tools vary in terms of reducing these overheads of updating vector clocks by summarizing vector clock information into a scalar thereby reducing O(n) operation to O(1) operation.

[8] DJIT+ essentially maintains following vector clocks: Firstly, each thread t keeps a vector clock Ct such that for any thread u, Ct(u) record the clock of the last operation of thread u that happens before the current operation of thread t. Clock of every thread

is incremented at each lock release operation. Secondly, Lm is a vector clock corresponding to each lock and when a thread u releases a lock m, DJIT+ algorithm updates Lm to Cu and if, the thread t subsequently acquires m then the algorithm updates Ct to be Ct ⊔ Lm. Finally, to identify conflicting access, DJIT+ algorithm keeps two vector clocks Rx and Wx that record the clock of thread that last read and write x from every thread respectively.

[9] Using these clocks DJIT+ determines a read access to x by thread u to be race free, if it happens after the last write by all the threads (Wx < Cu) and a write access to x by thread u is race free provided it happens after all access (read and write) to that variable (i.e Wx < Cu and Rx < Cu). Vector clocks are updated on synchronization operation that impose happens-before order between different threads. DJIT+ uses the full generality of the vector clocks thereby leads to overhead of O(n) in space as well as time where n is the number of threads.

[10] FastTrack is a vector clock based dynamic data race detector that provides same precision as DJIT+ but significantly improves the performance and space overhead of maintaining and updating multiple vector clocks. FastTrack works on the premises that the full generality of vector clocks as used in DJIT+is not required for detecting data races. Essentially, FastTrack switches effectively between vector and epoch (summary information from vector clock faded into a scalar) in order to reduce expensive O(n) operation to O(1) operations as much as possible in the quest of taming the overheads of maintaining full vector clock wherever possible for memory read and memory write operations.

[11] In order to reduce the overheads, FastTrack keeps the summary of a vector clock in the form of epoch. An epoch is denoted as c@t. An epoch c@t happens before a vector clock V iff c<=V(t). For each variable FastTrack maintains write epoch which essentially is the clock value of last thread that wrote x and for a read it adaptively switches between read epoch (clock value of last variable that read x) as well as completely general vector clock. A shrewd observation exploited by FastTrack to improve over DJIT+ is that just write epoch suffices instead of a write vector clock because writes to a variable are actually totally ordered. FastTrack also observes that in a race-free program, upon a write, all previous reads must happen before the write, so FastTrack adaptively switches from read epoch to read vector clock and from read vector clock to read epoch whenever necessary. For example, it switches from epochs to vector clocks, when it has to distinguish between multiple concurrent reads, since they all potentially race with a subsequent write. When reads are ordered by the happens-before relation, FastTrack uses an epoch for the last read otherwise, it uses a vector clock for reads.

[12] On each read access by a thread, FastTrack simply checks that the read happens after the last write by comparing with the Write epoch of the variable and this is a fast O(1) operation as compared to O(n) operation of DJIT+. On each write access by a thread, FastTrack first checks the conflicts with earlier write by comparison with the write epoch of x which is also O(n) operation and is not very expensive from performance or space point of view. However, in order to check the read-write kind of race FastTrack also compare with the read vector clock to detect, if there is a race with any of the reads happening before this write which is in general a relatively slow O(n) operation. However, FastTrack is able to reduce the overheads of keeping the full vector clock for fully ordered read such as thread local and lock protected data. In other cases, where reads are not completely ordered FastTrack adaptively switches to vector clock for read operations. Thus it indicates that, FastTrack reduces the general O(n) space and time overheads of vector clock such as in DJIT+ to O(1) for memory reads completely and memory writes partially. However, it does not make any attempt to reducing O(n) overheads of synchronization operation that happen say during acquire and release, and are on rise because of more number of threads in upcoming multi-cores and GPGPUs.

[13] FIG. 1 illustrates an example of functioning, FastTrack dynamic data race detector. Consider that there are 3 threads executing concurrently and accessing the variables x and y with the initial vector clock values as <1,0,0>, <0,1,0>, and <0,0,1> . The reads on x by the 3 threads are not ordered by happens before, so both FastTrack and DJIT+ store all the 3 in a vector clock.

[14] When the operation release on lock m is performed, vector clock of T_3 is copied to Lm in this case <0,0,1> is copied to Lm (this operation takes an O(n) time) and then T_3 increments its clock to <0,0,2>. When T_1 acquires Lm it performs a join operation with the vector clock of Lm, so the new vector clock is updated to <1,0,1>. When write on x is observed in the trace at T_1, FastTrack first checks for the write-write races and then read-write races. Since in this case there is no write-write race, as before there is no previous writes to x so it does not report any race. But there is a read-write race, since Rx on T_2 is not ordered by happens before with Wx in T_1. So, FastTrack checks for Rx happens before Wx at T_1, since they are not ordered by happens before, FastTrack reports a race. Now consider the access on the shared variable y, when the write access by T_3 happens, FastTrack updates the write epoch to 1@3 (since there are no access before this operation as there are no races). The next access on y is a write accesses by T_1, but WY at T_3 happens before WY at T_1 since there is synchronization operation rel(Lm) at T_3 followed by acq(Lm) at T_1. So the write epoch of x gets updated to 1@1. If the next access is by T_3 and is read access, then there is write-read race i.e RY at T_3 are not ordered by happens before with WY at T_1 and so, FastTrack reports this race (It compares 1@1 at write epoch at x with 0@1 at T_3). Suppose WY at T_2 occurs after

WY at T_1 in the trace then there is a write-write race and FastTrack reports this race (It compares 1@1 at write epoch of x with 0@1 at T_2).

