On Scalable Synchronization for Distributed Embedded Real-time Systems

Sherif F. Fahmy[1], Binoy Ravindran[1], and E. Douglas Jensen[2]

[1] ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA, fahmy@vt.edu, binoy@vt.edu
[2] The MITRE Corporation, Bedford, MA 01730, USA, jensen@mitre.org

Abstract. We consider the problem of programming distributed embedded real-time systems with distributed dependencies. We show that the de facto standard of using locks and condition variables in conjunction with threads can have significant overhead and semantic difficulty and suggest alternative programming abstractions to alleviate these problems. We also discuss several alternatives for implementing these programming abstractions and discuss the algorithms and protocols needed.

1 Introduction

As Moore's law appears to be reaching its limits, manufacturers of computing machinery are turning (again) to parallelism as the next frontier in the quest for faster computers. Today, most machines produced are multi-core and the use of distributed systems is on the increase. Coinciding with this new direction of using concurrency to increase application throughput, is the discovery of a rich set of applications that are a natural fit for parallel and distributed architectures. From distributed databases to emerging distributed real-time systems [1], such emerging applications are only meaningful in a distributed system with multiple computing cores cooperating to execute the semantics of the application.

This parallelism offers a great opportunity for improving performance by increasing application concurrency. Unfortunately, this concurrency comes at a cost: programmers now need to design programs, using existing operating system and programming language features, to deal with shared access to serially reusable resources and program synchronization. The de facto standard for programming such systems is using threads, locks, and condition variables. Using these abstractions, programmers have been trying to write correct concurrent code ever since multitasking operating systems made such programs possible.

Unfortunately, the human brain does not seem to be well suited for reasoning about concurrency [2]. The history of the software industry contains numerous cases where the difficulty inherent in reasoning about concurrent code has resulted in costly software errors that are very difficult to reproduce and hence debug and fix. Among the more common errors encountered in lock-based software systems are deadlocks, livelocks, lock convoying, and, in systems where priority is important (e.g, embedded real-time systems), priority inversion. Such errors stem from the difficulty in reasoning about concurrent code.

Transactions have proven themselves to be a successful abstraction for handling concurrency in database systems. Due to this success, researchers have attempted to take advantage of their features for non-database systems. In particular, there has been significant recent efforts to apply the concepts of transactions to shared memory. Such an attempt originated as a purely hardware solution [3, 4] and was later extended to deal with systems where transactional support was migrated from the hardware domain to the software domain [5]. Software transactional memory (or STM) has, until recently, been an academic curiosity because of its high overhead. However, as the state-of-the-art improved and more efficient algorithms were devised, a number of commercial and non-commercial STM systems have been developed (see implementations section of [6]). In this position paper, we discuss the issues involved in implementing software transactional memory in distributed embedded real-time systems.

2 Motivation

Currently, the industry standard abstractions for programming distributed embedded systems include OMG/Real-Time CORBA's client/server paradigm and distributable threads [7] and OMG/DDS's publish/subscribe abstraction [8]. The client/server and distributable threads abstractions directly facilitate the programming of causally-dependent, multi-node application logic. In contrast, the publish/subscribe abstraction is a data distribution service for logically-single hop communications (i.e., from one publisher to one subscriber), and therefore, higher-level abstractions must be constructed – on an application-specific basis – to express causally-dependent, multi-node application logic (e.g., publication of topic A depends on subscription of topic B; B's publication, in turn, depends on subscription of topic C, and so on). All of these abstractions rely on lock-based mechanisms for concurrency control, and thus suffer from their previously mentioned inherent limitations.

