Case Study: On-Demand Coherent Cache for Avionic Applications

Arthur Pyka*, Mathias Rohde*, Pavel G. Zaykov†, Sascha Uhrig*
{arthur.pyka, mathias.rohde, sascha.uhrig}@tu-dortmund.de
pavel.zaykov@honeywell.com

*Technical University of Dortmund, Germany
†Honeywell Aerospace, Advanced Technology Europe, Czech Republic

Abstract. In hard real-time systems, such as avionics, there is demand for high performance. A way to meet performance demands is by parallel computation on multicore systems. A main contributor for the application performance on multicore systems is size and type of cache memory. For multicore hard real-time systems, the usage of cache memory is problematic. Time-critical applications need reasonable worst case execution time (WCET) estimations. Caches complicate the estimation of memory access latencies, in particular for parallelised applications, since caching of shared data requires a cache coherence mechanism and typical coherence protocols lack of timing predictability. Hence, in hard real-time systems caches are not used for shared data, or not at all.
In this paper, we present a case study of the On-Demand Coherent Cache (ODC^2) applied on the 3D Path Planning application, as representative of the avionics domain. In our experimental results, we illustrate how a characteristic avionic application can benefit from using the ODC^2 with different functionalities in a multicore environment. With ODC^2 a significant speedup can be observed by permitting fast and coherent accesses to shared (and private) data, while preserving hard-real time requirements.

1 Introduction

In the hard real-time domain, requirements regarding high performance go along with the demand for timing predictability. The characteristics of hard real-time applications can be summarised to the indispensable necessity to hold given timing constraints and adequate system hardware to be executed on. Timing constraints are guaranteed by a static timing analysis which estimates the worst case execution time (WCET) of the application running on a particular system. Likewise a feasible estimation of memory access latencies depends on a highly predictable behaviour of the employed memory hierarchy. In multicore systems, memory is accessed by many cores and represents a critical shared resource. Therefore, multicore architectures were rarely used for hard real-time applications. But the constantly growing demand for computational power in the real-time domain pushed multicore systems into the focus and forces developers to find applicable solutions.

In multicore systems, cores simultaneously access shared memory and accesses to that memory may traverse a long way through the network (e.g. mesh or torus).

The desired benefits in decreased execution time which go along with parallelised computation, can be essentially narrowed by immensely raised memory access delays. Therefore, we expect a private cache memory per core to improve significantly the performance of applications running on a multicore system. Furthermore, cores involved in parallel execution cannot cache shared data without further effort; a cache coherence mechanism has to ensure that accesses to shared data are performed in a coherent way.

The existing cache coherence mechanisms provide solutions for performance orientated systems, but result in poor timing analysability of the application. Existing cache coherence protocols, aimed at increasing average case performance, induce unpredictable influences at the timing behaviour of memory accesses. The common mechanisms, invalidation- and update based, snooping and snarfing and directory-based, all rely on interaction among the core's caches. The state and content of a cache line is modified externally via actions triggered by other cores. These cache modifications result in unpredictable presence of data content and mutated timing behaviour as stated in the following:

- In invalidation-based coherence protocols, a cache line contains data shared with other cores. If another core intends to write on that data, then a cache line can become invalid at any time. Thus, the lifetime of a shared cache line is practically impossible to predict. The external invalidation can also affect the state of the replacement policy (e.g. Least-Recently-Used) and has a negative influence on the prediction of the lifetime of cache lines with private data.
- In caches following the write-back strategy, a modified shared data can be hold exclusively in one core's cache. If another core intend to access the same data, the possessing core writes the data to main memory (e.g. employ a snooping protocol) or delivers the data to the requesting core (e.g. employ snarfing protocol). In both cases, both cores experience extensively increased worst case access delays.
- In general, any coherence protocol that allows external modification to a cache, provokes the situation that any access of a core to its cache (even a cache hit) can be in competition with an external access from another core, resulting in additional delays for one of these accesses.

For static timing analysis, an adequate knowledge of the cache content and bounded access latencies are a precondition. The aforementioned unpredictable influences on the timing of cache accesses hinder a feasible worst case execution time estimation. Massive overestimations of cache access delays make the calculated results inapplicable for hard real-time systems. A coherence mechanism, suitable for hard real-time systems needs to be defined by the absence of external intervention. The timing behaviour has to be predictable from the core's point of view.