[15] FastTrack claims significant performance and space improvement over the DJIT+ algorithm and represent the state-of-the-art in vector clock based dynamic data race detector. Though, FastTrack improves the time and space overhead for memory operations (completely for write operations and partially for read operation), the synchronization operation such as acquire/release for maintaining and updating vector clocks also have considerable overheads of the O(n). These overheads are further going to increase with more and more threads contending in multi-cores and GPGPUs over the shared data.

Disclosure of Invention

Technical Problem

[16] The principal object of embodiments herein is to reduce overheads of maintaining and updating vector clocks during synchronization for dynamic data race detection.

[17] Another object of embodiments herein is to orthogonally improve the performance of vector clock based dynamic data race detection over the state-of-the-art techniques without affecting the precision of dynamic data race detection by maintaining and updating the vector clocks for synchronization operation.

Solution to Problem

[18] Accordingly embodiments herein provide a method for reducing overheads orthogonally during synchronization of threads in a vector clock based dynamic data race detection system. The method comprises opportunistically reducing the complexity of updating clock values during a thread synchronization operation.

[19] One embodiment herein provides a method A method for reducing overheads orthogonally during synchronization in a vector clock based dynamic data race detector between a first thread and a second thread using a lock when said second thread is acquiring said lock from said first thread, by updating entire vector of clock values in said second thread with corresponding maximum clock value for each thread where said maximum clock value for each thread is obtained by comparing clock value for each thread in said lock, the method characterized by maintaining previous version value in each among said threads being monitored, where said previous version of a thread among said threads being monitored is a version after which there are no updates from any thread other than said thread; maintaining previous version value in each lock, where said previous version is the version of a thread that last released said lock; checking for a condition, if previous version value of said first thread is not more than version value of said first thread in version vector of said second thread; and when previous version value of said first thread is not more than version value of said first

thread in version vector of said second thread, updating the clock value of said first thread to said second thread and retaining clock values of threads other than said first thread without updating.

[20] Embodiments herein also disclose a system for perform various methods disclosed herein.

[21] These and other aspects of the embodiments herein will be better appreciated and understood when considered in conjunction with the following description and the accompanying drawings. It should be understood, however, that the following descriptions, while indicating preferred embodiments and numerous specific details thereof, are given by way of illustration and not of limitation. Many changes and modifications may be made within the scope of the embodiments herein without departing from the spirit thereof, and the embodiments herein include all such modifications.

Brief Description of Drawings

[22] This invention is illustrated in the accompanying drawings, through out which like reference letters indicate corresponding parts in the various figures. The embodiments herein will be better understood from the following description with reference to the drawings, in which:

[23] FIG. 1 illustrates an example of functioning, FastTrack dynamic data race detector in the context of the invention,

[24] FIG. 2 illustrates handling of synchronization operations in dynamic data race according to prior art,

[25] FIG. 3 illustrate handling of synchronization operations in dynamic data race according to embodiments disclosed herein,

[26] FIG. 4 illustrates a thread interaction scenario associated with maintaining and updating vector clocks for synchronization operation, according to one embodiment,

[27] FIG. 5 illustrates the thread interaction scenario associated with maintaining and updating vector clocks for synchronization operation, according to another embodiment,

[28] FIG. 6 illustrates the thread interaction scenario associated with maintaining and updating vector clocks for synchronization operation, according to yet another embodiment,

[29] FIG. 7 illustrates the thread interaction scenario associated with maintaining and updating vector clocks for synchronization operation, according to further embodiment, and

[30] FIG. 8 and FIG. 9 illustrate an example computing environment that may be used in implementing the embodiments disclosed herein.

Mode for the Invention

[31] The embodiments herein and the various features and advantageous details thereof are explained more fully with reference to the non-limiting embodiments that are illustrated in the accompanying drawings and detailed in the following description. Descriptions of well-known components and processing techniques are omitted so as to not unnecessarily obscure the embodiments herein. The examples used herein are intended merely to facilitate an understanding of ways in which the embodiments herein may be practiced and to further enable those of skill in the art to practice the embodiments herein. Accordingly, the examples should not be construed as limiting the scope of the embodiments herein.

[32] The embodiments herein enable a method and system to reduce overheads of maintaining and updating vector clocks during synchronization by opportunistically reducing complexity of operations from O(n) in time and space overheads of synchronization to O(1). Referring now to the drawings, and more particularly to FIGS. 1 through 9, where similar reference characters denote corresponding features consistently throughout the figures, there are shown preferred embodiments.

[33] Embodiments herein enable opportunistic reduction of O(n) time and space overheads of synchronization, exploiting the fact that there is temporal locality in the thread interactions. Essentially, threads tend to interact locally with each other over time and interaction is not completely haphazard in nature. If a thread TX has lock Li k times, where k>=1 before TY acquires Li and there is no thread TZ, where z!=x and z!=y which acquires Li between successive releases of TX followed by final acquire of TY, and the last update of TY was received from TX, then the expensive O(n) join operation can be converted to a O(1) join operation (for all but first Joins), where n is the number of threads.

[34] Embodiments herein achieve the opportunistic reduction of complexity of join operations. The improvement is illustrated through the use of data structure of ThreadState and LockState representing state of a thread and state of a lock used by various threads respectively, according to an example implementation of a preferred embodiment showing improvement over an example implementation of FastTrack.

