A Non-Intrusive, Operating System Independent Spinlock Profiler for Embedded Multicore Systems

Lin Li*, Philipp Wagner†, Albrecht Mayer*, Thomas Wild† and Andreas Herkersdorf†
*Infineon Technologies AG, Germany, Email: {lin.li, albrecht.mayer}@infineon.com
†Institute for Integrated Systems, Technical University of Munich, Germany
Email: {philipp.wagner, thomas.wild, herkersdorf}@tum.de

***Abstract*—Locks are widely used as a synchronization method to guarantee the mutual exclusion for accesses to shared resources in multi-core embedded systems. They have been studied for years to improve performance, fairness, predictability etc. and a variety of lock implementations optimized for different scenarios have been proposed. In practice, applying an appropriate lock type to a specific scenario is usually based on the developer's hypothesis, which could mismatch the actual situation. A wrong lock type applied may result in lower performance and unfairness. Thus, a lock profiling tool is needed to increase the system transparency and guarantee the proper lock usage. In this paper, an operating-system-independent lock profiling approach is proposed as there are many different operating systems in the embedded field. This approach detects lock acquisition and lock releasing using hardware tracing based on hardware-level spinlock characteristics instead of specific libraries or APIs. The spinlocks are identified automatically; lock profiling statistics can be measured and performance-harmful lock behaviors are detected. With this information, the lock usage can be improved by the software developer. A prototype as a Java tool was implemented to conduct hardware tracing and analyze locks inside applications running on the Infineon AURIX microcontrollers.**

I. Introduction

"For accessing shared resources on a multi-core system, use locks to sequence the individual accesses." This advice is usually given in elementary programming courses, often together with an example of how to use mutex lock and unlock functions. Mutex locks are easy to use and simple to implement, making them the most widely used synchronization method. However, merely following the above advice rarely results in programs which are not only functionally correct but also show good performance.

To understand why this is the case, a look into the implementation of lock functions, as they are commonly provided by the operating system, is required. A lock can be implemented in many different ways [1]–[3], and while the functional behavior stays the same, the performance can be tuned towards various characteristics and use cases. No single lock implementation is known to be optimal in every use case, as found in a recent study by Guiroux and Lachaize [4]; even worse, a lock implementation which performs great in one scenario can have a significant performance problem in other cases. That's where lock profiling tools come to help. They observe the program execution during runtime, measure relevant metrics and possibly suggest improvements, e.g. which lock implementation might perform better in the given scenario.

Lock profiling tools face two major challenges. First, measuring the performance of an application requires a measurement tool which does not influence the performance of the observed application. Already slight changes in the program timing can result in different scheduling and therefore very different program behaviors. Second, the profiling tool must at least be able to recognize calls to the lock acquisition and releasing functions. Which function call represents a call to a lock function depends on the operating system or the used library. For example, on Linux and many other UNIX-like operating systems the lock interface is defined in the POSIX threads (pthreads) standard.

On embedded systems, which are the focus of this paper, no single operating system dominates the market. Instead, multiple different libraries and lock interfaces [5], [6] are used, which are often further tuned by system developers. This requires a lock profiling solution which is able to recognize locks independent of the naming conventions used by the operating systems or the libraries.

In this paper, we propose an operating-system independent spinlock profiling approach for embedded multi-core systems. Not all locks can be profiled with the proposed profiler because there are so many various lock implementations with different characteristics. We limit this work to the detection of lock functions that follow well-established patterns as discussed in literature and provided by embedded operating systems. The most frequently provided and used lock mechanism for mutual exclusion is the spinlock. In particular, we focus on binary semaphore spinlocks that operate on a single variable. This type of lock is used to protect a shared resource like a shared memory region or an I/O from multiple concurrent accesses by sequencing the accesses to it.

The proposed approach automatically detects binary semaphore spinlocks in embedded multicore systems. Then, statistics about the waiting and holding time, the number of failed attempts, etc. are created and presented to software developers. Hints related to inefficient spinlock behaviors are also provided to improve the spinlock performance. Our approach has two main characteristics.

- The proposed approach only uses commonly available tracing hardware to observe the software execution during runtime without altering its behavior. Therefore, no special-purpose hardware needs to be added to the chip.
- A multi-phase measurement and analysis approach is applied, which relies on the common hardware-level spinlock characteristics instead of specific operating systems or libraries.

