Chapter 1

Conclusions, Contributions, and Proposed Post Preliminary-Exam Work

We consider STM for concurrency control in multicore real-time software. Doing so will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy time constraints. Retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are scheduler-dependent). Thus, real-time CM is logical.

We investigate and design a number of real-time CMs. The first two CMs are directly based on dynamic and static priority of underlying tasks. Earliest Deadline-First CM with G-EDF scheduler (ECM) resolves conflicts based on absolute deadline of the underlying instances. Rate Monotonic Assignment with G-RMA scheduler (RCM) resolves conflicts based on period of underlying instances. We analyze retry cost and response time under ECM and RCM. We identify the conditions under which ECM and RCM have better schedulability than lock-free. Experiments show shorter retry cost for ECM and RCM than lock-free.

ECM and RCM conserve the semantics of the underlying real-time scheduler. This conservative approach results in a maximum retry cost- for a single transaction due to another transaction- of double the maximum atomic section length among all tasks. So, another CM is developed to reduce this retry cost. Length-based CM (LCM) considers not only static/dynamic priority of underlying instance, but also length of the interfering transaction compared to remaining length of interfered transaction. LCM is used with G-EDF and G-RMA. Although it can reduce retry cost, but it suffers from priority inversion. By proper choice of different parameters, additional cost due to priority inversion is bounded. Thus, the net result will be lower response time for tasks using LCM with G-EDF/G-RMA. We analyze retry cost and response time of LCM. We identify the conditions under which LCM has better schedulability than ECM, RCM and lock-free. Experiments show shorter retry

cost than ECM, RCM and lock-free.

ECM, RCM and LCM are affected by transitive retry. Transitive retry enforces a transaction to abort and retry due to another non-conflicting transaction. Transitive retry appears when multiple objects exist per transaction. So, we develop the Priority-based with Negative value and First access (PNF) contention manager. PNF avoids transitive retry and deals better with multiple objects than previous contention managers. PNF also tries to optimize processor usage by lowering priority of the job underlying retrying transaction. Thus, other jobs can proceed if there is no conflict. We upper bound retry cost and response time for PNF when used with G-EDF and G-RMA. Schedulability is compared between PNF on one side and ECM, RCM, LCM and lock-free on the other. Experiments show better retry cost for PNF than other competitors as long as transitive retry and contention exist.

1.1 Contribution

We design and investigate a number of contention managers that try to preserver real-time constraints besides data accuracy. Designing CMs is straightforward. The simplest logic is to keep the rational of the underlying real-time scheduler. This was shown in ECM and RCM. ECM allows transaction with earliest absolute deadline (dynamic priority) to commit first. RCM allows transaction with smallest period (fixed priority) to commit first. We derived upper bounds for retry cost and response time under both ECM and RCM. Lock-free schedulability was compared to schedulability of ECM and RCM. Under both ECM and RCM, a task incurs $2.s_{max}$ retry cost for each of its atomic sections due to a conflict with another task's atomic section. Retries under RCM and lock-free are affected by a larger number of conflicting task instances than under ECM. While task retries under ECM and lock-free are affected by all other tasks, retries under RCM are affected only by higher priority tasks.

STM and lock-free have similar parameters that affect their retry costs—i.e., the number of conflicting jobs and how many times they access shared objects. The s_{max}/r_{max} ratio determines whether STM is better or as good as lock-free. For ECM, this ratio cannot exceed 1, and it can be 1/2 for higher number of conflicting tasks. For RCM, for the common case, s_{max} must be 1/2 of r_{max}, and in some cases, s_{max} can be larger than r_{max} by many orders of magnitude.

We present Length-based contention manager (LCM) that is used with G-EDF and G-RMA. LCM tries to compromise between priority of transactions (which is priority of the underlying task), and remaining execution time of interfered transaction. As the remaining execution time of the interfered transaction decreases, it will be useless to abort it while it can shortly commit. To abort the interfered transaction or not, is determined by α and ψ parameters. α ranges between 0 and 1. When $\alpha \to 0$, LCM acts in a first-in-first-out manner. When $\alpha \to 1$, G-EDF/LCM acts like ECM, and G-RMA/LCM acts like RCM. We derived upper bounds on