[35] **Structure for ThreadState and LockState as used by FastTrack:**

[36] class ThreadState {

[37] int tid; //ThreadId

[38] int C[]; //Vector of Clocks of all the threads maintained by each thread

[39] int epoch; // clock value of tid (c@tid)

[40] int Version[]; //Contains last version value of each thread at the time join with that thread

[41] int Vepoch; //same as version value of tid(Version(tid))

[42] int Pversion;

[43] }

[44] class LockState {

[45] int C[]; //copy of vector clock of last thread that released the lock

[46] int Vepoch; //same as version value of tid with tid (v@tid) is the thread last released the lock.

[47] int Pversion;

[48] }

[49] **Improved structure for ThreadState and Lockstate**:

[50] class ThreadState {

[51] int tid; //ThreadId for this thread

[52] int C[]; // Clocks of all threads maintained by each

[53] // thread as its vector clock

[54] int epoch; // clock value of tid (c@tid)

[55] int Version[]; // version of each thread at the time of last join

[56] int Vepoch; // Version Number of thread after which there is no

[57] **int Pversion;** } // change in the clock of any other thread except this

[58] // tid

[59] class LockState {

[60] int C[]; // vector clock copy of last thread that released the lock

[61] int Vepoch; // version value of tid, last thread that released the lock

[62] **int Pversion;** } //Pversion of last thread that released the lock

[63] ***Some of the notations are explained further as follows:***

[64] ***Version*** is incremented every time there a change in vector clock

[65] ***Versiont [1::n]*** is a Version vector of thread where each element of version vector is version value of corresponding thread.

[66] ***Versiont[u]*** is the latest version received by the thread under consideration from the thread u that it joins.

[67] ***L.Vepoch*** is maintained for a lock L is same as version value of tid that is the thread last released the lock.

[68] ***Vepoch:*** (Version epoch v@t) is the current version v of thread t i.e (Versiont[t]) in the given vector clock.

[69] ***Pversioni (previous version of Ti) :*** Denotes the version number of Ti after which there is no change in the clock of any other thread except that of Ti in the vector clock maintained by Ti

[70] The improved versions of ThreadState and LockState introduce the variables Pversion, representing the previous version of a thread. Pversioni represents the previous version of thread Ti and denotes the version number of Ti after which there is no change in the clock of any other thread except that of Ti in the vector clock

maintained by Ti. Therefore, each thread maintains at least the following metadata:

[71] - Version of a thread is a scalar, is incremented every time there is a change in any element of the vector lock maintained by the thread. Versiont [1::n] is a Version vector of thread where each element of version vector, Versiont[u], is the latest version received by the thread t under consideration from the thread u that it joins.

[72] - L.Vepoch is maintained for a lock L, is same as version value of tid, the thread last released the lock.

[73] - Vepoch (Version epoch v@t) is the current version v of thread t i.e (Versiont[t]) in the given vector clock.

[74] - Pversioni (previous version of Ti) denotes the version number of Ti after which there is no change in the clock of any other thread except that of Ti in the vector clock maintained by Ti.

[75] - Let Li be a Lock and Ti be the thread that has last released Li. The corresponding vector clock (C), Ti.Vepoch, Pversion, thread id of Li are denoted by L.Ci, L.Vepochi , L.Pversioni ,L.Tidi, which are same as corresponding values of Ti at the time of release of Li by Ti.

[76] The improvement brought about by the embodiments herein may be stated through the following Lemma:

[77] - Let T_1 , T_2 be two threads such that T_1 releases a lock L_1 at time t_1, which is next acquired by T_2 and T_1 releases a lock L_2 at time t_2 which is next acquired by T_2 where $t_2 > t_1$. If the Pversion1 does not change in between t_1 to t_2 then reduce the O(n) acquire operation by T_2 at t_2 to O(1) acquire operation.

[78] The implementation of the aforementioned Lemma is described through the following illustration and subsequent examples:

[79] When T_1 releases lock L_1 it takes O(n) time for copying C_1 to L_1.C . This is followed by an increment of $C_1(1)$; i.e., the clock value of T_1 as maintained by T_1 in its vector C_1 is incremented. Further, Pversion1 is copied to L_1.Pversion.

[80] Subsequently, when T_2 acquires the lock L_1 from T_1, FastTrack does an expensive O(n) join operation by checking if $L_1.C(1) > C_2(1)$. However, embodiments herein avoid redundant O(n) join operation by checking if L_1.Pversion <= $Version_2(1)$, to check if version value of thread T_1 after which there are no updates for any other threads is not more than version value of thread T_1 in the vector of thread T_2.

[81] If the condition is true, that is if the current Pversion value of thread T_1 is not more than version value of thread T_1 as in the vector of thread T_2, it would mean that there was no acquisition of lock L_1 by thread T_1 or any other synchronization operation like join since last update, and that the last update of thread T_1 was received by thread T_2.

[82] If the condition is false, that is if the current Pversion value of thread T_1 is more than version value of thread T_1 as in the vector of thread T_2, it would mean that there was

another join operation involving threat T_1 and lock L_1 since the last join operation between threads T_1 and T_2, and that the last update to thread T_1 was not received by thread T_2.

[83] If the condition is true, there would be no need to update the entire clock vector of thread T_2 as only thread T_1 was updated since the last update to thread T_2. Therefore, in the improved method, the check is followed by updating Pversion2, $C_2(1)$ and $Version_2(1)$ to T_2.Vepoch, L_1.C(1), and L_1.Vepoch.

[84] However, if the condition is false, the complex O(n) operation as performed by FastTrack would be adopted to update thread T_2.

