Appears in: Proceedings of the 41st Real-Time Systems Symposium (RTSS 2020)
virtual, December 2020

Work In Progress: Control-Flow Migration for Data-Locality Optimisation in Multi-Core Real-Time Systems

Stefan Reif[1], Phillip Raffeck[1], Peter Ulbrich[2], Wolfgang Schröder-Preikschat[1]
[1]*Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Germany*
[2]*Technische Universität Dortmund, Germany*
{reif,raffeck,ulbrich,wosch}@cs.fau.de

***Abstract*—Multi-core real-time systems face the challenge of efficiently maintaining consistency of shared data despite concurrent operations. Existing synchronisation techniques ignore data locality, resulting in cache-related execution time overheads. This paper proposes Migration-Based Synchronisation (MBS), a transparent replacement for locks. In MBS, control flows are migrated to data, instead of moving data to control flows. The consequence is an improvement of data locality that reduces the worst-case execution time of critical sections, and indirectly, worst-case blocking bounds.**

***Index Terms*—real-time systems, multi-core systems, data locality, thread synchronisation, control-flow migration**

I. Introduction

Multi-core processors promise improved performance, especially as cores become available in abundance. However, despite all the recent improvements [1] in real-time engineering, their adoption in real-world systems is still plagued with the stigma of inadequate predictability. A widely accepted approach is still to partition the system (i.e., task set) into independent parts that operate with little, if any, dependencies. While this is a viable approach in settings with few cores and a simple application structure, it becomes inefficient with increasing system complexity as applications can no longer be partitioned. Consequently, tasks suffer from interdependencies and interferences caused by shared resources. A plethora of multi-core synchronisation approaches [2], [3] has been developed, including message passing, wait-free programs, and blocking synchronisation with locks. These solve the underlying coordination problem and allow an inference of blocking times at the abstraction level of WCETs. Synchronisation in multi-core systems, however, has a detrimental effect on bounding the WCET: It entails sharing data (i.e., critical sections) between control flows on different cores.

Contemporary multi-core processors typically exhibit a multi-level memory hierarchy with core-local first-level caches, a shared last-level cache, and shared main memory. In sum, access latency is minimal if data is stored core-local (i.e., first-level cache or scratchpad) and increases with cache misses throughout the hierarchy. Simultaneously, concurrent and conflicting accesses cause interference, with shared data structures being prone to such effects when distributed among cores. Consequently, synchronisation not only introduces runtime overheads but, much worse, has a severe impact on WCET analysis's pessimism. In the worst case, cache analysis must assume an unknown cache and that all shared data must be loaded from a remote core. In turn, the resulting over-approximations adversely affect the system's overall schedulability, thwarting the aspired multi-core advantage.

Migration in real-time systems faces very similar pessimism issues. The root cause is the task's state, the resident set size, to be transferred between cores. As this size typically varies substantially depending on the point of migration, analyses may be forced to operate under overly pessimistic assumptions. From the point of control-flow migration, however, remote-core interference is, contrary to synchronisation, non-existent. So far, established real-time synchronisation approaches typically do not exploit these similarities and potential synergies (we detail this aspect in Section II).

In this paper, we reconsider the challenge of predictable synchronisation mechanisms by leveraging both concepts. Our approach's underlying assumption is that we can determine dependencies and the residence set size through static analysis [4], [5], [6] (in this respect, we exploit the static nature of real-time systems). We propose Migration-Based Synchronisation (MBS), a novel synchronisation approach for shared data structures that can be used as a transparent replacement for locks. The goal of MBS is to avoid cache misses in critical sections by improving data locality. In MBS, control flows are migrated to data, rather than data to control flows. The resulting memory locality is vital to simplify cache and WCET analyses, thereby reducing over-approximations of critical sections. This increased predictability collaterally allows for tightening the bounds on blocking times in high-level schedulability analysis. Furthermore, by increasing cache locality, our approach also minimises synchronisation overheads, improving the overall performance.

The contributions of this paper are three-fold:

- We present MBS, a novel synchronisation technique for shared data structures in multi-core real-time systems.
- We outline that data locality is a key aspect for the performance, as well as WCET estimations, of critical sections. With MBS, the location of data is known at compile-time, which improves, and also simplifies, WCET analyses.
- We discuss the influence of WCET reduction of critical sections on blocking-bound and schedulability analyses.

