2018 IEEE Real-Time Systems Symposium

Work-In-Progress: Lock-Based Software Transactional Memory for Real-Time Systems

Catherine E. Nemitz
Department of Computer Science
The University of North Carolina at Chapel Hill, USA
nemitz@cs.unc.edu

James H. Anderson
Department of Computer Science
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

***Abstract*—We propose a method for designing software transactional memory that relies on the use of locking protocols to ensure that transactions will never be forced to retry. We discuss our approaches to implementing this method and tunable parameters that may be able to improve schedulability on an application-specific basis.**

***Index Terms*—multiprocess locking protocols, nested locks, priority-inversion blocking, reader/writer locks, real-time locking protocols, software transactional memory**

I. Introduction

Software transactional memory (STM) is a mechanism used to coordinate access to shared-memory resources. STM provides a separation of concerns regarding these resources: application development and resource-access synchronization are isolated from each other. For developers of real-time applications, STM presents a simple interface for guaranteeing that shared memory is accessed in a safe manner; instead of resorting to a parallelism-restricting coarse-grained solution or reasoning through race conditions, potential memory locations being accessed, and devising a fine-grained approach, a developer can simply mark a section of code as `atomic`. Separately from this development phase, the STM ensures a section of code in execution will access shared memory in a safe manner based on the required access (read or write). How this is accomplished is discussed later.

In the context of real-time systems, this separation of concerns has benefits beyond the initial development of an application. With the use of a certified STM, the burden of certifying an application for a safety-critical domain is reduced. Managing access to shared memory in the application is abstracted away, which allows a certifying body to focus on the easier task of ensuring that all code blocks for which synchronization is necessary are marked `atomic`; the STM handles the rest.

STM implementations can be considered to have several interdependent parts. An STM allows reads and writes to shared memory, detects and resolves conflicts, and ensures progress for each process. In the model typically envisioned [3], [4], [6], [8], [9], [11], [12], [14]–[16], [18], [21], [25], [26], these goals are accomplished with attempted transactions, a runtime conflict-detection component, and transaction rollbacks and retries with some priority or contention management to ensure eventual completion.

Because of the focus on timing constraints within real-time systems, we propose a new solution to reduce these concerns to two components: a conflict-detection mechanism and a locking protocol that *guarantees* that each transaction completes on its first execution with no retries. We believe that this approach can lead to improved schedulability over more traditional approaches because of tighter bounds on worst-case transaction execution times that can be guaranteed. Our approach can support a range of applications, from embedded to high performance computing.

Our new lock-based approach differs from prior work as described in Sec. II. Also in Sec. II, we describe our system model. Then, we describe components of our proposed implementation in Sec. III and discuss extensions to our new implementation in Sec. IV before concluding in Sec. V.

II. Background and Prior Work

We begin by providing an explanation of our model of tasks and resources. We then discuss prior work.

A. System model

We assume the standard sporadic or periodic task system model, in which a task system Γ is comprised of tasks $\tau_1, \ldots, \tau_n$. These n tasks are scheduled with a job-level fixed-priority scheduler on a platform with m processors.

Each task may need to *access* some resource ℓ_a. (How a resource maps to shared memory is discussed in more detail in Sec. III.) This access may be a *write* access—one which requires mutual exclusion—or a *read* access—one which may occur concurrently with other read accesses. We focus on using a lock L to protect specific resources. We define the notation L_x:$\{y\}$ to represent that Lock x protects the set of resources y. That is, a task must first acquire L_x before it may access any resource in y.

B. Prior work

The idea of transactional memory was proposed two-and-a-half decades ago [16], and many approaches for implementing

Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

such a framework have emerged that focus on hardware-based, software-based, and hybrid solutions. Much of the prior work focuses on software transactional memory [26], as flexible hardware implementations are not commercially available [7]. Many types of STM have been explored. These include STM implementations that define resources by shared-memory locations (*e.g.*, [6], [14]) or by shared data structures (*e.g.*, [15]). Some STM approaches have relied on non-blocking or lock-free techniques to coordinate transaction commits and retries (*e.g.*, [15], [26]), while others have used locks to coordinate these decisions (*e.g.* [6], [11]). However, the approaches suitable for use in a real-time system cover a much smaller space.