[85] EXAMPLE IMPLEMENTATION

[86] The synchronization operation of obtaining a lock by thread T_2 from thread T_1 as performed by FastTrack may be illustrated using the following pseudo code:

```
[87] Void join(ThreadState t, LockState m){
[88] // O(n) operation to update all thread clock values
[89] t.C[u] = max(t.C[u],m.C[u]) for all u;
[90] }
```

[91] As illustrated in FIG. 2, FastTrack always performs the synchronization operation with O(n) complexity.

[92] The improved synchronization operation of obtaining a lock by thread T_2 from thread T_1 according to embodiments herein may be illustrated using the following pseudo code:

```
[93] Void join(ThreadState t, LockState m){
[94] if (m.L[u] > t.C[u] for any u) {
[95] t.lastVersion = t.lastVersion + 1
[96] t.Pversion = t.lastVersion //update Pversion
[97] if(m.Pversion ≤ t.Version[u]) { //check for redundant join.
[98] t.Version[u] = vepoch(m) // where u is m.tid
[99] t.C[u]= m.C[u] //thread t clock is updated by clock of u
[100] return; //avoid the O(n) path below and return
[101] }
[102] t.C[u]= max(t.C[u],m.L[u]) for all u; //expensive 0(n) path
[103] }
```

[104] As illustrated in FIG. 3, the improved method performs a check to reduce the complexity of the synchronization operation to O(1).

[105] The improved method of performing synchronization may be illustrated further using the following examples:

[106] EXAMPLE 1: In example 1, threads T_1, T_2, and T_3 are interacting as depicted in FIG. 4. When T_3 acquires Lx from T_1, the vector clock, version vector (Pversion) and

version epoch of T_3 is updated that essentially takes O(n) time, where n is the number of threads. Similarly, when T_3 acquires Lz from T_2 then the vector clock, version vector (Pversion) and version epoch of T_3 is updated. Since there are no operations changing vector clock of T_1 between the release of Lx and Ly, except the clock and current version of T_1. So when T_3 acquires Ly, embodiments herein do O(1) check as Pversion of T_1 <= version of T_1 in T_3 and increment the T_3.Vepoch. Followed by update of $C_3(1)$, Pversion3 and Ver3(1) to clock of T_1, T_3.Vepoch and current version of T_1 respectively, when T_1 released Ly. Similarly when T_3 acquires Lw, perform O(1) operation. This is in contrast to the FastTrack method that performs O(1) check in case of last two acquires as explained above.

[107] EXAMPLE 2: In example 2, consider threads T_1, T_2, and T_3 out of many active threads are interacting as depicted in FIG. 5. When T_2 acquires Lx from T_1, the vector clock, version vector (Pversion) and version epoch of T_2 is updated. This takes O(n) time ,where n is the number of threads. Similarly when T_3 acquires Ly from T_1 then the vector clock, version vector (Pversion) and version epoch of T_3 is updated. Because, there are no acquire operations that change the vector clock of T_1 between the Rel(Lx) and Rel(Lz), only the clock and version epoch of T_1 are modified. Thereafter, when T_3 acquires Lz, the present invention perform a simple check to see if the Pversion of T_1 at the time of release of Lz <= $Version_1(3)$ (O(1) check) and increment the T_3.Vepoch. Then update the $C_3(1)$, Pversion3 and $Version_3(1)$ to clock of T_1, T_3.Vepoch and version epoch of T_1 at the time of release of Lz.. Similarly, when T_2 acquires La,. the present invention perform O(1) operation thereby reducing some O(n) operations to O(1), where n is the number of threads.

[108] EXAMPLE 3: In example 3, consider four threads (T_1, T_2, T_3 and T_4) out of many active threads which interact as depicted in FIG. 6. Suppose the threads T_2 and T_3 are in separate loops then, one of the possible interleaving between the T_2 and T_3 can be as follows. T_2 acquires Lx and releases Lx followed by acquire of Lx and release of Lx by T_3. This is further followed acquire and release of Ly where y!=x by T_2 (assume that intial acquire of Ly by T_2 is redundant join O(1)) for 'k'times and acquire of Ly by T_3. In this scenario, when T_3 first acquires Lx when it is first released by T_2 it does an O(n) join operation.

[109] Thereafter the all the subsequent consecutive acquire and release of Ly by T_2 only increments the clock and version epoch of T_2. The next time when T_3 acquires Ly, the present invention check to see if the Pversion of T_2 at the time of release of Ly i.e. Ly.Pversion <= $Version_3(2)$ (O(1) check) and increment T_2.Vepoch , T_3.Vepoch, followed by updating $C_3(2)$ and $Version_3(2)$ to clock of T_2 and version epoch of T_2 at the time of release of Lx. This reduces O(n) operations to O(1).Similarly in a scenario of T_3 executing 'k' times followed by acquire of L_1 by T_2, thus present method reduce

O(n) overheads to O(1).

[110] EXAMPLE 4: In example 4, consider four threads T_1, T_2, T_3, and T_4 out of many active threads with interaction as depicted in FIG. 7. Initially threads T_2 and T_3 interact followed by the interaction between T_2 and T_1, and followed by the interaction between T_3 and T_4.

[111] Consider the interaction between T_2 and T_3, when the first release and acquire on LU is performed, the join operation which takes place at T_3 takes O(n) time . Next time, when T_3 acquires Lv, the present method check to see if the Pversion of T_2 at the time of release of LV. <= $Version_3(2)$ (O(1) check) and increment the current version of T_3, followed by update of $C_3(2)$, $Version_3(2)$ to clock of T_2 and version epoch of T_2 at the time of release of Lv. This reduces O(n) operations to O(1). Similarly, the first release and acquire between T_3 and T_2 takes O(n) time and the subsequent release and acquire between T_3 and T_2 takes O(1) time. Similarly, O(n) operations between T_1 , T_2, T_3, and T_4 are reduced. Thus, embodiments herein reduce many O(n) operations to O(1) operation.