The rest of the paper is structured as follows. Section II briefly summarises the necessary background and related work. MBS is presented in detail in Section III. Section IV experimentally demonstrates that data locality can improve WCET estimations, and Section V concludes the paper.

II. Background and Related Work

Beyond ensuring consistency despite concurrent operations, a crucial aspect of synchronisation in real-time systems is to guarantee upper bounds on blocking times and to control priority inversion. Accordingly, the literature is rich in synchronisation approaches. Two fundamental approaches can be distinguished: lock-based and delegation-based.

Locks synchronise accesses to shared data by sequencing of critical sections. In general, this has the following temporal implications on a control flow. First, a locking protocol is executed to control priority inversion, which might cause the thread to block. For a given protocol and depending on the priority structure, an upper bound on the blocking time can be obtained [3], [7], [8]. Second, when the thread starts executing the critical section, the protected shared data is likely not in the local cache. In particular, without further assumptions, a sound worst-case analysis must assume the caches to be cold. However, as critical sections tend to be memory-intensive (as they protect shared data), numerous cache misses are detrimental for both the overall runtime performance and the schedulability analysis. The latter suffers from the interaction between WCET and blockade time analysis. Third, when the critical section has completed, and the thread releases the lock, the shared data stored in the local cache instantaneously becomes a burden as accesses are no longer permitted. In summary, the synchronisation and its temporal analysis suffer from poor data locality in numerous ways. In this paper, we, therefore, focus on cache locality in critical sections.

Taking data locality into consideration has the potential to tighten WCET bounds of critical sections significantly, and thereby indirectly foster schedulability analyses. Consequently, cache analysis is an extensively researched topic in real-time systems [9], as the simplistic but sound worst-case assumption of constant cache-miss leads to unacceptable overestimations. Sound memory and cache analysis is, however, a challenging task, which even in single-core systems still is an active research topic [10]. Research in multi-core systems largely focuses on compositionality [11] or creating abstractions equivalent to single-core systems [12]. In particular, interference between caches is avoided by mechanisms such as cache partitioning and cache locking [9]. MBS facilitates such analyses without the need for additional mechanisms as cache locality for shared resources is guaranteed by design.

An alternative to lock-based is *delegation-based* synchronisation, where the critical section is passed to another thread that coordinates its execution. Delegation is a form of partial control-flow migration and comprises various approaches: for example, Actors [13], Flat Combining [14], and Guards [15]. Closely related to MBS are Remote Core Locking [16] and ffwd [17], where critical sections are migrated to a single *server* core. Delegation-based synchronisation, however, requires the critical section to be contained as a delegatable unit, which consequentially requires source code modifications. Our approach with MBS is to avoid such modifications.

Moving the control flow to the requested data instead of the other way around is not an entirely new concept. The original version of the multiprocessor priority ceiling protocol (MPCP) [2] handles shared global data by pinning critical sections to dedicated processors. However, the focus of MPCP is solely on controlling priority inversion of global resources, without attention to detrimental core sharing and cache locality. Boyd-Wickizer et al. [18] propose a scheduling algorithm based on migrating control flows to the data they operate on using manual source-code annotations. They neither consider timing implications nor is their approach transparent to the application.

Further research on the exploitation of data locality by migration includes hardware-based techniques to reduce the number of cache-misses via thread migration [19], [20] as well as scheduling according to data-locality to minimise cache interference [21], [22]. Neither do these works consider real-time aspects nor do they have a particular focus on synchronisation of critical sections.

III. Approach

Unlike existing approaches, MBS redefines the underlying locking mechanism to eliminate interference on shared data entirely. Therefore, MBS replaces the traditional locking or dispatching of data by a migration-based strategy.

A. Migration-Based Synchronisation

For each shared resource in the system, a dedicated *synchronisation core* is reserved for accesses to this shared resource only. In other words, these cores are considered resources themselves that are assigned to a thread *exclusively*; acquiring them grants access to the associated shared data. Threads may only migrate to a synchronisation core (or back) if they explicitly ask for that migration. Synchronisation cores obey priorities but do not provide preemption. This ensures proper coordination at the expense of the generality of these cores: they must not be considered in any implicit migration-based technique, such as load balancing. Note that these restrictions do not apply to the remaining regular cores.