The *On-Demand Coherent Cache (ODC^2)* mechanism introduced by Pyka et al. [1] provides a predictable timing behaviour suitable for hard real-time systems. The mechanism is free of any interaction between caches in a multicore, thus unpredictable interferences do not appear. ODC^2 relies on software support to trigger hardware activities preserving coherence accesses to shared data. This paper illustrates how a parallelised implementation of an industrial application from the avionics domain takes advantage of the timing predictable cache coherence mechanism. We demonstrate how the application has to be extended with instructions to control the ODC^2 and how it cooperates with the used synchronisation techniques. We evaluate the application running on a multicore simulator using the ODC^2 in two different operating modes and compare the results to uncached accesses to shared data and to a single core execution.

The rest of the paper is organised as follows: Section 2 introduces the related work. Section 3 describes the ODC^2 mechanism. Section 4 presents an avionic use-case with ODC^2. Section 5 outlines the evaluation environment and conducted experiments. The paper concludes with Section 6.

2 Related work

Cache coherence mechanisms spawned a huge amount of approaches designed with regard to average execution time and optimisations have been proposed. Contrary to that, the hard real-time domain relies on worst case execution time. Timing predictability is the main criteria, which necessitates a basic redesign of handling coherent data accesses. Many approaches to increase timing predictability of memory accesses have been introduced. In the following we mention approaches related to this work.

Edwards and Lee [2] conceptualised the paradigm shift towards predictability with the PRET machine. In the PRET architecture cache-related timing interferences are avoided by using scratchpad memory. While the prediction of a cache state is highly affected by the replacement strategy (see [3]), scratchpad memory allows user-controlled allocation. Many approaches to dynamically allocate scratchpad memory have been proposed (e.g. [4]). Wasly et. al. [5] proposed a dynamic scratchpad memory unit, managed by the operating system with the help of Direct-Memory-Access (DMA) transfers. Aimed to multithreaded cores, these approaches do not solve coherence problems with shared data on multicore systems.

An often applied idea is to isolate accesses to shared cache. Cache partitioning mechanisms can provide isolation and prevent conflicts when cores access a shared cache. With cache-colouring, different tasks use different cache sets to maintain cache isolation. Ward et. al. [6] proposed combined cache-colouring with cache-locking to allow protected access to coloured cache ways. Similar to ODC^2, cores can access and cache shared data, controlled by a synchronisation technique. But this approach is bounded to assumptions (operating system support, no self-eviction and others) which may not be complied by the used system. However, cache-colouring is not applicable for caching shared data. Vera et. al. [7] proposed a mechanism to make data caches predictable for multitasking hard real-time systems, combining cache-partitioning and cache-locking. But cache coherence related problems are not considered in this approach.

Self-invalidation is a mechanism to avoid the holding of stale copies of shared data without inter-cache communication. Self-invalidation extends a coherence protocol with the forced invalidation of cache lines triggered by software operations. Lebeck et. al. [8] proposes dynamic self-invalidation, in which blocks are automatically selected to become invalid in the following. A mechanism which is closely related to this work was proposed by Ashby et al. [9]. A filter-based selective self-invalidation is applied at synchronisation points, considering cache lines that have been modified by other cores. Since the invalidation selection depends on the other cores behaviour, it lacks of predictability and is not suitable for time-critical systems.

Other related techniques are presented by Park et al. [10] and Ros et al. [11]. The former augment the memory hierarchy with an additional local cache only for write-shared data at page granularity, on which software coherence mechanisms can be applied. The latter classifies data as private or shared based on the used memory page. Since the class of a page is determined dynamically depending on the accessing cores, a complex process is required for a transition of a page from private to shared. Even though no coherence messages are required, the mentioned transition process impede a tight timing analysis.

A common workaround in real-time systems is to disable the caching of shared data. A way to achieve it is to use special instructions provided by the cores or to store shared data in non-cacheable memory areas. Other approaches are based on the idea of assigning *cacheable* and *non-cacheable* attributes to memory accesses, supported by a compiler analysis [12].

3 The Time Predictable ODC^2

The goal of ODC^2 is to provide a fast access to private and shared data in a coherent way. In ODC^2 shared data is preserved as long as necessary and flushed back to the shared memory after it is used. To identify instruction sequences which access shared data, the ODC^2 depends on synchronisation techniques which are anyway needed in parallel applications to protect accesses to shared resources and to show a deterministic behaviour. Assisted by these synchronisation primitives the application must be extended by operations that control the ODC^2 cache controller.