[112] FIG. 8 illustrates a computing environment implementing the application as disclosed in an embodiment herein. As depicted the computing environment comprises at least one processing unit that is equipped with a control unit and an Arithmetic Logic Unit (ALU), a memory, a storage unit, plurality of networking devices, and a plurality Input output (I/O) devices. The processing unit is responsible for processing the instructions of the algorithm. The processing unit receives commands from the control unit in order to perform its processing. Further, any logical and arithmetic operations involved in the execution of the instructions are computed with the help of the ALU. Processing unit can support more than one threads

[113] FIG. 9 illustrates another computing environment implementing the application as disclosed in an embodiment herein. As depicted the computing environment comprises of more than one processing units that are equipped with a control unit and an array of Arithmetic Logic Units (ALUs) and a multilevel local memory (cache hierarchy). Additionally, the computing environments have a storage unit, plurality of networking devices, and a plurality Input output (I/O) devices. The processing units in this case can be same, similar or widely different in their capabilities and can support plurality of threads. The overall computing environment can be composed of multiple homogeneous and/or heterogeneous cores, multiple GPUs of different kinds, special media and other accelerators. The processing unit is responsible for processing the instructions of the algorithm. The processing unit receives commands from the control unit in order to perform its processing. Further, any logical and arithmetic operations involved in the execution of the instructions are computed with the help of the ALU. Further, the plurality of process units may be located on a single chip or over multiple

chips.

[114] The instructions and codes required for the implementation are stored in either the memory unit or the storage or both. At the time of execution, the instructions may be fetched from the corresponding memory and/or storage, and executed by the processing unit.

[115] In case of any hardware implementations various networking devices or external I/O devices may be connected to the computing environment to support the implementation through the networking unit and the I/O device unit.

[116] In some embodiments, the methods disclosed herein may be implemented as part of a thread library. In some embodiments, the methods disclosed herein may be implemented as part of a runtime system like a Just-In-Time compile system. In some embodiments the methods disclosed herein may be implemented as part of an Operating System (OS).

[117] In some embodiments the methods disclosed herein may be made use of by a hardware system with specific instruction set architecture. Such a hardware system may use specific registers for storing state information of threads. The storing of state information may happen in the register memory or on an external system memory.

[118] In some embodiments, the methods disclosed herein may be implemented in a multi-thread embedded system environment.

[119] In various embodiments, the methods for reducing overheads during synchronization operations may further be enhanced for certain systems by performing sampling of thread interactions. Embodiments disclosed herein suggested monitoring all thread interactions. However, as number of threads and thread interactions grow, there may be a need to sample thread interactions to reduce overheads. Further, sampling of thread interactions may be implemented in systems that have severe memory usage restrictions during runtime.

[120] The embodiments disclosed herein can be implemented through at least one software program running on at least one hardware device. Therefore, it is understood that the scope of the protection is extended to such a program and in addition to a computer readable means having a message therein, such computer readable storage means contain program code means for implementation of one or more steps of the method, when the program runs on a server or mobile deviceor any suitable programmable device. The method is implemented in a preferred embodiment through or together with a software program written in e.g. Very high speed integrated circuit Hardware Description Language (VHDL) another programming language, or implemented by one or more VHDL or several software modules being executed on at least one hardware device. The hardware device can be any kind of portable device that can be programmed. The device may also include means which could be e.g. hardware means

like e.g. an ASIC, or a combination of hardware and software means, e.g. an ASIC and an FPGA, or at least one microprocessor and at least one memory with software modules located therein. The method embodiments described herein could be implemented partly in hardware and partly in software. Alternatively, the invention may be implemented on different hardware devices, e.g. using a plurality of CPUs.

[121] The foregoing description of the specific embodiments will so fully reveal the general nature of the embodiments herein that others can, by applying current knowledge, readily modify and/or adapt for various applications such specific embodiments without departing from the generic concept, and, therefore, such adaptations and modifications should and are intended to be comprehended within the meaning and range of equivalents of the disclosed embodiments. It is to be understood that the phraseology or terminology employed herein is for the purpose of description and not of limitation. Therefore, while the embodiments herein have been described in terms of preferred embodiments, those skilled in the art will recognize that the embodiments herein can be practiced with modification within the spirit and scope of the embodiments as described herein.

Claims

[Claim 1] A method for reducing overheads orthogonally during synchronization of threads in a vector clock based dynamic data race detection system, said method comprising
opportunistically reducing the complexity of updating clock values during a thread synchronization operation.

[Claim 2] The method as in claim 1, wherein said method opportunistically reduces complexity of said synchronization operation from O(n) to O(1) wherein n represents the number of threads being monitored.

[Claim 3] A method for reducing overheads orthogonally during synchronization in a vector clock based dynamic data race detector between a first thread and a second thread using a lock when said second thread is acquiring said lock from said first thread, by updating entire vector of clock values in said second thread with corresponding maximum clock value for each thread where said maximum clock value for each thread is obtained by comparing clock value for each thread in said lock, said method characterized by
maintaining previous version value in each among said threads being monitored, where said previous version of a thread among said threads being monitored is a version after which there are no updates from any thread other than said thread;
maintaining previous version value in each lock, where said previous version is the version of a thread that last released said lock;
checking for a condition, if previous version value of said first thread is not more than version value of said first thread in version vector of said second thread; and
when previous version value of said first thread is not more than version value of said first thread in version vector of said second thread,
updating the clock value of said first thread to said second thread and retaining clock values of threads other than said first thread without updating.