Existing approaches to STM development in the context of real-time systems have focused on the traditional structure described above. Tasks do prospective work on copies of resources, and then the transaction attempts to commit these changes to the original resource(s), retrying if any conflicts are detected. The research focus on the use of STM for real-time applications is necessarily focused on ensuring guarantees of progress for each memory access in order to provide a system that is schedulable. To this end, work has been done that focuses on bounding and lowering the number of retries required.

The most similar approach to the one we propose is one in which a set of locks must be acquired for a given transaction to complete, but a transaction may be forced to give up its locks and re-acquire them based on a time-stamped ordering [3]. Prior work has also experimented with prioritizing each transaction based on the deadline of the task issuing the transaction [18], [21], [25]. Other work has investigated the effect of eager vs. lazy conflict detection [4] and developed a contention manager that limits the number of transactions executing simultaneously in order to be able to guarantee progress [8], [9].

In contrast, our proposed approach eliminates the need for retries.

Our lock-based implementation described below is different from other STM implementations that are called "lock-based" [6], [11]; this term has often been used to describe implementations that use locks, rather than a non-blocking approach, as a method of synchronizing which transactions will commit and which will retry. This is in contrast to our approach, which relies on locks to ensure that a transaction will always commit the first time.

III. STM Implementation

Our approach to implementing a lock-based STM for use in real-time systems relies on two components: a conflict-detection mechanism and an underlying locking protocol. The conflict-detection mechanism determines which areas of shared memory must be classified as one or more resources, and the locking protocol grants access to these resources.

A. *Detection of potential conflicts*

We assumed above that any section of shared memory could be abstracted to a specific resource.

A required assumption of real-time applications, especially those intended for use in safety-critical systems, is that all tasks can be analyzed offline to determine worst-case execution times, resources that may be accessed, and the worst-case duration of any such resource access. In the context of an STM, we can leverage this knowledge to determine which tasks may conflict a priori. This gives us a significant advantage in both the method of execution and the offline analysis of schedulability.

Knowing what shared memory may be accessed by any given task allows us to define our set of resources offline. Any of these sections of shared memory that may be accessed by multiple tasks must be declared as a resource to be protected by a lock in our approach, while sections of shared memory that are only ever accessed by a single task are not considered resources. We plan to focus on this approach, also called a *word-based* approach, rather than an *object-based* approach. An object-based approach restricts the set of resources to the set of objects or data structures; finer-grained access to these resources would not be possible to provide with the object-based approach. Therefore, we will develop a word-based STM implementation, which will also allow for application-specific tuning, as described in Sec. IV.

B. *Locking approach*

In determining what constitutes a resource, as described above, we can ignore the manner in which these resources are accessed; specifically, we can ignore if a specific access requires mutual exclusion and if multiple resources must be accessed concurrently. However, when implementing the resource-access coordination supported by a locking protocol, we must consider read and write accesses, as well as multiple-resource accesses, typically seen in non-transactional code as nested resource accesses.

We plan to use existing techniques to transform marked `atomic` sections at compile time to sections containing properly nested lock and unlock calls.

It is crucial, in a lock-based approach, to focus on the use of *contention-sensitive* locking protocols—those which ensure the blocking a task experiences is proportional to the number of other tasks that share one or more resources with the task of interest [19], [22], [23]. This is in contrast to locking protocols that can only guarantee blocking bounds based on the number of tasks or the number of processors in the system.

When multiple resources may be accessed concurrently, achieving a contention-sensitive ordering of resource accesses is made more difficult by the potential for transitive blocking chains—chains of resource-access requests that can cause a task to be delayed by many other non-conflicting tasks. To achieve contention-sensitive resource access, a locking protocol must provide a mechanism that allows non-conflicting tasks to cut ahead of other tasks waiting for resource access.

While contention-sensitive locking protocols can also be used in contexts independent from our STM, the structure of the STM is what eases application development and certification burdens. Additionally, because the STM examines trans-

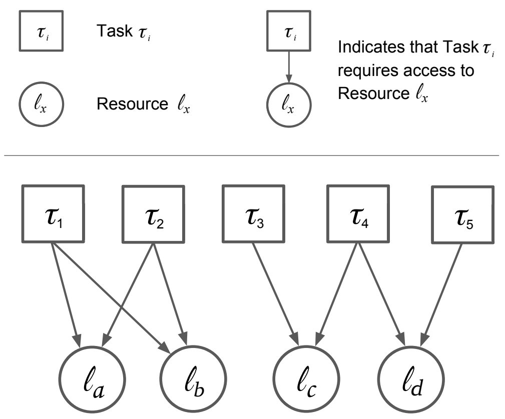

Fig. 1. Task system with five tasks requiring access to four resources.

actions before transforming these transactions into lock-based resource requests, we can tune performance in a fine-grained manner without modifying application code. We explore one method of tuning performance below as one of the additional components that can augment our STM implementation.