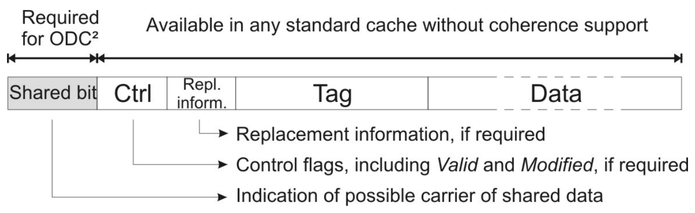
Required for ODC²
Available in any standard cache without coherence support
Shared bit
Ctrl
Repl. inform.
Tag
Data
Replacement information, if required
Control flags, including Valid and Modified, if required
Indication of possible carrier of shared data

Fig. 1. Block frame of ODC^2 including the *Shared bit*.

ODC^2 supports two different working modes, the *private mode* and the *shared mode*. The activation and deactivation of the *shared mode* is triggered via accesses to cache control registers.
In private mode, the ODC^2 acts as a standard cache controller without any coherence functionality since in this mode no accesses to shared data are allowed. At

the time a code section with accesses to shared data is entered, the shared mode is activated and the ODC^2 shows additional functionality. If a cache miss occurs, the loaded cache line is marked as shared by the *shared* bit. This additional information inside a block frame (see Figure 1) identifies the cache line as a potential carrier of shared data on which coherence actions are required. Notice that the cache is not aware of the status (private or shared) of the loaded data, so also private data loaded during shared mode will be marked as shared.

At the time of deactivation of *shared mode*, all subsequent memory accesses are stalled and the cache controller performs a *restore procedure*. In case of a *write-back* policy, all cache lines marked as *shared* and *modified* are written back to the main memory. In case of a *write-through* policy, no additional write-back is required. Independent from the write policy, all cache lines marked as *shared* are invalidated. After this operation all modified shared data is flushed to the main memory and no shared data remains in the cache. While the costs of ODC^2 control operations correspond to simple arithmetic instructions, the memory accesses in the *restore procedure* produce additional overhead. The amount of this overhead is bounded and fully predictable by means of a cache analysis.

Two additional functionalities of the ODC^2 can be activated, to account for special circumstances:

– **Address checking:** Seeing that some of the potential carriers of shared data can hold private data instead, this private data is needlessly marked as shared and will be invalidated (and flushed back) with no need. To minimise this negative effect, the address of an access can be checked during shared mode. By doing this, it is possible to identify and to exclude specific memory sections from being marked (e.g. stack) or even restrict the marking to a special *shared* section.
– **Forced write-through:** In case of a write-back strategy, write accesses to a specific shared data during shared mode provoke a flush-back of the cache line containing that data during the restore procedure. Depending on the access pattern of the application, it cannot be excluded that neighbouring data, which is written back together with the modified data inside the same cache line, has been modified in a copy of another core's cache, resulting in an incoherent condition. To avoid this, write operations to shared cache lines during shared mode can be performed as a write-through operation, with the effect, that no cache lines need to be flushed back after shared mode.

As mentioned above, the extension of the control operations is done with respect to the synchronisation techniques used in the application. ODC^2 is not restricted to a single technique. Since ODC^2 aims to hard real-time systems, only selected synchronisation techniques are considered.
The shared mode can be activated inside a critical section. This section can be protected by a locking technique like *spin lock*, *mutex lock*, *ticket locks* or *semaphores*. The control operations can be also bundled with these locking primitives, to automatically control the ODC^2 when getting and releasing the lock.
Instead of blocking access to code sections, synchronisation is often handled with *barriers* (and *conditionals*) permitting simultaneously running the same code while

accessing disjunct data. Since it is guaranteed that multiple cores do not modify the same portion of shared data between two barriers, ODC^2 can be used here as well. In general, ODC^2 supports any synchronisation mechanism which fulfils the condition that, for all shared data which is accessed in code sections encapsulated by the entering and leaving of the shared mode, solely one core is allowed to access and modify specific portion of data at a time. Read-only accesses of multiple cores covered by the shared mode are allowed.

4 An Avionic Case Study: 3D Path Planning Application

In this section, we consider an avionics case study as a representative of the hard real-time systems. With the introduction of augmented reality and increased system integration in the avionics domain, multicore systems become a feasible solution to meet those high-performance demands. As a result, we consider as highly relevant to evaluate the applicability of the time-predictable ODC^2 cache on a multicore system, employed in the avionics domain.

