Adding Software Transactions to the Kernel

Michael J. Schultz and Dennis Brylow

Marquette University
Milwaukee, Wisconsin

Building interrupt driven, real-time embedded systems with low latency and jitter remains a challenging task. Jitter is introduced when interrupts must be delayed during an atomic section of code to ensure that an operation succeeds. This delay can be harmful because interrupts must be handled within a fixed amount of time. Current solutions are not ideal. Heavy locking can cause significant jitter while lightweight techniques (such as semaphores) bring a different set of problems. If code being executed has a lock and is interrupted by code needing the same lock the system may deadlock. Though techniques exist to avoid or handle such cases, they frequently increase interrupt run time. In an embedded system responsiveness is paramount to meeting real-time deadlines. This work explores the use of transactional memory to guarantee that a system will make progress throughout execution.

Transactional memory was proposed by Herlihy and Moss in 1993 [1]. At the time, they suggested integrating hardware support for lock-free transactions into future architectures. In the past fifteen years the closest we have come are the compare-and-swap or load-linked/store-conditional operation codes. These only perform checks on a single word of memory at a time. Processor speeds have increased significantly in embedded systems since transactional memory was proposed. With faster processors we can take advantage of using software instead of hardware to perform and maintain memory transactions. By using software transactional memory we are able to begin testing systems immediately, reduce the need for specialized hardware in an embedded system, and avoid potential pitfalls of having hard-wired limits on atomic code blocks.

One potential downfall to using software transactional memory is its lack of efficiency. Recently Intel has released a prototype of their C library (which includes a software transactional memory library) and an attached research compiler [2]. Their compiler adds keywords to the language that tell the compiler to convert code in atomic sections to use transactional memory. Intel states that they provide an efficient library that uses software transactional memory so as to not slow down the system. This is important if software transactional memory is to become an integral part of embedded systems that must be low powered and meet hard deadlines.

Our approach is to apply software transactional memory to key portions of the XINU operating system—such as device drivers and interprocess communication—that would typically use other locking mechanisms. By using transactional memory in these atomic sections we argue that arriving interrupts will not be delayed. This is a safe claim because transactional memory allows the interrupt to access shared memory safely. Upon completion, the interrupted code is able to safely proceed and in the worst case roll back any attempted write operations. For our project we are exploring how transactional memory can reduce jitter; under what situations transactional memory is or is not a viable tool; and how much overhead, in both code size and run time, is added to the system.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, 2006.

[2] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.