This work-in-progress paper was presented as part of the main technical program at IEEE ETFA'2011

Towards Resource Sharing by Message Passing among Real-Time Components on Multi-cores*

Farhang Nemati, Rafia Inam, Thomas Nolte and Mikael Sjödin
Mälardalen Real-Time Research Centre, Mälardalen University, Sweden
Email: {farhang.nemati, rafia.inam, thomas.nolte, mikael.sjodin}@mdh.se

***Abstract*—In this paper we propose a message passing synchronization protocol for resource sharing among real-time applications on multi-core platforms where each application is allocated on a cluster of cores. In this protocol the resources that are only used within an application (local resources) are handled by shared memory synchronization while the resources shared cross applications (global resources) are accessed by means of message passing. In our protocol the global resources are safely accessed without requiring to lock the resources explicitly. The goal is to avoid resource locking using shared memory, since accessing shared memory in multi-cores is very time consuming, whereas message passing has the potential to be much more efficient in systems with deep memory hierarchies.**

I. Introduction

The availability of multi-core platforms has attracted much attention in multiprocessor embedded software analysis, run-time techniques, policies, and protocols. As the multi-core architectures are to be the defacto processors within a near future, the industry must cope with a potential migration of existing systems towards multi-core platforms.

Co-executing real-time applications on a shared multi-core platform, where each application is statically allocated on a dedicated sub set of cores (cluster), requires to overcome the problem of handling mutually exclusive shared resources among those applications. In the real-time community synchronization on mutually exclusive resources mostly has focused on shared memory synchronization (e.g., by means of binary semaphores). Message passing techniques for synchronization offer many advantages, e.g., they provide more isolation of the applications (no need for shared memory), message passing is a common technique for communications in commercial real-time operating systems, and resource locking using shared memory in multi-cores is time consuming.

Recently, in industry, co-executing of multiple applications on a multi-core platform (using virtualization techniques) has been considered to reduce the overall hardware costs [1]. In the virtualization techniques usually each application owns its own memory space and the applications do not share memory, hence message passing synchronization among those applications is a natural fit.

In the domain of distributed systems *message passing techniques* for handling mutually exclusive resources are well developed and used. Numerous algorithms have been proposed since the introduction of ordering of the events by Le Lann [2] and Leslie Lamport [3], to solve mutual exclusion problem through sending messages in the loosely-coupled distributed system where processes (applications) do not share a common memory. Two different families of these algorithms (token-based and permission-based algorithms) are commonly used for accessing shared resources. However, these techniques only consider satisfying *safety* (only one process can access the shared resource) and *liveness* (each request must be granted) properties. To our knowledge, scheduling analysis regarding real-time properties has not been considered in those techniques.

In this paper we propose a synchronization protocol for handling resource sharing among real-time applications on multi-core platforms by means of message passing and provide the real-time guarantees for this method. The main objective of our work is to avoid resource locking using shared memory as accessing shared memory in multi-cores is very time consuming, while message passing has the potential to be much more efficient in systems with deep memory hierarchies. In addition, message passing synchronization will provide better memory isolations among real-time applications.

Related Work: There are two major approaches for scheduling real-time systems on multiprocessors (multi-cores); global and partitioned scheduling [4], [5]. Under global scheduling, tasks are scheduled by a single (global) scheduler and any task can be scheduled to execute on any processor. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol. The generalization of global and partitioned scheduling algorithms is called clustered scheduling [6], [7], in which tasks are statically assigned to a sub set (a cluster) of processors, and within each cluster tasks are scheduled using a global scheduling algorithm.

In the context of shared memory locking protocols under multiprocessors, a non-exhaustive list of work includes [8], [9], [10], [11], [12], [13].

Recently we have proposed a shared memory locking protocol [14] for handling resource sharing among real-time applications on multiprocessors. In this work each application is represented by an interface which abstracts the resource requirements of the application.

We are not aware of any work regarding synchronization on multi-cores by message passing techniques where the real-time properties are considered.