retry cost and response time under LCM. We also compared schedulability of LCM against ECM, RCM and lock-free. We identified the conditions under which LCM performs better than the other synchronization techniques. LCM reduces retry cost of each atmic section to $(1+\alpha_{max})s_{max}$ instead of $2.s_{max}$ in case of ECM and RCM. In ECM and RCM, tasks do not retry due to lower priority tasks, whereas in LCM, they do so. In G-EDF/LCM, retry due to a lower priority job is encountered only from a task τ_j's last job instance during τ_i's period. This is not the case with G-RMA/LCM, because, each higher priority task can be aborted and retried by any job instance of lower priority tasks. Schedulability of G-EDF/LCM and G-RMA/LCM is better or equal to ECM and RCM, respectively, by proper choices for α_{min} and α_{max}. Schedulability of G-EDF/LCM and G-RMA/LCM is better or equal to lock-free synchronization as long as s_{max}/r_{max} does not exceed 0.5. By proper choice of αs, s_{max}/r_{max} can be increased to 2 under G-EDF/LCM, and to large values under G-RMA/LCM.

ECM, RCM and LCM suffer from transitive retry in case of multi-objects per transaction. So, we introduced Priority-based with Negative value and First access (PNF) contention manager. PNF avoids transitive retry effect suffered by ECM, RCM and LCM in case of multiple objects per transaction. PNF tries to optimize processor usage by reducing priority of aborted transaction. This way, other tasks can proceed if they do no conflict with other executing transactions. PNF implementation is not as simple as other CMs. We upper bounded PNF retry cost and response time. We compared PNF schedulability to other competitors to know when one is preferred to the others. PNF has better schedulability than lock-free as long as s_{max} does not exceed r_{max}. Experiments show longer retry cost for lock-free than PNF. For high contention and transitive retry, experiments show shorter retry cost of PNF than other CMs.

1.2 Post Preliminary-Exam Work

We propose the following post preliminary exam work: *Supporting nested transactions.* Transactions can be nested *linearly*, where each transaction has at most one pending transaction [14]. Nesting can also be done in *parallel* where transactions execute concurrently within the same parent [18]. Linear nesting can be *1) flat:* If a child transaction aborts, then the parent transaction also aborts. If a child commits, no effect is taken until the parent commits. Modifications made by the child transaction are only visible to the parent until the parent commits, after which they are externally visible. *2) Closed:* Similar to *flat nesting*, except that if a child transaction conflicts, it is aborted and retried, without aborting the parent, potentially improving concurrency over flat nesting. *3) Open:* If a child transaction commits, its modifications are immediately externally visible, releasing memory isolation of objects used by the child, thereby potentially improving concurrency over closed nesting. However, if the parent conflicts after the child commits, then compensating actions are executed to undo the actions of the child, before retrying the parent and the child. We propose to develop real-time contention managers that allow these different nesting models and es-

tablish their retry and response time upper bounds. Additionally, we propose to formally compare their schedulability with nested critical sections under lock-based synchronization. Note that, nesting is not viable under lock-free synchronization.

Generally, previous contention managers can be directly extended for different types of nesting. On conflict, the aborted transaction should be specified. In case of flat nesting, it is the outer parent transaction that should abort and restart. In case of closed and open nesting, only the interfered transaction should be restarted. In flat nesting, it will be useful to delay a lower priority transaction when it interferes with a higher priority one. Thus, it will not be required to restart the lower priority transaction from the beginning. This can reduce retry cost especially when the inner most transaction is the conflicting one.

ECM and RCM can use the same criteria to determine which transaction to abort or wait. LCM may need a redefinition of α parameter. Each child transaction can have its own α. So, for each depth of nesting, there is a corresponding α. Alternatively, there can be only one α for the outer transaction. Inner transactions do not have their αs. The first choice may be suitable for closed and open nesting, while the second choice seems more suitable for flat nesting. PNF can be used directly with all types of nesting. The only requirement is that each transaction checks conflict against itself, as well as its inner transactions. Executing transactions under PNF are non-preemptive. Thus, any nested transaction will not be aborted. This requirement for PNF simplifies implementation, but makes no use of nesting structure (i.e., PNF deals with transactions as if they are not nested). The previous requirement for PNF can be alleviated by checking conflict of any inner transaction only when this inner transaction begins. So, when a transaction starts, it checks its own objects- not inner transactions- against objects of executing transactions. If no conflict found, transaction executes. Consequently, when an inner transaction starts, it can find conflict with already executing transactions. Thus, this inner transaction should wait until the other conflicting executing transactions commit. This choice for PNF implementation may add additional waiting time to inner transactions. Thus, it is required to make tradeoffs between different choices.