The fundamental concept of MBS is then to place data structures in the local cache of their synchronisation core, and to move the control flow to the shared data on the respective synchronisation core, instead of moving the requested data to the core of the requesting control flow. Our approach can be used using the well-known interface of locks: The acquisition of an exclusive resource corresponds to the migration of the control flow to the associated synchronisation core. By migrating the control flow back to the original core, the resource gets released. As synchronisation cores ensure that only one control flow can run at each moment in time, and will not be preempted, mutual exclusion is guaranteed for each resource access.

B. Analysability

The main benefit of MBS is the guaranteed cache locality for shared data. As all shared resources are assigned to exactly one synchronisation core, data structures can be preloaded into the respective core-local caches on system startup. Since only control flows running on the associated synchronisation core are allowed to access the data structure (i.e., protection), the data never leave their local cache. As a consequence, no cache-coherency measures are necessary. This guaranteed cache locality is beneficial for the timing analysis of critical sections in two ways. Firstly, interferences due to cache coherence do not need to be modelled or estimated, which simplifies the analysis. Second, execution times of critical sections decrease because data structures are already present in the cache. In addition, static analyses can be made aware of this locality, promising tighter WCET bounds of critical sections.

For the *system-level* timing analysis, MBS can be implemented in a way that is equivalent to locks. First, the local scheduler of synchronisation cores can employ the same assignment strategy as a lock (in particular, adhere to priorities). Thereby, unbounded priority inversion is avoided by design, as the core-local scheduler of synchronisation cores is priority-aware, and resource assignment cannot conflict with processor assignment strategies. Second, if threads do not release their original core at migration (i.e., the original core remains idle), the schedule is equivalent to locks. In that (rather inefficient) variant of MBS, existing blocking-bound and schedulability analysis techniques of traditional locks are directly applicable. More efficient variants that release the original core are left for future work, as they require modification to blocking-bound analyses, but promise an improvement of processor utilisation. Additionally, even when using the same system-level analysis techniques, MBS yields better results due to the WCET reduction of critical sections.

C. Overheads

The introduction of migration as a tool for synchronisation brings in all the overhead induced by migration, namely the copying of thread-related data between cores. In the case of MBS, stack-local data and (parts of) the thread control block associated with the control flow requesting the resource need to be transferred to the synchronisation core and back. Depending on the specific hardware platform and memory layout, theses overheads may or may not be significant [23], [24]. The advantage of MBS is that the transfer of control-flow related data replaces the transfer of the shared data. Depending on the size of the shared data, control-flow related data might be significantly smaller, leading to less data that needs to be transferred. This effect increases if only the necessary parts of stack-local data are transferred to the synchronisation core.

The overheads of MBS are, in the worst case, similar to locks that release the processor while waiting. Consider the following worst-case scenario of a control flow waiting for access to a shared resource guarded by a traditional lock. The control flow requests the currently occupied lock and is thus preempted and writes its local processor state to memory.

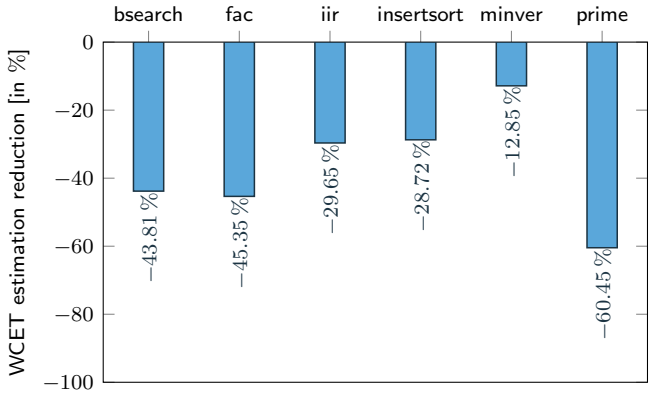

Fig. 1. Reduction achieved by hot caches relative to cold-cache execution in the WCET estimates by TimeWeaver for several TACLeBench benchmarks.

