Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design

Hendrik Borghorst
Embedded System Software
Computer Science 12, Technische Universität Dortmund
Email: hendrik.borghorst@udo.edu

Olaf Spinczyk
Embedded System Software
Computer Science 12, Technische Universität Dortmund
Email: olaf.spinczyk@udo.edu

***Abstract*—Real-time operating systems have been around for some time, but they are never designed for being used on modern multi-core processors with unpredictable timing behavior. An important source of unpredictability is the different timing between the processor and the DRAM-controller. Operating-system-based cache management is one possibility to reduce the timing variations of the processor by controlling the code and data which resides in the cache. The cache eliminates the timing differences between the memory and the processor.**

I. Motivation and Related Work

With increasing complexity of today's multi-core processors, their timing behavior gets more unpredictable, which leads to big fluctuations of the execution times for tasks and operating system functions like interrupt handling. This means that the overall response time of a system depends on the timing behavior of all the shared resources like the caches or buses [1]. This problem prohibits the use of such systems for time-critical applications like cyber-physical systems. Cyber-physical systems need to react on certain events within a predictable time bound. Therefore it is critical that the overall response time of the operating system is stable. Different timings of the main processor and the memory can be neutralized by the use of caches. But caches can introduce new problems like unwanted cache eviction which would also lead to unstable execution times.

Cache partitioning can be used to prevent cache eviction for multi-task or multi-core applications. Cache preloading can be used to prevent timing variations caused by simultaneous bus accesses from multiple participants.

R. Mancuso et al. proposed a cache management framework for applications running on the Linux operating system [2]. The approach, presented in their paper [2], loads specific application code and data to a partition of the shared cache and locks it afterwards. This approach shows an significant reduction of the application's execution time variation. Their method eliminates the timing variations caused by shared caches and random memory accesses. In contrast to this method the later presented approach works on the level of the operating system. The advantage of managing the cache within the operating system allows operating system functionality to be predictable as well.

J. Liedtke et al. worked on operating system controlled caches for single-core processors [3]. They used a technique called cache coloring to reduce the risk of cache eviction for multi-tasking applications. They could show that it is possible to reduce the variation of the execution time with the use of cache partitioning. Nonetheless their work is based on single-core systems and does not consider the properties of a multi-core system with shared resources such as the memory and buses.

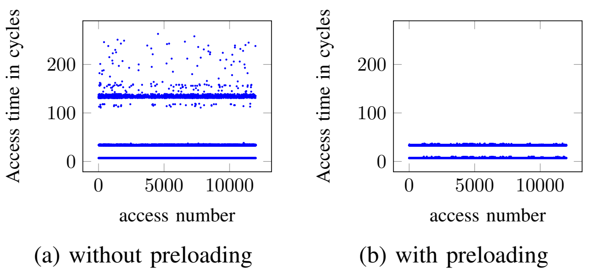

Fig. 1: Comparison of data access times (64kB range)

As a preparation to proof the later presented operating system concept, we created a prototype operating system which used a basic cache management to preload tasks on activation. To evaluate the concept of cache resource management, we ran four tasks on a dual-core *ARM Cortex-A9* processor. Each task was confined to a distinct memory area and accessed random memory addresses with and without preloading and locking of the shared L2 cache. The results of this test are shown in Figure 1 where single memory accesses are shown with their corresponding access time. The diagram in Figure 1a illustrates that there is a very high fluctuation of memory access times. For comparison Figure 1b demonstrates that the preloading of the data to access shows a significant reduction of the previously mentioned access time fluctuation. The benchmark was done for a memory area of 64 kB per task which is twice the size of the level 1 cache, so there are already level 1 cache misses which are represented by the upper one of the two distinct lines in the diagrams.

The execution times for the cache preloading itself were tested separately. It was measured that the execution times are proportional to the preloading size, if it is guaranteed that only one processor core is preloading at the same time. This knowledge is crucial to the whole idea to get the system predictable.

With this knowledge it is possible to create an operating system that takes control over the content inside the cache so that the execution times for operating system functions and interrupt handling become predictable. To achieve this we present an operating system model which is designed with the sources of unpredictability in mind.