[Claim 4] The method as in claim 3, wherein said method opportunistically reduces complexity of said synchronization operation between said first thread and second thread from O(n) to O(1) wherein n represents the number of threads being monitored.

[Claim 5] The method as in claim 3, wherein said method comprises sampling

thread interactions before checking for said condition to reduce overhead.

[Claim 6] A system for performing a method according to at least one of claims 1 to 5.

[Claim 7] The system as in claim 6, wherein said system is a single processor system.

[Claim 8] The system as in claim 6, wherein said system is a multi-processor system.

[Claim 9] The system as in claim 6, wherein said system is a homogeneous processor system.

[Claim 10] The system as in claim 6, wherein said system is a heterogeneous processor system.

[Claim 11] A computer program product embodied in a computer readable medium including program instructions which when executed by a processor cause the processor to perform a method for reducing overheads orthogonally during synchronization of threads in a vector clock based dynamic data race detection system, said method comprising opportunistically reducing the complexity of updating clock values during a thread synchronization operation.

[Claim 12] The computer program product as in claim 11, wherein said method opportunistically reduces complexity of said synchronization operation from O(n) to O(1) wherein n represents the number of threads being monitored.

[Claim 13] A computer program product embodied in a computer readable medium including program instructions which when executed by a processor cause the processor to perform a method for reducing overheads orthogonally during synchronization in a vector clock based dynamic data race detector between a first thread and a second thread using a lock when said second thread is acquiring said lock from said first thread, by updating entire vector of clock values in said second thread with corresponding maximum clock value for each thread where said maximum clock value for each thread is obtained by comparing clock value for each thread in said lock, said method characterized by

maintaining previous version value in each among said threads being monitored, where said previous version of a thread among said threads being monitored is a version after which there are no updates from any thread other than said thread;

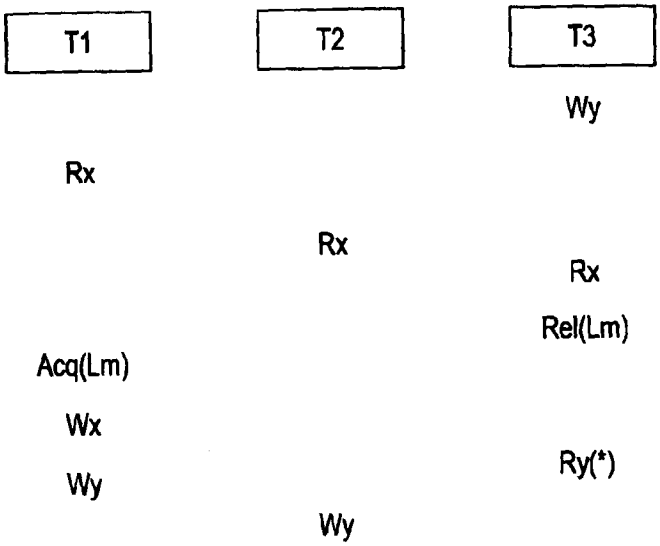
maintaining previous version value in each lock, where said previous

version is the version of a thread that last released said lock;
checking for a condition, if previous version value of said first thread is not more than version value of said first thread in version vector of said second thread; and
when previous version value of said first thread is not more than version value of said first thread in version vector of said second thread,
updating the clock value of said first thread to said second thread and retaining clock values of threads other than said first thread without updating.

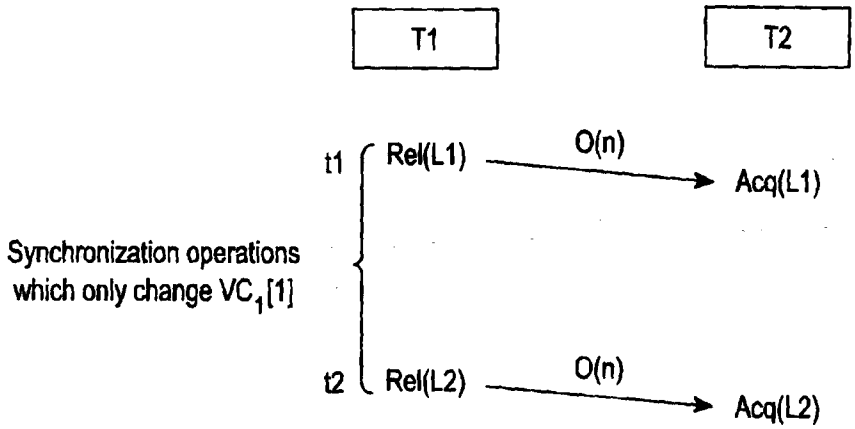
[Claim 14] The computer program product as in claim 13, wherein said method opportunistically reduces complexity of said synchronization operation between said first thread and second thread from O(n) to O(1) wherein n represents the number of threads being monitored.

[Claim 15] The computer program product as in claim 13, wherein said method comprises sampling thread interactions before checking for said condition to reduce overhead.