While waiting for the lock, the warm caches get scrambled as other threads run and overwrite the cache state. Once the lock is granted, the control flow restores its processor state from memory and begins execution of the critical section with cold caches. In summary, the worst-case costs of a lock acquisition involve storing the current thread state to memory and restoring it later, which is effectively a migration, but *in the time dimension* rather than between cores.

A drawback of MBS is the need for many processor cores, as a dedicated synchronisation core is necessary for each shared resource. Especially, MBS is not going to work in single-core environments. But also the use in multi-core systems might be problematic if most of the cores are used as synchronisation cores, as the performance of the system may suffer. The dedication of a processor core for synchronisation is a beneficial trade-off if the number of hardware cores is sufficiently large and data structure accesses are relatively important for the application performance.

IV. Evaluation

As the main advantage of MBS lies in the guaranteed cache locality, it is helpful to have a look at the potential benefits. We conducted first measurements on a Xilinx Zedboard Zynq-7000 [25] featuring an ARM Cortex-A9 dual-core [26]. Selected benchmarks taken from the TACLeBench benchmark suite [27] were executed both with a cold and a hot cache. As we primarily aim to facilitate worst-case analyses for tighter WCET and blocking-time bounds, the obtained execution traces were used as the basis of a hybrid WCET analysis with the TimeWeaver tool from AbsInt [28]. For each benchmark, the analysis was performed both using traces with hot caches only and using all obtained traces. Figure 1 shows the improvement gained by the known cache locality, yielding up to 60.45 % lower WCET bounds.

These results are, of course, not necessarily representative for whole real-time systems and their critical sections. Additionally, they are highly dependent on the memory and cache hierarchy. They show, however, that future implementations of the MBS approach can yield significant improvements in both

system performance and analysability. We, therefore, deem future research of that approach fruitful and necessary.

V. Conclusion

This paper has presented Migration-Based Synchronisation (MBS), a synchronisation approach that aims at optimising data locality. We have shown that data locality is an important matter in real-time systems, as it affects the WCET, blocking times, and schedulability. Thereby, MBS takes a constructive approach that purposefully places data in core-local caches and migrates control flows to the data they operate on. In consequence, MBS simplifies cache-state analyses and, thus, the WCET estimation, while at the same time achieving lower WCET estimates. MBS is straight-forward to implement and does not rely on specific program structures or additional run-time mechanisms. It can transparently replace locks—both in the source code and in blocking-bound analyses.

Future work will integrate MBS in a multi-core real-time operating system [29] to evaluate system-level effects in realistic application scenarios. The trade-off between the cost of dedicating a processor core for the synchronisation of a shared data structure and the improvement in synchronisation performance depends on the application structure as well as the number of available hardware cores. Besides, as the number of cores is rather small in current-generation COTS hardware, the optimal solution is likely a mixture of MBS with other synchronisation techniques. The mapping of data structures to synchronisation cores with consideration of cache capacity, data structure size, thread-related data size, and also contention patterns, is another optimisation problem that remains as future work. Besides, variants of MBS can selectively keep a reservation of the original processor core at migration, to support lock nesting or to improve processor utilisation. Such a reservation can utilise low-power sleep states to save energy. Furthermore, the inclusion of latency-hiding techniques might reduce migration-related overheads.

Acknowledgment

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants no. SCHR 603/8-2 (“LAOS”), SCHR 603/9-2 (“AORTA”), SCHR 603/10-2 (“COKE”), SCHR 603/13-1 (“PAX”), and project number 146371743 – TRR 89 “Invasive Computing”.

References

[1] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, “A survey of timing verification techniques for multi-core real-time systems,” *ACM Computing Surveys*, vol. 52, no. 3, Jun. 2019.

[2] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-time synchronization protocols for multiprocessors.” in *Proc. of the 9th IEEE Intl. Real-Time Systems Symp.*, vol. 88, 1988, pp. 259–269.

[3] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, “A flexible real-time locking protocol for multiprocessors,” in *Proc. of the 13th Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2007, pp. 47–56.

[4] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, “From Intent to Effect: Tool-based Generation of Time-triggered Real-time Systems on Multi-core Processors,” in *Proc. of the 19th IEEE Intl. Symp. on OO Real-Time Distributed Computing*, 2016, pp. 134–141.