II. Operating System Model

The idea of the new operating system is to sort out some problems that existing real-time operating systems present when they are executed on modern multi-core architectures that utilize some shared resources like caches and memory buses.

Modern multi-core processors often include shared caches that are structured as associative caches which features multiple cache ways to reduce the cache miss rate. Each cache way represents a part of the whole cache. The target architecture used for this paper features an shared second level cache with 16 cache ways with 64 kB capacity each.

To solve the issues of unpredictable caches and memory access latencies, the operating system and the applications have to fit inside the partitioned shared L2 cache. One solution to achieve this, is to divide the system into small pieces. We call each of these pieces *operating system component* (OSC). The implementation of the operating system is done in a highly modular way so that we can define very fine granular components. These components can than be grouped together into larger components to be optimal for the desired target platform. The optimal component size depends on the specific sizes of the cache structure of the hardware. For example a component needs to be smaller than the biggest shared cache and not to small which would effectively be the same like one random memory access. A good size would be a multiple of the cache way size.

One problem with existing embedded operating systems is that there is usually only one stack per core when using operating system functions. This makes it hard to predict where the local data is located when the processor jumps to operating system code. This could lead to cache eviction if operating system functionality is requested. To solve this, each OSC contains its own stack by what we enable the operating system to contain all code and data on the level of OSCs.

Another problem with existing solutions is that normal function calling allows no control over the data and control flow which could lead to cache eviction problems. To solve this the new operating system prohibits direct data passing between OSCs. Instead the system operates on a strictly event-based nature. These events are handled by the operating system so that it can control the contents of the cache.

Each OSC can define *input triggers* which will activate a specific OSC. Each input trigger needs a function which is called after the OSC is activated. To activate these input triggers, output events, that each OSC can define, are required. These events can be connected to the input triggers of other OSCs. The creation of the connections between events and triggers of OSCs is done during the time of compilation. For

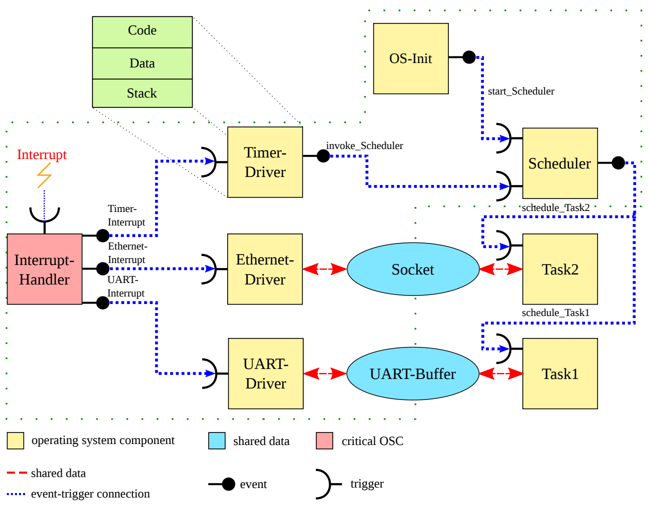

Fig. 2: Operating system model with critical/non-critical components

performance reasons this is a static linkage with hard coded function pointers. If an OSC wants to send an event it needs to do it by the use of a *system call*.

To solve the issue of uncontrollable data flow, the operating system specification allows shared data between two or more OSCs. Shared data must stay inside the cache until no OSC needs it anymore. These shared data objects need to be cache-aware by design so that the application developer needs to make the data structures efficient on constrained space. There are several approaches on cache-aware data structures and their optimizations. For example T. Chilimbi et.al. present a way to make pointer-based data structures cache-aware [4]. They introduce a method which can optimize different data structures, that are based on indirect data accesses, via a modified version of the dynamic memory management method *malloc*. In addition they present a way to specifically optimize tree-based data structures so that they reduce the number of cache-misses drastically. Those methods could be integrated within the operating system so that the application developer is presented with an API that takes care of the cache prefetching. It should be noted that the focus of this operating system is not on heavy data computation but on comparable small real-time task-sets with data structures that fit into the shared caches.