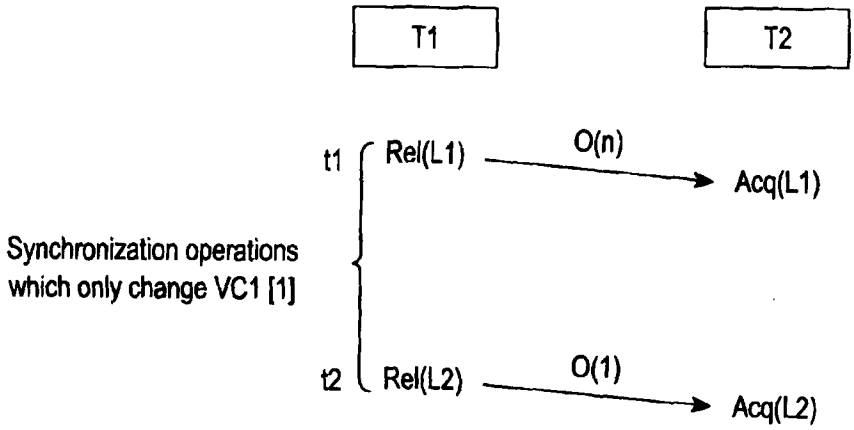
[Fig. 1]


T1
T2
T3
Wy
Rx
Rx
Rx
Rel(Lm)
Acq(Lm)
Wx
Ry(*)
Wy
Wy

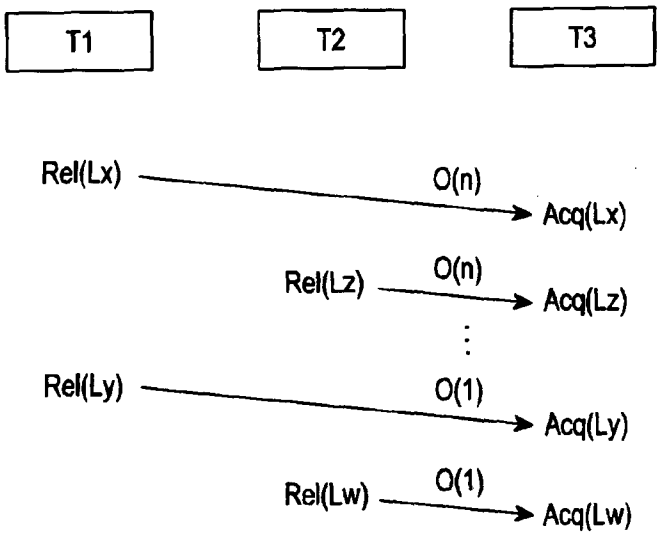
[Fig. 2]


T1
T2
t1
Rel(L1)
O(n)
Acq(L1)
Synchronization operations which only change VC1[1]
t2
Rel(L2)
O(n)
Acq(L2)

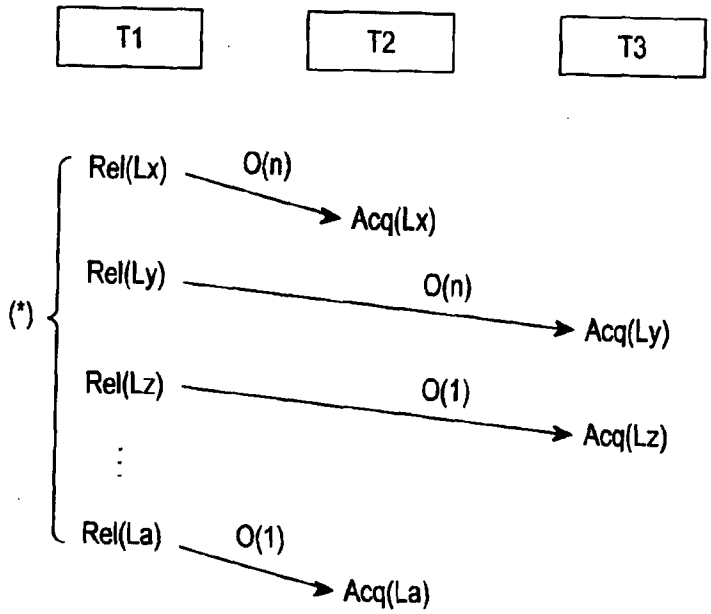
[Fig. 3]


T1
T2
t1
Rel(L1)
O(n)
Acq(L1)
Synchronization operations
which only change VC1 [1]
t2
Rel(L2)
O(1)
Acq(L2)

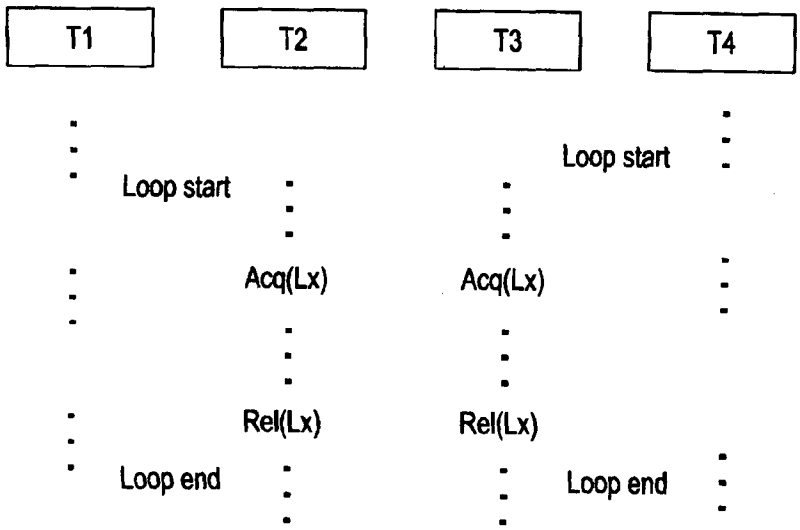
[Fig. 4]


T1
T2
T3
Rel(Lx)
O(n)
Acq(Lx)
Rel(Lz)
O(n)
Acq(Lz)
Rel(Ly)
O(1)
Acq(Ly)
Rel(Lw)
O(1)
Acq(Lw)