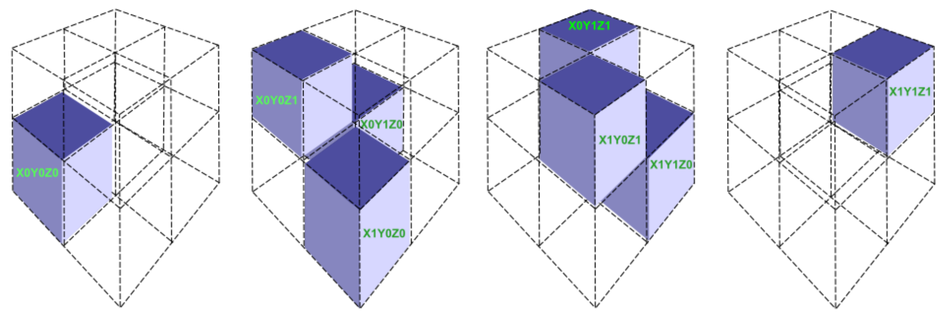

Fig. 2. 3DPP application: scheduled compartments in the pipelined operation

As a representative of the avionics domain, we consider the 3D Path Planning (3DPP) application, delivered by Honeywell International. The 3DPP application is used in the navigation of Unmanned Aerial Vehicles (UAV). It computes the path between the current UAV position (obtained from a satellite) and the destination position (defined by a user), while avoiding obstacles in a 3D environment. The 3DPP application receives an obstacle map and waypoints as an input and calculates the desired position and velocity. The most computational intensive part of the application is highly suitable for data parallelisation. The 3D environment can be divided into multiple compartments (3-dimensional sub-grids) that can be processed by multiple cores in parallel. All cores share the surrounding information (obstacle map and waypoints) and cores with neighbouring compartments exchange data during computation.

The threads in the 3DPP application are mapped to the sub-grids and perform the computation in a pipeline manner, as visualised in Figure 2. While parts of the computation are free of data dependencies and can be completely processed in parallel, the main threads (respectively cores in a multicore archtiecture) from different pipeline stages require synchronisation with the neighbouring cores. Due to the mutually exclusive memory access, fine-grained synchronisation techniques like mutexes are not demanded at this level. The parallel computation is synchronised via barriers or conditional variables.

```
-- ... --
-- Synchronisation Point --

while (still_data_to_process){

    -- Synchronisation with previous sub-grids --

    ENTER_SHARED_MODE
    (-- Computation step of a sub-grid (e.g. X0Y0Z0) --)
    EXIT_SHARED_MODE

    -- Synchronisation with next sub-grids --
}
-- Synchronisation Point --
-- ... --
```

Fig. 3. An exemplary source code extension for ODC^2 in the 3DPP application

Applying the ODC^2

We perform the following two operations, when we apply the ODC^2 in the 3DPP application:

1. As already introduced in Section 3, the ODC^2 code extensions shall be introduced with regard to the synchronisation technique. This implies that synchronisation variables (e.g. lock variables, conditionals etc.) are not allowed to be cached. Thus, all synchronisation data in the application must be mapped into an uncached memory region.
2. The ODC^2 code extensions shall be introduced on the shared data structures. All shared data accesses has to be covered by the shared mode and accordingly encapsulated by the control operations. At synchronisation points in the application, no shared data must be held inside a cache and the main memory has to be updated. In the 3DPP application, parallel computation with accesses to shared data takes turns with synchronisation points. At the end and at the beginning of these synchronisations, shared mode has to be activated and deactivated, respectively. In Figure 3, we introduce an exemplary source code extension (ENTER_SHARED_MODE, EXIT_SHARED_MODE) in a simplified manner. It shows the annotation of the pipelined sub-grid computation as visualised in Figure 2.

In the 3DPP application, the course-grained synchronisation employes barriers and conditional variables, as a result multiple threads/cores do not modify the same shared data at a time. But depending on the size of a cache line, accesses of multiple cores to different shared variables can aim to the same cache line. Therefore, the ODC^2 has to perform write accesses to shared data as a *forced write-through* access, as described in Section 3. This functionality ensures coherent accesses independent from the selected cache setting.