In particular, lock-based concurrency control can easily result in local and distributed deadlocks, due to programming errors that occur as a result of the conceptual difficulty of the (lock-based) programming model. Detecting and resolving deadlocks, especially distributed deadlocks, that can potentially arise due to distributed dependencies is complex and expensive. Note that deadlocks can only be detected and resolved, as opposed to being avoided or prevented, in those distributed embedded systems where it is difficult to obtain a-priori knowledge of which activities need which resources and in what order. When a deadlock is detected in such systems, the usual method of resolving it is to break the cycle of the waiting processes by terminating one of them. Unfortunately, the choice of which process to terminate is not a simple one in real-time systems. By terminating one of the processes that are waiting in a cycle, we produce a chain of waiting processes. Depending on how, i.e., where, we break this cycle, it may or may not be feasible to meet the timing requirements of the remaining processes. Thus, we need to consider the structure of the dependency chain, after terminating a process to end the deadlock, in order to break the cycle in a way

that optimizes end-to-end timeliness objectives. Furthermore, a process's dependencies must be taken into account when making the choice about which process to terminate. For example, if a significant number of processes depend on the result of a process, terminating it to resolve a deadlock may not be in the best interest of the application. In addition, the cost of deadlock detection/resolution is exacerbated by the extra work necessary to restore the system to an acceptable state when failure occurs. Thus, deadlock resolution is a complex process.

The problem of distributed deadlock detection and resolution has been exhaustively studied, e.g., [9–15]. A number of these algorithms turned out to be incorrect by either detecting phantom deadlocks (false positives) or not detecting deadlocks when they do exist, e.g., [11,16]. These errors occur because of the inherent difficulty of reasoning about distributed programs. This led to attempts at providing a formal method for analyzing such protocols to ensure correct behavior (e.g., [13]). Despite the difficulty of reasoning about distributed deadlock, solutions for this problem on synchronous distributed systems have been developed. Unfortunately, for asynchronous systems, errors in the deadlock detection process become inevitable. For real-time systems, these issues become more severe [9]. The semantic difficulty of thread and lock based concurrency control and the high overhead associated with detecting and resolving distributed deadlock, as indicated above, are the driving motivations for finding different programming abstractions for distributed embedded real-time systems.

3 Previous work

3.1 Alternatives to lock-based programming

Academia, and certain parts of industry, have realized the limitations of lock-based software, thus a number of proposed alternatives to lock-based software exist. The design of lock-free, wait-free or obstruction-free data structures is one such approach. The main problem with this approach is that it is limited to a small set of basic data structures, e.g., [17–19]. For example, to the best of our knowledge, there is no lock-free implementation of a red-black tree that does not use STM. Most of the literature on lock-free data structures concentrates on basics such as queues, stacks, and other simple data structures. It should be noted that lock-freedom, wait-freedom and obstruction-freedom are concepts and as such can encompass non lock-based solutions like STM. However, we use these terms in this context to refer to hand crafted code that allows concurrent access to a data structure without suffering from race conditions.

The discrete event model presented in [20,21] provides an interesting alternative to thread based programming. While interesting and novel, it still remains to be seen whether programmers find the semantics of the model easier than the semantics of thread-based computing. In addition, the requirement of static analysis to determine a partial order on the events makes the system inapplicable to dynamic systems where little or no information is available a priori.

Transactional processing, the semantic ancestor of STM, has been around for a significant period of time and has proven its mettle as a method of providing

concurrency control in numerous commercial database products, in addition, it does not place any restriction on the dynamism of the system on which it is deployed. Unfortunately, the use of a distributed commit protocol, such as the two-phase commit protocol, increases the execution time of a transaction and can lead to deadline misses [22]. STM is a lighter-weight version of transactional processing, with no distributed commit protocol required in most cases. As such, it allows us to gain the benefits of transactional processing (i.e., fault tolerance and semantic simplicity), without incurring all its associated overhead.

We believe that STM is an attractive alternative to thread and lock-based distributed programming, since it eliminates many of the conceptual difficulties of lock-based concurrency control at the expense of a justifiable overhead that becomes less significant as the number of processors in the system scales.

3.2 Software transactional memory

Since the seminal papers about hardware and software transactional memory were published, renewed interest in the field has resulted in a large body of literature on the topic (e.g, see [23–25]). This body of work encompasses both purely software transactional memory systems and hybrid systems where software and hardware support for transactional memory are used in conjuncture to improve performance. Despite this large body of work, to the best of our knowledge, only three papers investigate STM for distributed systems [26–28].