Another critical problem of the system is the interrupt handling because it is impossible to predict when interrupts arrive. Therefore it is critical that the whole minimal first stage of the interrupt handling is locked permanently to the cache. The first stage would then emit an event with the interrupt number. This event is handled like any other event. This ensures that the interrupt handling stays predictable by assuring that the unpredictable part always remains inside the cache. The preloading of the remaining interrupt handling is by definition predictable. With this model a periodic behavior is also possible to achieve by using a timer with a periodic configuration but the system is not limited to periodic configurations.

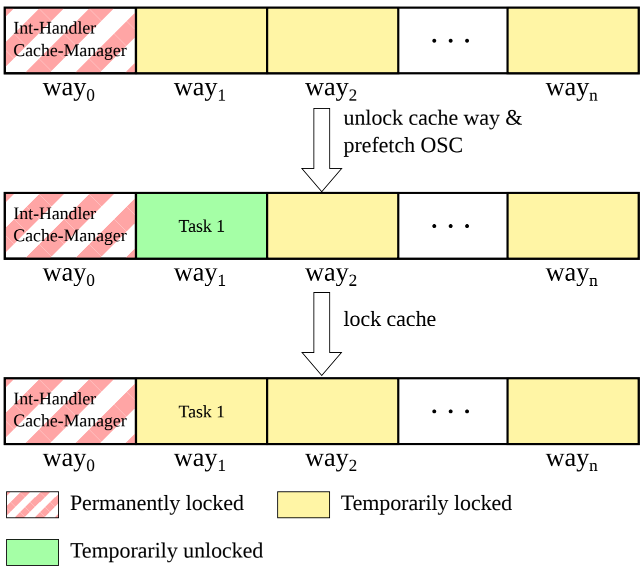

Fig. 3: Cache way states during an OSC-transition

A schematic representation of the presented operating system model is shown in Figure 2. The figure visualizes how different OSCs could be connected with each other. As highlighted in the figure, each OSCs consists of an separate code, data and stack segment. Events connect OSCs with each other as visualized by the punctuated lines. The *Interrupt-Handler* is marked in red because it is time critical and needs to stay locked permanently. The ellipsis in Figure 2 represents a shared data object. The figure shows an operating system which uses a timer component to emulate a time-based behavior. The operating system itself is not limited to time-based events and could react predictable to sporadic events as well because the critical part of the interrupt handler handles interrupts within guaranteed time bounds.

This operating system needs a special kind of scheduler because it does not schedule tasks directly but needs to schedule the execution of events. Events can be prioritized so that time critical events are handled before uncritical events. The scheduler needs to minimize the cache eviction and data flow from the main memory as well. As a result of this it needs to optimize which OSCs are active inside the cache and which can be swapped away.

Figure 3 shows the different states of the cache during the execution of the system. It represents an simplified version of a cache structured into n cache ways. Each cache way can be locked individually. Therefore it is possible to control the cache content manually by unlocking only one cache way at once which guarantees that the data is allocated to that specific way during prefetching. The uppermost row visualizes the state in which only the critical parts are locked and loaded inside the cache. This is the state in which the operating systems resides after successful initialization. The row in the middle of Figure 3 represents the cache state in which an OSC was prefetched, right before the needed cache way gets locked again. The cache management unlocks only the cache ways that are needed for the OSC to activate. This is not limited to only one cache way per OSC. It is also possible for OSCs to spread across multiple cache ways. In this case the cache ways would be unlocked and prefetched consecutively. After successful prefetching of the OSC the cache management locks all cache ways again to prevent cache eviction from happening which is the state of the bottom row in Figure 3.

III. Hardware Platform

For now the operating system needs special hardware features to control the cache. Cache locking is needed to prevent cache eviction when loading new OSCs. For the purposes of evaluation we used a Texas Instrument OMAP4460 ARM processor [5] that uses an external level 2 cache controller and is compatible to the ARM Cortex-A9 processor. This cache controller has sophisticated control features like cache lockdown by cache way and by core [6]. This means that it is possible to control in what cache way new cached data gets allocated. The processor was clocked at 921 MHz during the experiments.