[5] T. Klaus, P. Ulbrich, P. Raffeck, B. Frank, L. Wernet, M. Ritter von Onciul, and W. Schröder-Preikschat, “Boosting Job-Level Migration by Static Analysis,” in *Proc. of the 15th Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2019, pp. 17–22.

[6] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, “Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis,” in *Proc. of the 25th Real-Time and Embedded Technology and Applications Symp.* Montreal: IEEE, 2019, pp. 318–330.

[7] B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[8] B. Ward and J. Anderson, “Fine-grained multiprocessor real-time locking with improved blocking,” in *Proc. of the 21st Intl. Conf. on Real-Time Networks and Systems*. ACM, 2013, pp. 67–76.

[9] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, “A survey on static cache analysis for real-time systems,” *Leibniz Trans. on Embedded Systems*, vol. 3, no. 1, pp. 05–1, 2016.

[10] G. Stock, S. Hahn, and J. Reineke, “Cache persistence analysis: Finally exact,” in *Proc. of the 40th Real-Time Systems Symp.* IEEE, 2019, pp. 481–494.

[11] S. Hahn, M. Jacobs, and J. Reineke, “Enabling compositionality for multicore timing analysis,” in *Proc. of the 24th international conference on real-time networks and systems*, 2016, pp. 299–308.

[12] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “Wcet(m) estimation in multi-core systems using single core equivalence,” in *Proc. of the 27th Euromicro Conf. on Real-Time Systems*, 2015, pp. 174–183.

[13] G. Agha, “Concurrent object-oriented programming,” *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, Sep. 1990.

[14] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” in *Proc. of the 22nd Annual Symp. on Parallelism in Algorithms and Architectures*, 2010, pp. 355–364.

[15] S. Reif and W. Schröder-Preikschat, “POSTER: A predictable synchronisation algorithm,” in *Proc. of the 23rd Symp. on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 415–416.

[16] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, “Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications,” in *Proc. of the USENIX Annual Technical Conf.* USENIX, 2012, pp. 65–76.

[17] S. Roghanchi, J. Eriksson, and N. Basu, “ffwd: Delegation is (much) faster than you think,” in *Proc. of the 26th Symp. on Operating Systems Principles*. ACM, 2017, pp. 342–358.

[18] S. Boyd-Wickizer, R. Morris, and F. Kaashoek, “Reinventing scheduling for multicore systems,” in *Proc. of the 12th Work. on Hot Topics in Operating Systems*. USENIX, 2009, pp. 1–5.

[19] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, “Directoryless Shared Memory Coherence Using Execution Migration,” *Parallel and Distributed Computing and Systems*, 2011.

[20] P. Michaud, “Exploiting the cache capacity of a single-chip multi-core processor with execution migration,” in *Proc. of the 10th Intl. Symp. on High Performance Computer Architecture*. IEEE, 2004, pp. 186–195.

[21] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki, “Fact: A framework for adaptive contention-aware thread migrations,” in *Proc. of the 8th ACM Intl. Conf. on Computing Frontiers*, 2011, pp. 1–10.

[22] M. Lee and K. Schwan, “Region scheduling: Efficiently using the cache architectures via page-level affinity,” *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 451–462, 2012.

[23] A. Bastoni, B. Brandenburg, and J. Anderson, “Cache-related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability,” in *Proc. of the 6th Intl. Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2010, pp. 17–22.

[24] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “LITMUSˆRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers,” in *Proc. of the 27th IEEE Intl. Real-Time Systems Symp.*, Dec. 2006, pp. 111–126.

[25] Xilinx, “Zynq-7000 soc: Technical reference manual,” 2018.

[26] ARM, “Cortex-a9 technical reference manual,” 2012.

[27] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, “Taclebench: A benchmark collection to support worst-case execution time research,” in *Proc. of the 16th Intl. Work. on Worst-Case Execution Time Analysis*, 2016.

[28] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, “Timeweaver: A tool for hybrid worst-case execution time analysis,” in *19th Intl. Work. on Worst-Case Execution Time Analysis*, 2019.

[29] P. Raffeck, P. Ulbrich, and W. Schrder-Preikschat, “Work-in-progress: Migration hints in real-time operating systems,” in *Proc. of the 40th Real-Time Systems Symp.*, 2019, pp. 528–531.