We believe that distributed embedded systems stand to benefit significantly from STM. Such systems are most distinguished by their need to: 1) react to external events asynchronously and concurrently; 2) react to external events in a timely manner (i.e., real-time); and 3) cope with failures (e.g., processors, networks) – one of the raison d'être for building distributed systems. Thus, concurrency that is fundamentally intrinsic to distributed embedded systems naturally motivates the usage of STM. Their need to (concurrently) react timely to external events in the presence of failures is also a compelling reason – such behaviors are very complex to program, reason about, and obtain timing assurances using lock-based concurrency control mechanisms.

There has also been a dearth of work on real-time STM systems. Notable work on transactional memory and lock-free data structures in real-time systems include [18,29–32]. However, most of these works only consider uni-processor systems (with [32] being a notable exception). In this position paper, we propose to study the issues involved in implementing STM in distributed embedded real-time systems. Past work has shown that STM has lower throughput for systems with a small number of processors compared to fine-grain lock-based solutions but that this difference in performance is quickly reversed as the number of processors scales [33]. This, coupled with easier programming semantics of STM, makes it an attractive concurrency control mechanism for next generation embedded real-time systems with multi-core architectures and high distribution.

With STM, deadlocks are entirely or almost entirely precluded. This will immediately result in significant reductions in the cost of scheduling and resource

management algorithms, as distributed dependencies are avoided and no expensive deadlock detection/resolution mechanisms are needed. Implementing higher level programming constructs, like, for example, Hoare's conditional critical regions (or CCR) [34], on top of STM [33], allows programmers to take advantage of the deadlock freedom and simple semantics of STM in their programs.

4 STM for distributed embedded systems

There are a number of competing abstractions for implementing STM in distributed embedded real-time systems. An interesting abstraction is the notion of real-time distributed transactional objects, where code is immobile and objects migrate between nodes to provide a transactional memory abstraction. Another alternative is to allow remote invocations to occur within a transaction, spawning sub-transactions on each node (where they are executed using STM), and using a distributed commit protocol to ensure atomicity. A third alternative is to provide a hybrid model, where both data and code are mobile and the decision of which is moved is heuristically decided either dynamically or statically. Several key issues need to be studied in order to use STM in distributed embedded systems, these are:

- Choosing an appropriate abstraction for including STMs in distributed embedded systems;
- Designing the necessary protocols and algorithms to support these abstractions;
- Implementing these abstractions in a programming language by making necessary changes to its syntax and in the run-time environment; and
- Designing scheduling algorithms to provide end-to-end timeliness using these new programming abstractions.

4.1 Choosing an appropriate abstraction.

STM is a technology for multiprocessor systems, to use it in a multicomputer environment, we need to develop appropriate abstractions. We are currently considering three competing programming abstractions into which to incorporate STM:

- A model where cross-node transactions are permitted using remote invocations and atomicity is enforced using an atomic commit protocol;
- A model where a distributed cache coherence protocol is used to implement an abstraction of shared memory on top of which we can build STM; and
- A hybrid model where code or data is migrated depending on a number of heuristics such as size and locality.

In the first approach, we manage concurrency control on each node using STM, but allow remote invocations to occur within a transaction. Thus we allow a transaction to span multiple nodes. At the conclusion of the transaction,

the last node on which transactional code is executed acts as a coordinator in a distributed commit protocol to ensure an atomic commitment decision. Our preliminary research, which we intend to elaborate upon, indicates that such an approach may be prone to "retry thrashing" especially when the STM implemented on each node is lock-free.

Since lock-free STM is an optimistic concurrency control mechanism, extending the duration of a transaction by allowing it to sequentially extend across nodes results in a significantly higher probability of conflicts among transactions. Such conflicts lead to aborted transactions that are later retried. Retrying is antagonistic to real-time systems since it degrades one of the most important features of real-time systems: predictability. Lock-based STM tends to reduce some of this "thrashing" behavior since it eliminates part of the "optimism" of the approach. However, long transactions are still more susceptible to retries and introducing locks into the STM implementation necessitates a deadlock detection and resolution solution. Fortunately such a solution does not need to be distributed since it only needs to resolve local deadlocks.