The level 2 cache features 16 cache ways, each with a size of 64 kB. Thus a optimal size for the OSCs would be 64 kB or multiples of this value. For now the OSCs get aligned to this size during the linking process which makes it convenient to prefetch those components to specific cache ways.

IV. Ongoing and future work

The presented operating system is just a proof of concept for now. We evaluated that it is possible to take control over the contents of the shared cache with a basic cache control implementation that prefetched data and code to the cache and locked the cache afterwards. Another thing we measured is the required time to prefetch bulk data. Our results show that we can achieve a prefetch time which is linear to the prefetch size. It was measured that the prefetch time per byte is around 8 clock cycles if more than 128 bytes are prefetched in a bulk transfer. For the component size of 64 kB this bulk transfer require around 0.57 ms.

In the future we intend to focus our research on some specific topics regarding the operating system model. One part of this will be the scheduling of the event dispatching. There are several optimization criteria for the scheduling. For instance the minimization of cache evictions, to maximize the overall processor utilization and to keep the overall response time of the system minimal.

Furthermore we intend to analyze the timing behavior of the operating system. This includes analysis of the transition times, prefetch times and the OSC function execution times to guarantee that the execution time of the whole system will stay inside a time bound.

Also the operating system needs a good software development model. It is important that the implementation of the event-based system is not overly complicated. One possible solution for this could be the use of an aspect-orientated language like *AspectC++* [7].

Another topic to explore is how to extend the supported hardware base. One potential substitute for locking critical OSCs inside the cache could be a static ram which many new embedded processors include. It may also be possible to isolate one core of the system to interrupt handling. This would mean that the first stage interrupt handler should not be evicted from the level 1 cache if it is small enough. For systems lacking the support for cache locking the use of traditional software-based cache partitioning algorithms is necessary [8].

Finally the operating system needs evaluation under several circumstances. We expect that the manual management of the cache content will introduce some overhead on the computational performance of the system. Therefore an comparison with existing operating systems like RT-Linux [9] or RTEMS [10] is needed. The overall system response time also needs evaluation with various workloads.

V. Conclusion

This paper presents a possible solution for the unstable execution times of modern multi-core systems on the level of the operating system. This is done by manually controlling which data and program code resides in the cache. By this the operating system shifts the unpredictability of random DRAM-accesses to predictable bulk memory transfers. To realize this the operating system operates on a event-based nature and is structured as a set of OSCs, which can be loaded into the cache on-demand or permanently based on a cache scheduling strategy. At the moment the operating system only exists as a proof of concept but we intend to explore this concept further.

References

[1] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, “Identifying the sources of unpredictability in COTS-based multicore systems,” in *2013 8th IEEE International Symposium on Industrial ES (SIES)*, June 2013, pp. 39–48.

[2] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 45–54.

[3] J. Liedtke, H. Haertig, and M. Hohmuth, “OS-controlled cache predictability for real-time systems,” in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS ’97)*, ser. RTAS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 213–.

[4] T. Chilimbi, M. Hill, and J. Larus, “Making pointer-based data structures cache conscious,” *Computer*, vol. 33, no. 12, pp. 67–74, Dec 2000.

[5] “OMAP4460 ES1.x Technical Reference Manual,” http://www.ti.com/lit/pdf/swpu235, accessed: 2015-02-20.

[6] “PL310 Cache Controller - Technical Reference Manual,” http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246a/DDI0246A_l2cc_pl310_r0p0_trm.pdf, accessed: 2015-04-18.

[7] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, “AspectC++: An aspect-oriented extension to C++,” in *Proceedings of the 40th International Conference on Technology of OO Languages and Systems (TOOLS Pacific ’02)*, Sydney, Australia, Feb. 2002, pp. 53–60.

[8] F. Mueller, “Compiler support for software-based cache partitioning,” *SIGPLAN Not.*, vol. 30, no. 11, pp. 125–133, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/216633.216677

[9] “Real-Time Linux Wiki,” https://rt.wiki.kernel.org/index.php/Main_Page, accessed: 2015-04-29.

[10] A. Colin and I. Puaut, “Worst-case execution time analysis of the RTEMS real-time operating system,” in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 191–198.