5 Evaluation

We evaluate the execution time and memory accesses produced with the 3D Path Planning application on a multicore platform using ODC^2. The evaluation is performed within the *parMERASA simulator*, a many-core simulation platform, developed in the parMERASA project [13]. It is based on the SoCLib simulation platform [14], a collection of SystemC hardware modules (processors, networks, memory, etc.). Our basic platform consists of multiple clusters, connected via routers in a form of a 3x3 mesh. A cluster itself is composed by a local memory and several cores which are connected via a local crossbar. A core is a PowerPC processor with instruction and data caches of 16 kB each, organised in 1024 direct-mapped sets.

Three different platform configurations are chosen for the evaluation setting:

1. **Single cluster**: The parallel computation is localised in a single cluster. All cores share one local memory for instruction and data (private and shared) and accesses solely traverse the local crossbar.
2. **2x2 mesh**: This configuration is limited to a 2x2 mesh network. A maximum of two cores per cluster are involved in the computation. The shared data (including synchronisation variables) is located in the lower left cluster. This allows fast access for the core situated there, which has to perform special tasks in the application, but imply longer latencies for the distant clusters.
3. **3x3 mesh**: In this configuration the complete 3x3 mesh is used and only one core per cluster is involved in the computation. The shared data is now located in the middle cluster, permitting fair access latencies to all cores.

For each configuration, the 3D Path Planning application is configured from 2 up to 8 cores involved in the parallel execution (1 thread per core). As objectives we calculated the execution times and the amount of different accesses to the memory. We distinguish between uncached accesses, cache misses and forced write-through accesses with ODC^2. The results of the execution with *ODC*^{2} are compared with a platform using an incoherent cache and uncached accesses to shared data as a common solution for time-critical systems (denoted as *Uncached*). Accesses to instructions and private data are cached in all three platforms. As a third variant, the ODC^2 is applied with the address checking functionality (*ODC*^{2} *AC*). The memory layout of the simulator provides separate memory sections for private and shared data, so the marking of cache lines can be restricted only to shared data. As a general reference to the parallel execution, the results are compared to the execution on a single core with full cache usage to private and shared data (*single core*).

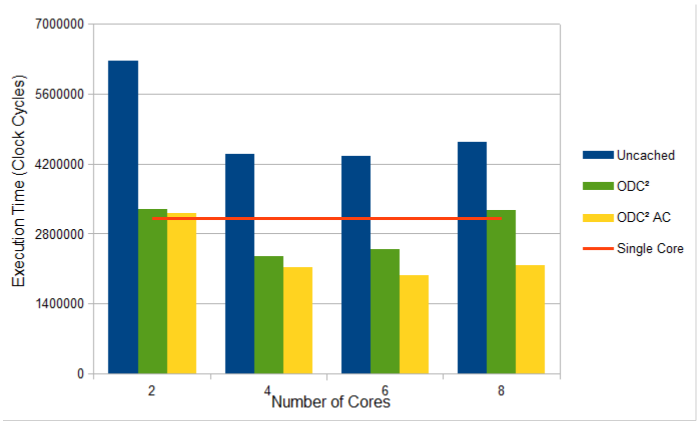

Fig. 4. 3DPP execution time for single cluster configuration.

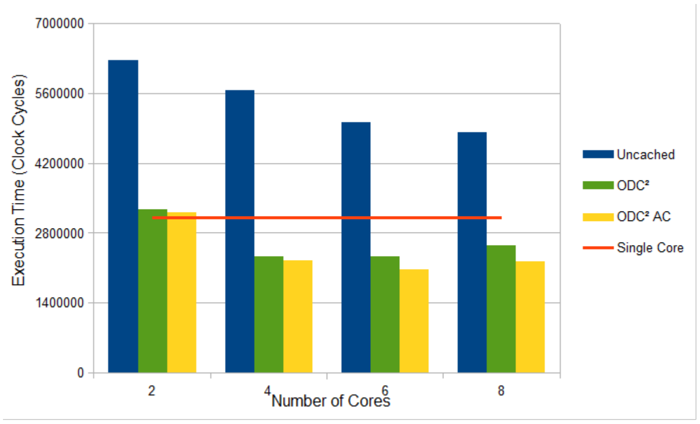

Fig. 5. 3DPP execution time for 2x2 mesh configuration.