Implementing STM on top of a distributed cache coherence protocol has been investigated in [26,27]. In this approach, code is immobile, but data objects move among nodes as required. The approach uses a distributed cache coherence protocol to find and move objects. We intend to design real-time cache coherence protocols, where timeliness is an integral part of the algorithm. We plan to design STM on top of these protocols and compare their performance to the flow control abstraction. An important advantage of this approach is that it eliminates the need for a distributed commit protocol. Since distributed commit protocols are a major source of inefficiency in real-time systems [22], such an approach is expected to yield better performance.

The last approach we intend to study is touched upon in [28]. This is a hybrid approach where either data objects or code can migrate while still retaining the semantics of STM. By allowing either code or data to migrate, we can choose a migration scenario that results in the least amount of communication overhead. For example, suppose we have a simple transactional program that increments the value of a shared variable X and stores the new value in the transactional store. Assume further that X is remote, using a data flow abstraction would necessitate two communication delays; one to fetch X from its remote location and the other to send it back once it has been incremented. Using a control flow abstraction in this case may be more efficient since it will only involve a single communication delay.

On the other hand, assume that several processes need access to a small data structure and that these processes are in roughly the same location and are far away from the data they need. Since communication delay depends on distances, it may make sense to migrate the data to the processes in this case rather than incur several long communication delays by moving the code to the data. In short, the choice of whether to migrate code or data can have a significant effect on performance. In [28], this is accomplished under programmer control by allowing an *on* construct which a programmer can use to demarcate code that

should be migrated. We intend to elaborate on this by coming up with solutions that would use static analysis at compile-time (or dynamically at run-time) to make decisions about which part of the application to move using a number of heuristics such as, for example, size of code/data and locality considerations.

4.2 Designing suitable protocols and algorithms.

The algorithms and protocols that need to be designed depend on the programming abstraction we choose to implement. Some of the necessary abstractions have been touched upon in Section 4.1, here we elaborate on these points.

For the model where code migrates, creating cross-node transactions, and data is immobile, the main abstraction that needs to be designed is a real-time distributed commit protocol. Since cross-node transactions are permitted, with each node involved hosting part of the transaction, a distributed commit protocol is necessary to ensure atomicity. A number of distributed commit protocols have been studied in the literature, with the two phase commit protocol being the most commercially successful protocol. Unfortunately, the blocking semantics of the two phase commit protocol may not be very suitable for real-time systems. Therefore alternatives like the three phase commit protocol (despite its larger overhead) may be more appropriate due to its non-blocking semantics. Other alternatives that involve the relaxation of certain properties of distributed commit protocols in order to improve efficiency are discussed in [22]. We intend to design distributed commit protocols whose timeliness behavior can be quantified theoretically and/or empirically, in order to allow the system to provide guarantees on end-to-end timeliness.

For the approach where code is immobile and data migrates, the most important protocol that needs to be designed is a distributed real-time cache coherence protocol. This protocol needs to be location aware in order to reduce communication latency and should be designed to reduce network congestion. The cache coherence problem for multiprocessors has been extensively studied in the literature [35]. There are also some solutions for the distributed cache coherence problem (see [36–39] for a, not necessarily representative, sample of research on this issue). Distributed cache coherence bears some similarity to distributed hash table (or DHT) protocols which have been an active topic of research recently due to the popularity of peer-to-peer applications. Examples of DHT algorithms that are of interest are [40–42].

We envision a cache coherence algorithm based on hierarchical clustering to reduce network traffic and path reversal to synchronize concurrent requests, an approach used in [26]. Other approaches for implementing distributed cache coherence will also be considered. An important part of our research in this area will be to design cache coherence protocols that can provide timeliness guarantees that we can verify theoretically and empirically.

For the hybrid abstraction, where both code and data can move, several issues need to be determined. Among the issues that need to be resolved are the different methods of distributing transactional meta-data in order to ensure efficient execution of the STM system, providing a mechanism to support atomic

commitment when code is allowed to migrate thus resulting in multi-node transactions, aggregating communication in order to reduce the effect of the extra communication necessary to manage the STM system (possibly by piggybacking this information over normal network traffic) and optimizing network communication to reduce latency. It is also necessary to design appropriate mechanisms for choosing whether data or code migration is going to occur. Currently, the choice of which part of the program to migrate is performed under programmer control [28]. We intend to design automated methods for deciding which part of the program moves through either compile-time analysis or at run-time.