The remainder of this paper is organized as follows. Section II summarizes the related work and the current state of the art. In the following Section III the lock profiling approach is presented. We then perform a case study using this tool in Section IV. Section V concludes the paper.

II. Related Work

In order to detect spinlocks and to suggest improvements in their usage, an understanding of spinlock implementations is required. We present our findings on this in the first part of

978-3-9815370-8-6/17/$31.00

this section. After that the state of the art in the development of lock profilers is discussed.

Spinlocks have been studied for years as a classic synchronization method to regulate shared resource accesses for multi-core systems. A primitive and simple spinlock implementation is the test-and-set spinlock. It spins using atomic operations, which is harmful to the performance of both the spinning thread and the other threads [1]. Due to this drawback, an improved implementation test-test-and-set lock was proposed [2]. It spins with normal read operations and only tries to apply atomic operations when the lock is potentially free. The performance is better than the previous one but it still has issues. In a cache-coherent machine, if several threads are waiting for a lock, only one will acquire it after releasing. The others' cache will still be invalidated, causing additional memory accesses and lower performance. The backoff spinlock was created to avoid such collisions [1]. However, all locks mentioned above do not guarantee fairness, so queue-based spinlocks e.g. MCS [3] were proposed. The MCS spinlock has a queue to sequence the lock acquisition, which is suitable for a large number of threads competing for one lock [7]. There are many more lock types with different advantages and disadvantages. Recent research [4] has found that no single lock is best for all applications. The best lock varies with the number of threads, the number of cores and the hardware architecture.

Lock profilers are tools that give insight into the system and enable developers to improve lock usage. There are many different lock profiling tools for different applications already available as shown in Table I. Locks and wait analysis in Intel VTune [8] uses interrupts to sample basic information about threads and synchronization objects. VTune works for x86 architecture. HPCToolKit [9] utilizes sampling instead of software instrumentation. It can be applied from desktop to supercomputers [10]. In order to make use of performance counters efficiently, Demme et al. created LiMiT [11], which instruments applications and provides a fast interface to access performance counters without kernel calls, achieving a lower overhead. For the same reason, HaLock [12] is applied and a specific uncacheable memory address is reserved to record lock information so that the memory inference to normal memory operations is minimized. HaLock leverages a hybrid mechanism which combines software-based lock detector and hardware-based trace collector. Lockmeter [13] was developed and released as a Linux kernel patch to record the spinlock usage by applications. A framework named PEPs [14] allows developers to manually annotate the interesting source code region. Then timestamped event tracing per thread is collected during execution and presented to the developer. The target platform of PEPs is not specified but it relies on a specific operating system (OS). Lock profiling tools are also useful to virtual machines. In a kernel based virtual machine, a virtual CPU (vCPU) could be preempted while holding a lock. Then other vCPUs have to spin extra-long time to acquire this lock. Zhang et al. proposed ANOLE [15], which instruments the kernel based virtual machine (KVM) to minimize this issue. Lock profiling tools for Java applications also have been studied, where instrumentation can be either done in the Java virtual machine [16] or in Java libraries [17].

Compared to the above existing tools, our approach is focused on binary semaphore spinlocks operating on a single variable without OS dependency. This feature is favored by the embedded field due to the fact that there are many embedded OSes existing. As far as we know, there are no other studies focused on the lock profiling for embedded systems without OS dependency and the proposed approach is the first pure hardware-based solution. It makes uses of non-intrusive hardware tracing, provided by many COTS such as Infineon's multi-core debug solution (MCDS), Freescale's Nexus and ARM's CoreSight. In this way, software instrumentation and execution overhead are avoided.

TABLE I
COMPARISON OF LOCK PROFILING TOOLS

Tool	*Data collection method*	*Target*
ANOLE [15]	Virtual machine instrumentation	Linux KVM
Free Lunch [16]	Virtual machine instrumentation	Java
HaLock [12]	HW-assisted kernel instrumentation	Linux
HPCToolKit [9]	Sampling	Desktop [10]
Jucprofiler [17]	Library instrumentation	Java
LiMiT [11]	Software instrumentation	Linux
Lockmeter [13]	kernel instrumentation	Linux
PEPs [14]	Application instrumentation	OS specific
Vtune [8]	Interrupt-based sampling	x86
Our method	Hardware tracing	Embedded systems

III. LOCK PROFILING APPROACH

Our approach is designed to detect and analyze spinlocks in an operating-system-independent, non-intrusive way. In the following, we discuss how the hardware sees spinlocks, how hardware trace can be used to observe spinlocks, and finally, how the gathered data can be used for profiling.