IV. Additional Components

We elaborate on the benefit our implementation can provide by exploring additional components that can be added, but are not fundamental, to an implementation of STM. First we introduce a method by which to improve performance on a per-application basis. Then we describe an approach to accommodate a wider set of resources and discuss how we plan to manage the additional schedulability challenges that may pose.

A. Tunable locking granularity

Generally, locking granularity refers to how many resources are protected by a single lock. As mentioned above, a fine-grained locking approach tends to protect each resource by its own lock, while a coarse-grained approach may protect all resources by a single lock. With the latter approach, different resources could never be accessed concurrently. We illustrate this difference with the following example.

Example 1: Consider tasks τ_1 to τ_5 and resources ℓ_a to ℓ_d. Each task requires access to at least one resource, as shown in Fig. 1. For example, τ_1 requires access to ℓ_a and ℓ_b. A coarse-grained approach to protecting access to these resources would define a single lock $L_1{:}\{\ell_a, \ldots, \ell_d\}$ to protect all resources. With this approach, τ_1 would be prevented from writing to ℓ_a while τ_3 writes to ℓ_c, even though τ_1 and τ_3 do not share any of the same resources. A fine-grained approach would instead use four locks: $L_1{:}\{\ell_a\}$, $L_2{:}\{\ell_b\}$, $L_3{:}\{\ell_c\}$, and $L_4{:}\{\ell_d\}$. This would allow τ_1 and τ_3 to write data concurrently. This benefit comes at the cost of maintaining more locks, which requires additional time and memory.

Between these two extremes, however, there is a middle ground. A few resources could be protected by the same lock, and multiple such groups of resources could exist.

Example 1 (cont'd): A different approach for the task set shown in Fig. 1 could be to have two locks: $L_1{:}\{\ell_a, \ell_b\}$ and $L_2{:}\{\ell_c, \ell_d\}$. Using these two locks allows some concurrent resource access (τ_1 and τ_3 could access their respective resources concurrently), but prevents τ_3 and τ_5 from concurrent resource access, though they do not share a common resource. However, notice that τ_1 and τ_2 see no change in potential access conflicts with this reduction of locks compared to those provided in the most fine-grained lock assignment.

Thus, one method by which we seek to improve performance is by adjusting the locking granularity on an application-specific basis. This performance is ultimately measured by schedulability. Affecting schedulability, however, are the *overhead* of the locking protocol—how long the locking protocol takes to process a resource-access request—and the worst-case *blocking* a task may experience—the sum of all resource-access durations that may delay the task from its resource access.

With fewer resources, the overhead of a locking protocol may be reduced; there would be less lock state to maintain and tasks would generally require fewer of these locks. If lock state were to be held in cache by techniques explored elsewhere [1], [5], [17], [20], [28], [29], the reduced cache footprint that would come with using fewer locks could cause other performance improvements as well. Prior work has shown that, even with a very modest number of locks for which state must be maintained, the reduction in overhead when the protocol is able to run cache-hot is substantial [23].

In contrast, reducing the number of resources may cause more tasks to compete for a given lock. This occurred in Ex. 1 for τ_3; the lock τ_3 must acquire in the most fine-grained approach had only one other task, τ_4, ever requiring the same lock. In the suggested middle-ground approach, the lock τ_3 required was also required by τ_4 and τ_5 for their respective resource accesses. An increase in the number of tasks contending will increase the worst-case blocking, which must be accounted for in schedulability analysis.

Though varying lock granularity has been studied in other contexts [12], [13], to our knowledge it has not been explored in real-time systems, in which the chosen locking granularity can affect schedulability, and thus, system correctness. Choosing the locking granularity for use with a specific application will depend on the number of resources, the memory footprint of the locking protocol state, the worst-case resource-access durations, the number of tasks, and the number of processors on the platform. Additionally, how resources ought to be assigned to locks is a non-trivial question that depends not only on the task-resource relationships, but also which tasks could incur higher blocking and remain schedulable.