4.3 Programming language implementation.

We need to incorporate the programming abstraction chosen and the protocols and algorithms necessary to support them into a suitable programming language. Issues that need to be addressed are extending the programming language syntax to include support for higher level abstractions built upon STM. We introduce a number of syntactic modifications to support the new constructs we propose to implement. The most basic syntactic extension required is a method for demarcating atomic blocks (i.e. blocks of code that will be executed within the context of STM), additions such as programmer controlled retry and providing alternative transactional execution can also be considered.

In addition to these syntactic extensions, modifications to the run-time environment are also required. Our top candidate for implementing these abstractions is the emerging DRTSJ RI. We choose this language for a number of reasons. First, the language is still under development with a substantial part of the implementation details coming out of our research group. Second, the RI will be evaluated by the standard's expert community (e.g., Sun's JSR-50 experts group in the case of DRTSJ) as part of the standard's approval process, resulting in immediate and invaluable user feedback. Third, using a garbage collected language alleviates some of the issues involved in memory management associated with STM (by, for example, eliminating the problem of having transactions free allocated memory explicitly while other transactions are still working on it). Of course this necessitates augmenting the garbage collector with information about STM in order to prevent harmful interference with STM's meta-data.

Naturally, the actual modifications made to the programming language will depend on the programming abstraction chosen. Regardless of the choice made about the abstraction used to incorporate STM in distributed embedded systems, modifications to the run-time environment are necessary to support STM. The actual modifications made are dependent on the particular design we choose for our implementation of STM and so will not be elaborated upon in this position paper. However, some of the design issues involved are choosing appropriate meta-data to represent STM objects, providing appropriate mechanisms to atomically commit transactions (for example by using atomic hardware instructions such as compare-and-swap, or CAS, on suitably indirected meta-data), providing implementations for the different design choices of STM (e.g., visible reads versus invisible reads and weak versus strong atomicity).

4.4 Scheduling algorithms and analysis.

Finally, we will design scheduling algorithms that allow systems programmed using STM to meet end-to-end timeliness requirements. This is a challenge due to the fact that the retry behavior of STM is antagonistic to predictability. There have been several attempts at providing timing assurances when STM is used in real-time systems or when lock-free data structures are used in real-time systems [18, 29–31]. These approaches only consider uni-processor systems and use the periodic task arrival model to bound retries.

Some of the approaches are fairly sophisticated and use, for example, linear programming [30] to derive schedulability criteria for lock-free code. The basic idea of these approaches is that, on a uni-processor system, the number of retries is bounded by the number of task preemptions that occur. This bound exists because a uni-processor can only execute one process at a time. Since it is not possible for a process to perform conflicting operations on shared memory, and hence cause the retry of another process, unless it is running, the number of preemptions naturally bounds the number of retries on uni-processors. Given this premise, the analysis performed in [18, 29–31] bounds the number of retries by bounding the number of times a process can be preempted under different scheduling algorithms. This analysis allows the authors to derive schedulability criteria for different scheduling algorithms based on information about process execution times, execution times of the retried code sections, process periods, etc.

More recently, attempts have been made at providing timeliness guarantees for lock-free data structures built on multiprocessor systems [32]. The approach used in [32] is suitable for Pfair-scheduled systems and other multiprocessor systems where quantum-based scheduling is employed. The most restrictive assumption made in this approach is that access to a shared lock-free object takes at most two quanta of processor time. Using this assumption, the authors go on to bound the number of retries by determining the worst-case number of accesses that can occur to a shared object during the quanta in which it is being accessed. For an M processor system, the worst-case number of processes that can interfere with access to a particular shared object is $M - 1$. Given an upper bound on the number of times a process can access a shared object within a quanta, it is possible to derive an upper bound on the number of retries in such a system. The authors also go on to describe how it is possible to use the concept of a "supertask", basically a single unit that is composed of several tasks that are to be scheduled as one unit, to reduce the worst-case number of retries and hence improve system performance.