A. Spinlock Characteristics

In order to detect spinlocks without instrumentation at the source code level and without knowing the lock and unlock function calls, our approach relies merely on the data available on chip. This includes all program instructions as they are executed on CPU, and all memory accesses. With this data, we formulate a pattern to detect binary semaphore spinlocks.

A binary semaphore spinlock has only a single semaphore with a binary value for the availability. In this paper, only the ones based on a single variable are considered. This group includes many classic spinlock types e.g. test-and-set [1], test-test-and-set [2], or backoff locks [1]. Many queue based locks are not included because they usually have a copy for each thread. Several assumptions are made to limit our analysis scope. (i) The architecture supports atomic instructions and the target lock types also use atomic instructions. Modern multi-core architectures support atomic instructions and most lock types employ these instructions. (ii) The lock variable is allocated statically during compile time. This is true for most embedded applications, especially for safety-critical ones. (iii) Thread ID information is provided. Thread tracing is already supported by many commercial tools and is not the focus of this paper, so this information is considered as given.

During the initialization, the initialized value is considered as free while the opposite value is defined as busy. A binary semaphore spinlock, described by the memory address of its lock variable, is observed, if all three conditions below hold.

1) An atomic instruction writing to a memory location is observed.
2) Only the binary values 0 or 1 are written to this memory location.
3) The value of this memory location is only changed by its owner thread, which is defined as the thread that temporarily owns the memory location. A thread becomes the owner by changing a lock's value from free

to busy. Then it changes the value back to free, meaning not the owner anymore. During this time, the lock value is not supposed to be changed by any other threads.

B. Runtime Spinlock Detection

By applying the described characteristic pattern matching to the observations from an embedded system, spinlocks during the program execution can be detected. A prerequisite is the ability to collect the required data during runtime. For this task, we use the hardware tracing support embedded in most of today's COTS chips, e.g. in the form of NEXUS 5001, ARM CoreSight, or Infineon MCDS.

Tracing is able to observe low-level hardware-related operations e.g. program flow, bus transfers and memory accesses. The trace data is decoded by an off-line post-processing tool. Table II shows one tracing example from Infineon's MCDS [18]. This example is decoded and slightly simplified.

TABLE II
A HARDWARE TRACING EXAMPLE

Index	*Time*	*Opoint*	*Origin*	*Address*	*Operation*	*Value*
1	19	CPU0	CPU0	0x80000A48	IP	-
2	20	CPU0	CPU0	0x80000A4C	IP	-
3	21	CPU1	CPU0	0x70000030	R32	0x0
4	30	CPU1	CPU0	0x70000030	R32	0x0
5	37	CPU1	CPU0	0x70000030	W32	0x1

A time stamp records the time when an operation finishes in clock cycles. The location where the message is generated is defined as an observation point (*Opoint*). The *Origin* column displays the origin of an operation, for instance the master in a bus transfer. *Address* indicates the IP address and memory access address. *Operation* distinguishes program flow tracing (IP) from memory access tracing (R or W) and also shows the details of memory accesses for instance access width. For example, the message in the first row records an instruction (0x80000A48) executed by CPU0. The fifth message shows that CPU0 issues a 32-bit write to the local memory of CPU1.

Our lock detection is built on top of hardware tracing. Locks are detected in three steps. First, we obtain a list of candidate locks by observing the full program stream and picking out all program counters which execute atomic instructions. The memory locations that are accessed by the atomic instructions are marked as lock candidates. Second, the obtained list of lock candidates is refined by following the above requirements 2 and 3. The memory locations with non-binary values are removed from the list. The rule that the busy lock's value is not allowed to be modified except for its owner should be conformed to. This rule is checked against the remaining lock candidates and the not complying candidates are abandoned. Using this approach, we are able to obtain a list with all memory locations of spinlocks used by the application.