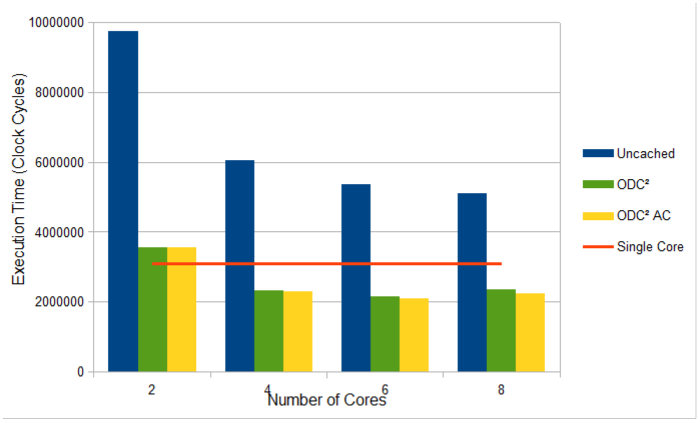

Fig. 6. 3DPP execution time for 3x3 mesh configuration.

5.1 Experimental Results

The results of the execution of the avionic application show that the usage of the timing predictable On-Demand Coherent Cache permits a significant performance speedup compared to uncached access to shared data. This can be observed for all three configurations and across the different number of used cores. Additional profit can be attained with the address checking functionality ($ODC^2\ AC$). Furthermore, in the majority of cases the parallel execution using ODC^2 offers a performance speedup compared to a fully cached single core execution. However, the uncached execution results in a highly increased execution time and thus, is no feasible approach for parallelisation. The gain of a parallelised execution depends

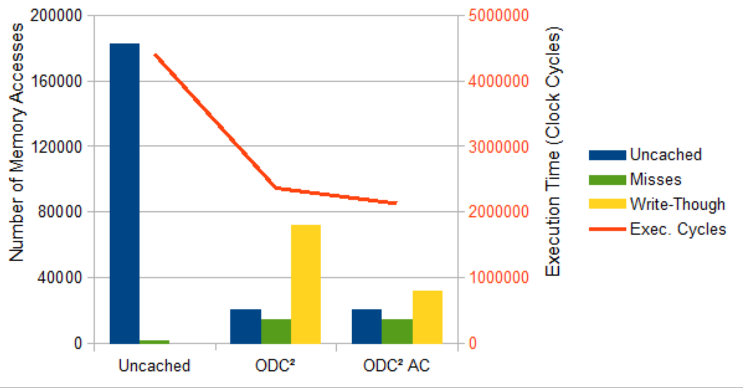

Fig. 7. Amount of different memory accesses for execution with 4 cores in single cluster configuration.

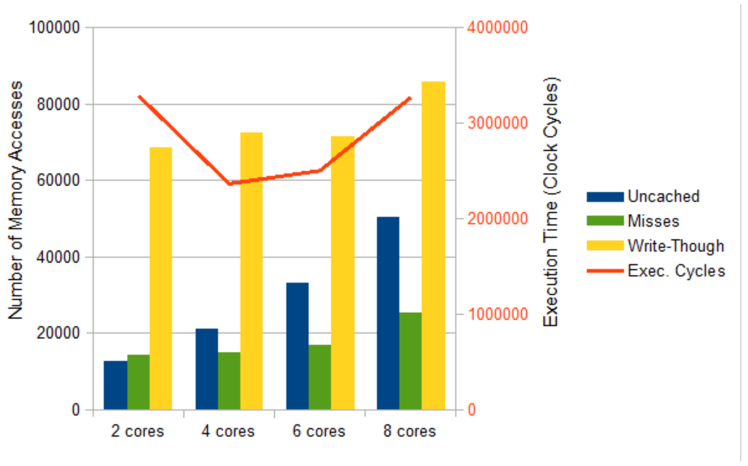

Fig. 8. Amount of different memory accesses for execution with ODC^2 in single cluster configuration.

on the number of participating cores as well as the used system configuration. Due to special characteristics of the computation, the application does not profit from a parallelisation with only two cores. Thus, with two cores no speedup can be realised compared to a *single core* execution, neither with the *uncached* platform nor the ODC^2 (see Figure 4 - 6).

In Figure 4, we present single cluster configuration. The results suggest 3DPP execution time reduction of 23,7% when using *ODC*^{2} (31,3% with *ODC*^{2} *AC*) with 4 cores. Similar results can be observed for the mesh configurations. In contrast, the *uncached* platform produce an execution time which is 42,9 % higher than the results for the single core execution (single cluster configuration) and even higher for the 2x2 mesh and 3x3 mesh (83,0% and 95,3% respectively).