The particular method used to bound the number of retries in the system(s) we develop will depend on the model we target. There are two possible alternatives. The first approach is to target uni-processor distributed systems. In such systems, each node has only one processor. In order to provide scheduling criteria for such systems, we would use the approaches developed for uni-processor systems to derive the number of retries that can occur on each node, and then combine these bounds to determine the number of retries that can

occur to cross-node transactions, thus deriving schedulability criteria for STM implementations.

The second approach is to consider multiprocessor distributed systems. In such systems, each node is a multiprocessor or multi-core machine. Schedulability analysis and scheduling algorithms for such systems are considerably more difficult due to the difficulty in deriving bounds on the number of retries in the system. A first possible approach is to consider the Pfair-scheduling algorithm considered in [32] for obtaining bounds on the number of retries on each node and then combining these bounds to obtain bounds for cross-node transactions. Other approaches will also be considered in order to reduce the number of assumptions made on the system model. We will design scheduling algorithms that can ensure that timeliness requirements are not violated by the retry behavior of STM on distributed systems, and provide analytical expressions for the schedulability criteria of these scheduling algorithms.

5 Conclusions

Programming distributed systems using lock-based concurrency control is semantically difficult and computationally expensive. In order to alleviate some of these problems, we propose the use of STM for concurrency control. In order to achieve this goal, a number of issues need to be addressed. This position paper outlines these issues and proposes a method for solving them.

Three different abstractions for incorporating STM into distributed embedded real-time systems are mentioned, and the algorithms and protocols necessary for implementing these abstractions are briefly outlined. We also briefly indicate the type of schedulability analysis that will be required to provide timeliness guarantees for systems programmed using these abstractions.

References

1. Cares, J.R.: Distributed Networked Operations: The Foundations of Network Centric Warfare. iUniverse, Inc. (2006)
2. Lee, E.A.: The problem with threads. Computer **39**(5) (2006) 33–42
3. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture. (1993)
4. Knight, T.F.: An architecture for mostly functional languages. In: Proceedings of ACM Lisp and Functional Programming Conference. (Aug 1986) 500–519
5. Shavit, N., Touitou, D.: Software transactional memory. In: PODC. (1995) 204–213
6. Wikipedia: Software transactional memory — wikipedia, the free encyclopedia (2008) `http://en.wikipedia.org/w/index.php?title=Software_transactional_memory&oldid=213906392`, [Online; accessed 24-May-2008].
7. OMG: Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group (September 2001)
8. Pardo-Castellote, G.: Omg data-distribution service: Architectural overview. ICDCSW **00** (2003) 200