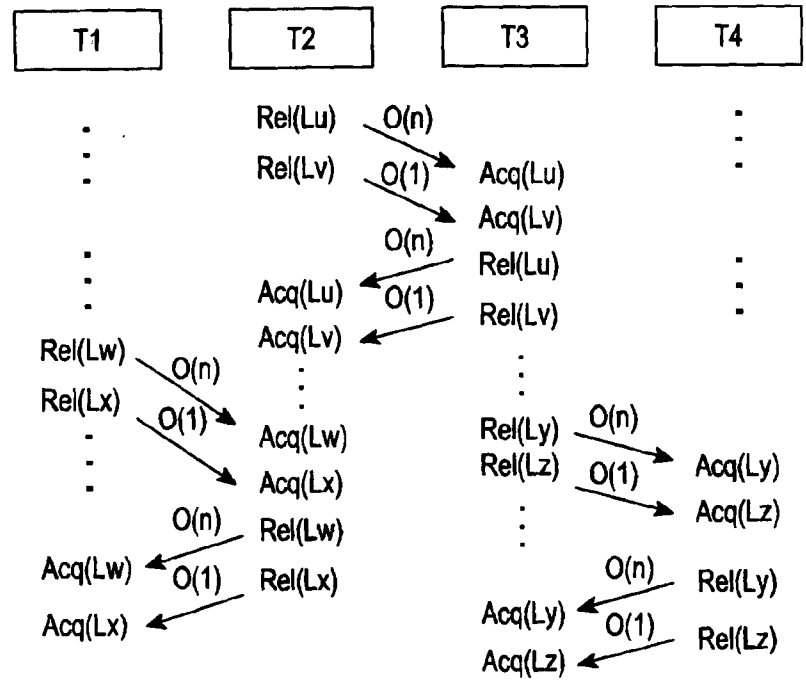
[Fig. 5]


T1
T2
T3
(*)
Rel(Lx)
O(n)
Acq(Lx)
Rel(Ly)
O(n)
Acq(Ly)
Rel(Lz)
O(1)
Acq(Lz)
Rel(La)
O(1)
Acq(La)

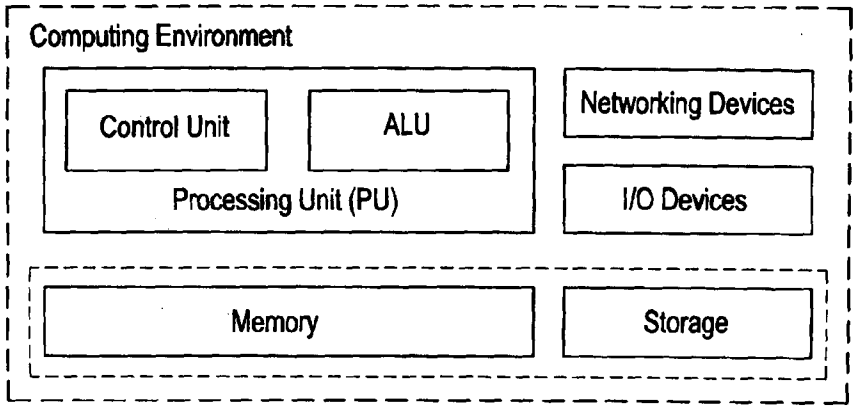
[Fig. 6]


T1
T2
T3
T4
Loop start
Loop start
Acq(Lx)
Acq(Lx)
Rel(Lx)
Rel(Lx)
Loop end
Loop end

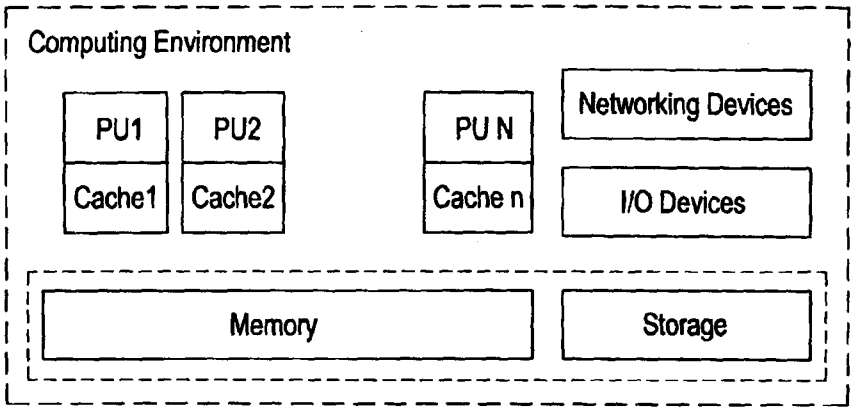
[Fig. 7]


T1
T2
T3
T4
Rel(Lu)
O(n)
Rel(Lv)
O(1)
Acq(Lu)
Acq(Lv)
O(n)
Rel(Lu)
Acq(Lu)
O(1)
Rel(Lv)
Acq(Lv)
Rel(Lw)
O(n)
Rel(Lx)
O(1)
Acq(Lw)
Acq(Lx)
Rel(Ly)
O(n)
Rel(Lz)
O(1)
Acq(Ly)
Acq(Lz)
O(n)
Rel(Lw)
Acq(Lw)
O(1)
Rel(Lx)
Acq(Lx)
O(n)
Rel(Ly)
Acq(Ly)
O(1)
Rel(Lz)
Acq(Lz)

[Fig. 8]


Computing Environment
Control Unit
ALU
Processing Unit (PU)
Networking Devices
I/O Devices
Memory
Storage

[Fig. 9]


Computing Environment
PU1
PU2
Cache1
Cache2
PU N
Cache n
Networking Devices
I/O Devices
Memory
Storage