* This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

978-1-4577-0018-7/11/$26.00

II. Message Passing Techniques for Mutual Exclusion under Distributed Systems

The techniques for handling of sharing mutually exclusive resources in loosely-coupled distributed systems are divided in two general categories; token-based and non-token-based (permission-based) approaches. In this section we briefly describe the two traditional message-passing approaches for employing mutual exclusion algorithms in the distributed systems community. These two principles characterize two families of these algorithms:

A. Token-based Algorithms

In this approach a unique token is shared among the processes (applications) of the whole system, the process that owns the token is granted the access to the resource. Hence the mutual exclusion is guaranteed, i.e., safety property is satisfied. A token-based algorithm can be implemented in either of the following forms: (i) The token moves in a directed ring; a process can access the resource when it receives the token, and if the process has not requested the resource, it simply forwards the token to the next process [2], [15]. (ii) The process requesting a resource requests for the token (the token does not move). If the process owns the token, it accesses the resource, otherwise it has to wait until the token is granted [16].

B. Permission-based Algorithms

In this approach, whenever a process requests a resource, it requires permission from all other processes and waits until the permission is granted [17]. In Permission-based algorithm, usually a time-stamping mechanism is used to attain such a permission. Many improvements have been suggested specially the use of acyclic directed graph and the use of a logical tree structure [18]. To satisfy the safety property, enough number of permissions is taken from other processes. The liveness property is guaranteed by the total-ordered requests using timestamps or by using acyclic directed graph.

C. Complexity Analysis

In traditional distributed systems, the complexity analysis for message-passing algorithms is dependent on the following complexities:

Communication Complexity: The communication complexity is dependent on the total number of messages required per critical-section (CS) access, and synchronization messages, i.e., the total number of messages required before and after the accessing the shared resource to synchronize the processes.

Time Complexity: The total time interval (response time) a process waits from requesting for a shared resource until it accesses the resource.

III. System and Platform Model

In this paper we assume that a real-time application, denoted by A_k, is allocated on a dedicated cluster consisting of one or more identical, unit-capacity processors (cores). The real-time application consists of a set of real-time tasks. We assume that the tasks in application A_k are scheduled using a global scheduling if A_k is allocated on more than one core otherwise the tasks are scheduled using a uniprocessor scheduling processor.

Application A_k consists of a set of sporadic tasks, τ_i. The tasks in application A_k may share a set of mutually exclusive resources, R_{A_k}. The set of shared resources R_{A_k} consists of two sets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks within the same application while a global resource is shared by tasks from more than one application. In this paper, we focus on non-nested critical sections.

We also assume that the applications do not use (have) shared memory for synchronization on global resources and message passing is the only way of communication among the applications.

We assume that sharing the local resources is handled by shared memory synchronization protocols. However, as an application is allowed to be allocated on more than one core, applications will benefit from handling sharing their local resources by means of message passing as well. In this paper we only consider the partial problem regarding the global resources.

IV. The Synchronization Protocol

We have provided our proposed protocol, for synchronization on global resources among real-time applications on a multi-core platform, with two alternatives. The first alternative is similar to permission-based algorithms in distributed systems, however our algorithm is much simpler with regards to the number of messages (actually an application sends out only one request message any time it requests a resource). We call this alternative as *time out algorithm*. The second alternative is similar to the token-based algorithms in the domain of distributed systems. We call the second alternative as *token-based algorithm*

To be able to guarantee schedulability and consistency in accessing the mutually exclusive resources shared among such applications, for each application *Resource Hold Times* (RHT's) of the global resources it shares have to be calculated and provided in the abstraction of the application.

Definition 1: *Resource Hold Time* of a global resource R_q by application A_k is denoted by $RHT_{q,k}$ and is the maximum duration of time that the global resource R_q can be held by any task in A_k.

The calculation and minimization of resource hold times for real-time applications on a shared uniprocessor platform has been presented in [19], [20]. We have extended the calculation and minimization of RHT's to multi-core platforms in a recent work [21].

A. Time Out Algorithm

In this approach any application A_k contains a request queue $RequestQ_{q,k}$ for each global resource R_q it shares where it maintains incoming requests to access R_q from other

applications.A request from an application A_i for accessing resource R_q is denoted by $RQ_{q,i}$. Whenever application A_i starts using global resource R_q, it will hold it for at most $RHT_{q,i}$ time units (Definition 1). Thus A_k will remove the corresponding request ($RQ_{q,k}$) from $RequestQ_{q,k}$ after $RHT_{q,i}$ time units (by setting a timer).

Furthermore, application A_k contains a data structure to maintain the information from other applications regarding global resources, i.e., a table called $RHTTable_k$ that contains the RHT's of other applications for the resources they share with A_k.