The massive amount of execution cycles needed to execute the application with the *uncached* platform is caused by the huge amount of accesses to shared data which always results in uncached memory accesses, as seen in Figure 7. This type of memory access is comparatively rare when using ODC^2 since it solely represents accesses to synchronisation variables. It is dominated by the forced write-through accesses during the shared mode. When address checking is applied with ODC^2, these accesses can be significantly reduced (e.g. by 56% when using 4 cores in a single cluster configuration).

When increasing the number of cores involved in the parallel computation, the synchronisation overhead also increases. While the accesses associated to the computation do not scale with the number of cores, accesses to synchronisation data constantly grow (see *uncached* accesses in Figure 8). The synchronisation overhead may pollute the network and delay other memory accesses which reduce the speedup gained from the parallelisation. For the cluster configuration this effect goes together with the fact that all cores share the memory for accesses to code and private data. Therefore, the performance cannot be raised with more than 4 cores (*ODC*^{2}) or 6 cores (*ODC*^{2} *AC* and *uncached*) as displayed in Figure 4. For the ODC^2 similar outcome can be observed with the mesh configurations. The *uncached* platforms evaluated on 2x2 and 3x3 mesh configurations achieve continues speedup even with 8 cores, but still suffer from execution cycles way beyond the single core execution (e.g. 64,8% higher in a 3x3 mesh, see Figure 6).

6 Conclusion

The use of multicore systems for hard real-time applications is problematic when cache coherence is required. Common average performance oriented cache coherence protocols are of no use since they lack of timing predictability. We performed a case study with an avionics industrial application, in which we evaluated the timing predictable On-Demand Coherent Cache mechanism in a multicore environment. With ODC^2 significant speedup of about 25% with 4 cores could be achieved compared to a single core execution and even more to a platform with uncached accesses to shared data. In contrast to not considering the cache for shared data, the ODC^2 turned out to be a feasible opportunity to achieve performance speedup in a real-time multicore system.

In a next step, worst case execution time estimations has to be calculated to evaluate effect of the ODC^2 on a static timing analysis of the industrial application.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

References

1. A. Pyka, M. Rohde, and S. Uhrig, “A Real-time Capable First-level Cache for Multi-cores,” in *Workshop on High Performance and Real-time Embedded Systems (HiRES) in conjunction with HiPEAC’13*, Jan 2013.
2. S. Edwards and E. Lee, “The case for the precision timed (PRET) machine,” in *Design Automation Conference, 2007. DAC ’07. 44th ACM/IEEE*, 2007, pp. 264–265.
3. J. Reineke, D. Grund, C. Berg, and R. Wilhelm, “Timing predictability of cache replacement policies,” *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, Nov. 2007.
4. J.-F. Deverge and I. Puaut, “Wcet-directed dynamic scratchpad memory allocation of data,” in *Real-Time Systems, 2007. ECRTS ’07. 19th Euromicro Conference on*, 2007, pp. 179–190.
5. S. Wasly and R. Pellizzoni, “A dynamic scratchpad memory unit for predictable real-time embedded systems,” in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013, pp. 183–192.
6. B. C. Ward, H. J. L., C. J. Kenna, and J. H. Anderson, “Making shared caches more predictable on multicore platforms,” in *ECRTS ’13: Proceedings of the 25th Euromicro Conference on Real-Time Systems.* IEEE Computer Society, 2013, pp. 157–167.
7. X. Vera, B. Lisper, and J. Xue, “Data cache locking for higher program predictability,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 272–282, 2003.
8. A. Lebeck and D. Wood, “Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors,” in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, 1995, pp. 48–59.
9. T. Ashby, P. Diaz, and M. Cintra, “Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters,” *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 1175–1185, 2011.
10. J. Park, C. Jang, and J. Lee, “A software-managed coherent memory architecture for manycores,” in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213.
11. A. Ros and S. Kaxiras, “Complexity-effective multicore coherence,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 241–252.
12. H. Cheong, “Life span strategy – a compiler-based approach to cache coherence,” in *Proceedings of the 6th international conference on Supercomputing*, ser. ICS ’92. New York, NY, USA: ACM, 1992, pp. 139–148.
13. parMERASA - Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability, “http://www.parmerasa.eu/.”
14. “The SoCLib project. Mainpage.” last download Nov. 17, 2012. [Online]. Available: http://www.soclib.fr/