C. Profiling statistics and report

Given the lock locations after the refinement, all the memory accesses to these locations are collected and ordered according to time stamps. After initialization, lock acquisition is identified by detecting a value change and a write operation to the lock with the busy value. Accordingly, lock releasing is marked by a similar mechanism with the free value. For example, the last three messages in Table II show an lock acquisition example. A lock exists at 0x70000030 in CPU1's local memory. CPU0 successfully acquires this lock using the test-test-and-set mechanism.

The *Origin* shows which CPU issues a memory access. Using the information when a given thread is active on a core, a mapping between thread IDs and lock operations is established. A lock protects a shared resource, i.e. a shared variable or an I/O. It is possible to associate a shared resource with a lock variable using the Lockset algorithm [19], [20]. The algorithm first creates a candidate list for each shared resource. Each time a resource is accessed, locks except for the ones that are currently held are deleted from the list. After several iterations, only the lock that is always held when the resource is accessed remains. This lock is then interpreted as protecting the resource. Resources which are shared among threads but not protected by locks are highlighted for developers.

Many statistics can be calculated to show the lock usage. Waiting time is defined as the time spent from the first attempt to acquire a lock until the lock acquisition succeeds. Holding time is the time between the lock acquisition and the lock releasing. The number of failed attempts is the number of attempts from one thread to an occupied lock by another thread. The first attempt acquisition ratio, as the name indicates, is the ratio of the number of attempts that successfully acquire the lock at the first try to the total number of first attempts. Inefficient lock behaviors observed during the tracing such as spinning with atomic operations and thread preemption while holding a lock are reported as warnings. Improvements can be adopted by developers based on this information.

IV. A CASE STUDY

As an example of how to use our approach, we implemented a lock profiling tool which gathers trace data from an Infineon AURIX TC29x microcontroller using MCDS and profiles the lock usage of the running application. The application we selected is an Ethernet demonstration application which runs on top of the Erika OS [6]. It demonstrates how Ethernet can be used in an automotive context to transfer large amounts of data between control units. It consists of two parts. The first and central part runs on CPU0 and provides a service capable of receiving, processing and sending UDP frames. The second part is the UDP testing program, running on CPU1 without OS. Its main purpose is to stimulate and verify the first part by periodically generating data and sending it out using the UDP service provided in the service layer. It also checks the received UDP frames and validates their correctness. The interaction between the UDP testing program (Thread 200) and the UDP service (Thread 14) is protected by a spinlock.

The implemented lock profiling tool is applied to check the spinlock usage in the Ethernet demonstration application. A lock named `eth_test_lock` is detected. It protects buffers which are shared to facilitate UDP testing. The tool displays the statistics in a table and also reports two inefficient lock behaviors to the user: spinning with atomic operations and thread preemption while holding a lock. It is noticed that the current lock is a test-and-set lock and interrupts are not disabled inside the critical section. Two optional improvements are available, i.e. applying the test-test-and-set lock and disabling interrupts inside the critical section.

With these two options, four alternative implementations, namely *test-and-set without protection* (without any improvement), *test-and-set with protection*, *test-test-and-set without protection* and *test-test-and-set with protection* are investigated. The holding time and the waiting time for the four

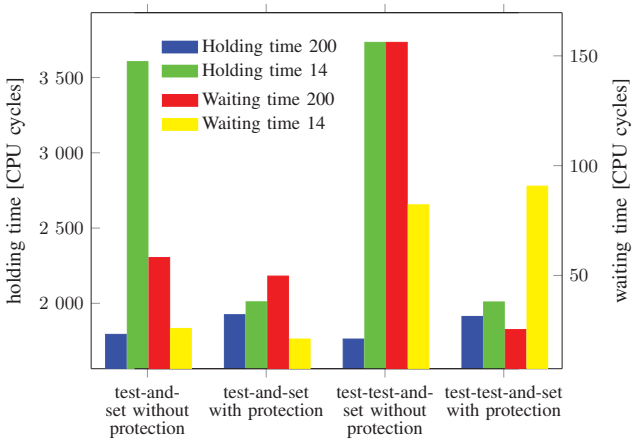

Fig. 1. The holding time and the waiting time of thread 14 and thread 200 owning `eth_test_lock`

alternatives are shown in Fig. 1 (average over ten measurements).