Request Rules

Rule 1: Whenever any task τ_i in application A_k requests a global resource R_q, it time stamps a request message $RQ_{q,k}$ and sends it to all applications sharing R_q. Time stamping is applied to order incoming request messages in the request queues in correct order. However, it may happen that more than one application issue requests at the same time (i.e., at the same tick). This means that multiple requests may have the same time stamp. Considering this, the incoming requests have to be ordered by their time stamps first and the requests containing the same time stamp will be ordered by the index of the applications issuing the requests.

If $RequestQ_{q,k}$ is empty (i.e., no other application has requested R_q), τ_i can access R_q. All recipient applications add $RQ_{q,k}$ to their request queue for R_q. After $RHT_{q,k}$ time units all recipient applications as well as A_k will remove $RQ_{q,k}$ from their corresponding request queue.

In this paper we assume atomic message transmit. However, in reality it may happen that there is a delay between the time a request message is issued and the time the message is received. In this case to guarantee mutual exclusion, any application whose request for a global resource is at the top of its corresponding request queue has to wait for up to $WCMT$ time units where $WCMT$ denotes the worst case time that may take for a message to be received from the time it is issued. An alternative solution could be to exchange more messages similar to permission-based approaches in distributed systems.

Rule 2: If $RequestQ_{q,k}$ is not empty (i.e., some applications have already requested R_q), all recipient applications as well as A_k add $RQ_{q,k}$ to their request queues for R_q. In this case τ_i suspends.

Rule 3: Any request $RQ_{q,l}$ at the top of $RequestQ_{q,k}$ will be removed after $RHT_{q,l}$ time unites (the RHT values are extracted from $RHTTable_k$). When a request of A_k is at the top of $RequestQ_{q,k}$, the eligible task (e.g., the highest priority task waiting for R_q) in A_k can access R_q and after $RHT_{q,l}$ time units A_k's request is removed from $RequestQ_{q,k}$. When the task releases R_q, no more tasks in A_k are allowed to access R_q even if $RQ_{q,k}$ is not yet removed and there exists tasks in A_k waiting for R_q. This limitation is applied to restrict any application not to hold a global resource more than its corresponding RHT. This process will continue as long as $RequestQ_{q,k}$ is not empty.

Rule 4: At some point of time if multiple request queues in A_k contain requests from A_k at their tops (i.e., A_k is allowed to access multiple global resources at the same time), the corresponding resources will be accessed in the order of the time stamps in the requests and for the requests with equal time stamps the requests are ordered based on application indexes.

B. Token-based Algorithm

In the token-based approach, for each global resource R_q there exists a unique token TK_q and the application that holds the token is allowed to access R_q. Each token TK_q contains a request queue where the requests of applications requesting the token are located. Each application contains a queue denoted by GQ_q, where it maintains its tasks requesting R_q.

Request Rules

Rule 1: Whenever any task τ_i in application A_k requests a global resource R_q, if A_k holds TK_q and the request queue of TK_q is empty, τ_i accesses R_q. If the request queue of TK_q is not empty (which means that another task within A_k is accessing R_q), τ_i will be added to the request queue of TK_q and to GQ_q, and suspends. If A_k does not hold TK_q it will add τ_i to GQ_q and sends a request message $RQ_{q,k}$ to all applications sharing R_q. The request message is ignored by all applications except the one that holds the token.

Rule 2: When an application A_i holding token TK_q receives a request message it adds the request to the request queue of TK_q.

Rule 3: While task τ_i within application A_k is accessing R_q, it may happen that more tasks within A_k request R_q. In this case these tasks will be added to GQ_q and for each of the tasks a request is added to token's request queue.

Rule 4: When task τ_i within application A_k releases resource R_q (i.e., exits from the corresponding critical section), A_k removes its request from the top of the request queue of TK_q. Token TK_q will be forwarded to the application whose request is at the top of the token's request queue. It may happen that after τ_i has released R_q the request queue of TK_q is empty. In this case TK_q is called inactive and remains in A_k until a request message requesting TK_q comes in.

Rule 5: If at some point of time application A_k holds multiple tokens, the corresponding resources are accessed in the order of the time stamps of the requests and for the requests with equal time stamps access will occur in the order of application indexes.

C. Remarks

Here we point out some remarks regarding the efficiency of our proposed algorithms.