B. Support for additional resources

To this point, the only resources we have considered are those stored in shared memory. With few exceptions [27], traditional STMs can only provide access protection to shared-memory resources. However, the lock-based approach that we propose can easily be adapted to support additional resources, such as hardware resources (*e.g.*, GPUs). In our model, any such resources can simply be added to those that must be protected by some lock.

Our initial approach to supporting a GPU or some other accelerator will treat the entire device as a single resource. This approach allows timing guarantees to be made easily, as a given task will always run in isolation on the device. (This reflects previous approaches to managing GPUs [10], though the complexities of these devices are being explored to determine if each GPU could be considered as multiple resources based on the internal components and proprietary scheduling decisions [2], [24].)

While supporting an additional set of resources may be relatively simple to implement, it may have large effects on the system as a whole. In general, reading from or writing to shared memory is very quick, but accessing a hardware resource may be significantly more time-consuming. This variation can also depend on the type of computation being done with, or on, the resource. Mixing resource types could increase the worst-case access delay to the point of making a system unschedulable.

Therefore, we will explore the effects of mixing different resource types. We will experiment with developing protocols that have a fast-path mechanism for certain resource types. We will investigate grouping resources based on characterizations such as access type (read or write), access duration (short or long), and number of resources accessed together (one, few, or many). One of these groups can then be selected to use the fast-path mechanism of our locking protocol.

Note that by choosing one of these metrics on which to group resources, we will influence how to best choose resource assignments to individual locks based on locking granularity. Therefore, applying additional techniques to improve the schedulability of an application, such as allowing other shared resources or modifying locking granularity, cannot be done in isolation.

V. Conclusion

We have presented our vision for a lock-based software transactional memory implementation for use in real-time systems. We described how shared-memory access within a transaction can be transformed to a set of lock and unlock calls for specific resources and how these can be supported by contention-sensitive locking protocols. This structure will provide a separation of concerns between the development and certification of a real-time application and the coordination of concurrently running tasks accessing and modifying shared resources.

Building on this proposed implementation, we discussed tuning the granularity of the locks on a per-application basis. We then suggested how non-shared-memory resources could be accommodated by our STM implementation and shared our planned examination of how to reduce the impact that variations in resource type have on other tasks.

References

[1] S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, 2014.

[2] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *RTSS*, 2017.

[3] A. Barros, L. Pinho, and P. Yomsi. Non-preemptive and SRP-based fully-preemptive scheduling of real-time software transactional memory. *Journal of Systems Architecture*, 61(10):553–566, 2015.

[4] C. Belwal and A. Cheng. Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *IEEE Embedded Systems Letters*, 3(1):37–41, 2011.

[5] M. Campoy, A.P. Ivars, and J.V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, 2001.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[7] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, 2011.

[8] M. El-Shambakey and B. Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *DAC*, 2012.

[9] M. El-Shambakey and B. Ravindran. On real-time STM concurrency control for embedded software with improved schedulability. In *ASP-DAC*, 2013.

[10] G. Elliott, B. Ward, and J. Anderson. Gpusync: A framework for real-time gpu management. In *RTSS*, 2013.

[11] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006.

[12] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[13] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, 1975.

[14] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM Sigplan Notices*, volume 38, pages 388–402. ACM, 2003.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[16] M. Herlihy and J. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[17] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS*, 2011.

[18] S. Hirve, A. Lindsay, B. Ravindran, and R. Palmieri. On transactional memory concurrency control in distributed real-time programs. In *CLUSTER*, 2013.

[19] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS*, 2015.

[20] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.

[21] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Rivière. Deadline-aware scheduling for software transactional memory. In *DSN*, 2011.

[22] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. In *RTNS*, 2017.

[23] C. Nemitz, T. Amert, and J. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *ECRTS*, 2018.

[24] N. Otterness, M. Yang, T. Amert, J. Anderson, and F. D. Smith. Inferring the scheduling policies of an embedded cuda gpu. In *OSPERT*, 2017.

[25] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *RTCSA*, 2009.

[26] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[27] H. Volos, A. Tack, N. Goyal, M. Swift, and A. Welc. xCalls: safe I/O in memory transactions. In *EuroSys*, 2009.

[28] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.

[29] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *RTAS*, 2017.