Due to the fact that interrupts are not disabled in alternatives *test-and-set without protection* and *test-test-and-set without protection*, thread 14 can be preempted while holding the lock. This results in extra-long holding time as shown in Fig. 1, while the holding time of thread 200 does not change much among the different alternatives. As the holding time of thread 14 without protection is much larger than the one with protection, the blocking chance of thread 200 is also higher, resulting in longer waiting time of thread 200 in Fig. 1. The waiting time of thread 14 with test-test-and-set lock is always higher than the first two alternatives. This is because the spinlock is located in the CPU1's local memory and thread 200 accesses this lock much faster than the remote thread 14. It is easier for a local thread to acquire with the test-test-and-set implementation. *test-test-and-set without protection* has extremely long waiting time for both threads. They are having many collisions in this timing. According to the results, protection is necessary to avoid preemption while holding a lock. The test-test-and-set lock gives the local thread a better chance. For this Ethernet demonstration application, the test-and-set lock with protection could be selected to decrease the waiting time of the thread 14.

The experience in profiling the application on Erika indicates the feasibility of the proposed OS-independent lock detection approach. The above profiling analysis explains how to apply the proposed approach to increase the system transparency and improve the lock usage. The lock usage situation is complicated and sometimes out of developer's expectation. The suitable lock type depends on different working conditions. A profiling tool clarifying this complicated situation is therefore necessary. Especially for embedded applications, such an OS-independent tool is convenient to use.

V. Conclusions

As a commonly used synchronization method, spinlocks are widely applied for embedded multi-core systems. There are various types of spinlocks and lock profilers are beneficial to proper spinlock selection. In this paper, an OS-independent lock profiling approach for binary semaphore spinlocks is proposed for embedded multi-core systems. It makes use of existing hardware tracing and common spinlock characteristics to detect spinlocks without relying on specific OS knowledge. Many spinlock statistics and inefficient lock behaviors are reported by this approach. The proposed approach is implemented as a post-processing tool running on a PC. According to the case study, the implemented tool helps to clarify the spinlock usage and to optimize the performance, which shows the effectiveness of the proposed approach. In the future, more types of spinlocks can be supported by extending the common characteristics and additional metrics can be designed.

Acknowledgment

This work was funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi) as part of the project "SoC Doctor." The responsibility for the content remains with the authors.

References

[1] T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.
[2] L. Rudolph and Z. Segall, *Dynamic decentralized cache schemes for MIMD parallel processors*. ACM, 1984, vol. 12, no. 3.
[3] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.
[4] H. Guiroux and R. Lachaize, "Multicore locks: The case is not closed yet," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 649–662.
[5] "AUTOSAR Release 4.2.2," http://www.autosar.org/, [Online; accessed 14-Nov-2016].
[6] *ERIKA Enterprise Manual*, Evidence S.r.l.
[7] P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Reactive spin-locks: A self-tuning approach," in *8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)*. IEEE, 2005, pp. 6–pp.
[8] H. Shojania, "Hardware-based performance monitoring with vtune performance analyzer under linux."
[9] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
[10] J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, and N. Tallent, "Hpctoolkit user's manual."
[11] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 353–364.
[12] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen, "Halock: hardware-assisted lock contention detection in multithreaded applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 253–262.
[13] R. Bryant and J. Hawkes, "Lockmeter: Highly-informative instrumentation for spin locks in the linux kernel," in *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, 2000.
[14] Z. Benavides, R. Gupta, and X. Zhang, "Parallel execution profiles," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 215–218.
[15] J. Zhang, Y. Dong, and J. Duan, "Anole: a profiling-driven adaptive lock waiter detection scheme for efficient mp-guest scheduling," in *2012 IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 504–513.
[16] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the free-lunch profiler," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 291–307.
[17] "Profiling java.util.concurrent locks," https://www.infoq.com/articles/jucprofiler, 2010, [Online; accessed 31-July-2016].
[18] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Debug support, calibration and emulation for multiple processor and powertrain control socs," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*. IEEE Computer Society, 2005, pp. 148–152.
[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.
[20] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 256–267, 2004.