Communication complexity: The time out algorithm has much less number of messages for handling mutual exclusion as compared to the permission-based algorithms in distributed systems. In a permission-based algorithm, generally at least one round of messages are exchanged; the node (application) requesting for a resource sends a request message to all nodes sharing the resource and waits for permission message (request grant) from all the other nodes. Furthermore, after each release of the resource a release message is sent to

all other nodes that share the resource. On the other hand, in the time out algorithm whenever an application requests a resource it is only required to send a request message to the applications sharing the resource and it does not need to wait for the grant messages neither does it need to send any release message to other applications. However, setting a timer for removing a request at the top of the request queue will introduce overhead. The time out algorithm will perform well when it is used on multi-cores with a single clock, however when applications are executing on a multi-core with more than one clock, the clock synchronization may also increase the communication complexity. In the token-based algorithm, less messages are exchanged as compared to the token-based algorithms in distributed systems.

Run time complexity: In the time out algorithm, the run time performance can be increased as any application A_k always holds a resource R_q for $RHT_{q,k}$ time units while in the permission-based algorithms the resource becomes available to other applications as soon as it is released by A_k. From the schedulability analysis point of view this is not a drawback as in the schedulability analysis the worst case has to be considered where application A_k holds any global resource R_q for $RHT_{q,k}$ time units. Our token-based algorithm however, does not have this drawback as it forwards the token to the next requesting application as soon as the resource is released by the application holding the token.

Comparison of time out and token-based algorithms: The time out algorithm suffers from the overhead of timers as well as always holding a resource as long as its RHT value. Furthermore, in the time out algorithm each application contains a request queue for each global resource it shares and keeping the consistency among the request queues in different applications may also introduce overhead. On the other hand the token-based algorithm suffers from the overhead regarding the size of the tokens as they contain a request queue.

Message passing using shared global memory In this paper we have assumed that the applications do not have or use shared memory. However, today's multi-core platforms usually contain a global memory shared among all cores. In this case our both time out and token-based algorithms will be simpler and perform better as the request queues and the tables containing information regarding RHT's can be located in the shared memory. The size of tokens in the token-based algorithm will be reduced to the size of a very short message and in the time out algorithm, the applications will not suffer from multiple request queues and timers.

V. Conclusions

In this paper we have proposed two algorithms (called time out and token-based) for synchronization on mutually exclusive resources among real-time applications. The applications communicate solely by message passing thus shared memory synchronization techniques cannot be used.

We have mentioned the advantages and disadvantages of each algorithm.We are currently working on deriving the detailed schedulability analysis of the algorithms as well as the formal proofs of the correctness of them.

In the future we will study the cases where the multi-core contains multiple clocks and investigate the clock synchronization methods and their effect on each algorithm. Another interesting future work is to implement both algorithms in a real-time operating system (RTOS) and study the performance of the algorithms. Furthermore, we will study the possibility of message passing synchronization via global shared memory which will simplify our proposed algorithms. An interesting future work is to compare the performance of message passing synchronization (with and without global shared memory) to the performance of shared memory locking mechanisms.

References

[1] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper*.

[2] G. Le Lann. Distributed systems - towards a formal approach. *Information of Processing*, pages 155–160, 1977.

[3] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of ACM*, 1978.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[5] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[6] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *ECRTS'07*, pages 247–258, 2007.

[7] Theodore P. Baker and Sanjoy K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*, 2007.

[8] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS'88*, 1988.

[9] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[10] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS'01*, pages 73–83, 2001.

[11] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA'07*, pages 47–56, 2007.

[12] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS'09*, pages 377–386, 2009.

[13] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS'10*, pages 49–60, 2010.

[14] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS'11, to appear*, 2011. (Available at www.mrtc.mdh.se).

[15] A.J. Martin. Distributed mutual exclusion on a ring of processors. *Science of computer Programming*, 25:256–276, 1985.

[16] M. Naimi and M. Trehel. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. In *Int. Phoenix IEEE conf. on Comp. and Comm.*, pages 35–39, 1987.

[17] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of ACM*, 24(1):9–17, 1981.

[18] D. Agrawala and A.E. Abbadi. An efficient solution to distributed mutual exclusion problem. In *Proc. of 8th ACM Symposium on PODC*, pages 193–200, 1989.

[19] N. Fisher, M. Bertogna, and S. Baruah. Resource-locking durations in edf-scheduled systems. In *RTAS'07*, pages 91–100, 2007.

[20] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *IPDPS'07 Workshops*, pages 1–8, 2007.

[21] F. Nemati and T. Nolte. Resource hold times under multiprocessor static-priority global scheduling. In *RTCSA'11 , to appear*, 2011. (Available at www.mrtc.mdh.se).