9. Shih, C., Stankovic, J.A.: Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, Amherst, MA, USA (1990)
10. Roesler, M., Burkhard, W.A.: Resolution of deadlocks in object-oriented distributed systems. IEEE Trans. Comput. **38**(8) (1989) 1212–1224
11. de Mendívil, J.R.G., Federico Fari n., Garitagoitia, J.R., Alastruey, C.F., Bernabeu-Auban, J.M.: A distributed deadlock resolution algorithm for the and model. IEEE Trans. Parallel Distrib. Syst. **10**(5) (1999) 433–447
12. Kshemkalyani, A.D., Singhal, M.: A one-phase algorithm to detect distributed deadlocks in replicated databases. IEEE Trans. on Knowl. and Data Eng. **11**(6) (1999) 880–895
13. de Mendivil, J.R.G., Demaille, A., Auban, J.B., Garitagoitia, J.R.: Correctness of a distributed deadlock resolution algorithm for the single request model. In: PDP '95: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (1995) 254
14. Elmagarmid, A.K.: A survey of distributed deadlock detection algorithms. SIGMOD Rec. **15**(3) (1986) 37–45
15. Mitchell, D.P., Merritt, M.J.: A distributed algorithm for deadlock detection and resolution. In: PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1984) 282–284
16. Choudhary, A.N., Kohler, W.H., Stankovic, J.A., Towsley, D.: A modified priority based probe algorithm for distributed deadlock detection and resolution. IEEE Trans. Softw. Eng. **15**(1) (1989) 10–17
17. Cho, H., Ravindran, B., Jensen, E.D.: Space-optimal, wait-free real-time synchronization. IEEE Transactions on Computers **56**(3) (2007) 373–384
18. Anderson, J., Ramamurthy, S., Moir, M., Jeffay, K.: Lock-free transactions for real-time systems. In: Real-Time Databases: Issues and Applications, Amsterdam: Kluwer Academic Publishers. (1997)
19. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. icdcs **00** (2003) 522
20. Zhao, Y., Lee, E.A., Liu, J.: Programming temporally integrated distributed embedded systems. Technical Report UCB/EECS-2006-82, EECS Department, University of California, Berkeley (May 2006)
21. Zhao, Y., Liu, J., Lee, E.A.: A programming model for time-synchronized distributed real-time systems. In: RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (2007) 259–268
22. Gupta, R., Haritsa, J., Ramamritham, K., Seshadri, S.: Commit processing in distributed real-time database systems. Real-Time Systems Symposium, 1996., 17th IEEE (4-6 Dec 1996) 220–229
23. Marathe, V.J., Scott, M.L.: A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept. (Jun 2004)
24. Bobba, J., Rajwar, R., Hill, M.: Transactional memory biblography `http://www.cs.wisc.edu/trans-memory/biblio/swtm.html`.
25. Korenfeld, B., Medina, M.: Transactional memory. Technical Report MIT/LCS/TM-475, University of Tel-Aviv Computer Engineering Dept. (Jun 2006)
26. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing **20**(3) (2007) 195–208

27. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP '06. ACM Press (Mar 2006) 198–208
28. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP '08, New York, NY, USA, ACM (2008) 247–258
29. Manson, J., Baker, J., Cunei, A., Jagannathan, S., Prochazka, M., Xin, B., Vitek, J.: Preemptible atomic regions for real-time java. RTSS **0** (2005) 62–71
30. Anderson, J., Ramamurthy, S.: A framework for implementing objects and scheduling tasks in lock-free real-time systems. In: Proceedings of IEEE RTSS, IEEE (dec 1996) 92–105
31. Anderson, J., Ramamurthy, S., Jeffay, K.: Real-time computing with lock-free shared objects. In: Proceedings of IEEE RTSS, IEEE Computer Society Press (December 1995) 28–37
32. Holman, P., Anderson, J.H.: Supporting lock-free synchronization in pfair-scheduled real-time systems. J. Parallel Distrib. Comput. **66**(1) (2006) 47–67
33. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Object-Oriented Programming, Systems, Languages, and Applications. (Oct 2003) 388–402
34. Hoare, C.: Towards a theory of parallel programming. In Hoare, C., Perrott, R., eds.: Operating System Techniques, Academic Press (1972) 61–71
35. Stenström, P.: A survey of cache coherence schemes for multiprocessors. Computer **23**(6) (1990) 12–24
36. Chang, Y., Bhuyan, L.N.: An efficient tree cache coherence protocol for distributed shared memory multiprocessors. IEEE Transactions on Computers **48**(3) (1999) 352–360
37. Tamir, Y., Janakiraman, G.: Hierarchical coherency management for shared virtual memory multicomputers. Journal of Parallel and Distributed Computing **15**(4) (1992) 408–419
38. Aguilar, J., Leiss, E.L.: A general adaptive cache coherency-replacement scheme for distributed systems. In: IICS '01: Proceedings of the International Workshop on Innovative Internet Computing Systems, London, UK, Springer-Verlag (2001) 116–125
39. Kent, C.A.: Cache coherence in distributed systems. WRL Technical Report 87/4 (1987)
40. Hildrum, K., Krauthgamer, R., Kubiatowicz, J.: Object location in realistic networks. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM (2004) 25–35
41. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM (1997) 311–320
42. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, London, UK, Springer-Verlag (2001) 329–350
43. Jensen, D., Wells, D.: A framework for integrating the real-time specification for java and java's remote method invocation. In: ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, IEEE Computer Society (2002) 13