Worst case response time analysis, just as design of CMs, needs modification to cope with nested transactions. The simplest method is to combine all accessed objects in all nested transactions in one group. Then, to act as if this group is accessed by only one non-nested transaction. This simplifies calculations but looses upper bounds, especially with closed and open nesting. As closed and open nesting only restarts the inner aborted transaction, it is no use to include the length of the outermost parent in calculations.

Combinations and optimizations of LCM and PNF contention managers. Current implementation of PNF is centralized. This implementation acts as a centralized contention manager that controls all transactions. Contention managers are decentralized as said in Section **??**. Each transaction should maintain its own contention manager as this reduces overhead and transaction blocking. Thus, one future trend it to make PNF decentralized. Besides, current implementation of PNF uses locking which increases overhead. Additional trend is to use

lock-free PNF implementation.

LCM is designed to reduce the retry cost of a transaction when it is interfered close to the end of its execution. In contrast, PNF is designed to avoid transitive retry when transactions access multiple objects. An interesting direction is to combine the two contention managers to obtain the benefits of both algorithms. One issue with this combination is that LCM allows abortion of running transactions depending on α value. PNF does not permit abortion of any executing transaction. Thus, the first attempt to combine LCM and PNF depends on the transitive retry level. If transitive retry level is high, then PNF will be used. Otherwise, LCM is used. Another way to combine PNF and LCM is to change α with time. Thus, at some time point, α can equal 0. This means the current transaction cannot be aborted by any other transaction. This is similar to executing transactions in PNF. But α should not be 0 all the time. Thus, α should change with time. Further design optimizations may also be possible to reduce retry costs and response times, by considering additional criteria for resolving transactional conflicts. Importantly, we must also understand what are the schedulability advantages of such a combined/optimized CM over that of LCM and PNF, and how such a combined/optimized CM behaves in practice. This will be our second research direction.

Formal and experimental comparison with real-time locking protocols. Lock-free synchronization offers numerous advantages over locking protocols, but (coarse-grain) locking protocols have had significant traction in real-time systems due to their good programmability (even though their concurrency is low). Example such real-time locking protocols include PCP and its variants [8, 12, 16, 17], multicore PCP (MPCP) [13, 15], SRP [1, 7], multicore SRP (MSRP) [10], PIP [9], FMLP [3, 4, 11], and OMLP [2]. FMLP has been established to be superior to other protocols [6]. How does their schedulability compare with that of the proposed contention managers? How do they compare in practice? These questions constitute our third research direction. Optimal mutliprocessor locking protocol (OMLP) [5] is similar to FMLP and simpler in implementation. OMLP do not require changes in G-EDF as FMLP. Under OMLP, each resource has a FIFO queue of length at most m, and a priority queue. Requests for each resource are enqueued in the corresponding FIFO queue. If FIFO queue is full, requests are added to the priority queue according to the requesting job's priority. The head of the FIFO queue is the resource holding task. Other queued requests are suspended until their turn come. OMLP achieves $O(m)$ priority inversion (pi) blocking per job under suspension oblivious analysis. In suspension oblivious, suspension time is added to task execution in contrast to suspension aware analysis. This is why OMLP is asymptotically optimal under suspension oblivious analysis. We intend to implement OMLP on ChronOS real-time OS. Then, analytically and experimentally compare OMLP against different CMs for both nested and non-nested transactions.

Bibliography

[1] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3:67–99, 1991.

[2] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, pages 119–128, 2007.

[3] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47 –56, aug. 2007.

[4] B.B. Brandenburg and J.H. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS-RT. In *RTCSA*, pages 185–194, 2008.

[5] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 49 –60, 30 2010-dec. 3 2010.

[6] Bjrn Brandenburg and James Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmus rt. In Theodore Baker, Alain Bui, and Sbastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 105–124, 2008.

[7] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[8] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990.

[9] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377 –386, dec. 2009.

[10] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189 – 198, may 2003.

[11] P. Holman and J.H. Anderson. Locking under pfair scheduling. *TOCS*, 24(2):140–174, 2006.

[12] D.K. Kiss. Intelligent priority ceiling protocol for scheduling. In *2011 3rd IEEE International Symposium on Logistics and Industrial Informatics*, LINDI, pages 105 –110, aug. 2011.

[13] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 469 –478, dec. 2009.

[14] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186 – 201, 2006.

[15] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, pages 116–123, 2002.

[16] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[17] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175 –1185, sep 1990.

[18] H. Volos, A. Welc, A.R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. *ECOOP 2009–Object-Oriented Programming*, pages 123–147, 2009.