WCET Analysis of Parallel Benchmarks using On-Demand Coherent Cache

Arthur Pyka*, Lillian Tadros*, Sascha Uhrig*
Hugues Cassé**, Haluk Ozaktas**, Christine Rochange**
{arthur.pyka, lillian.tadros, sascha.uhrig}@tu-dortmund.de
{hugues.casse, haluk.ozaktas, christine.rochange}@irit.fr

*Technical University of Dortmund, Germany
**Université Paul Sabatier, Toulouse

Abstract. The rise of multi-core architectures has reached the embedded hard real-time domain, in which predictable timing behaviour is the key factor. Although cache memory and even cache coherence mechanisms are provided in most of these systems, when it comes to the execution of timing critical applications, caches are typically disregarded and accesses to shared data are performed uncached. There is a strong demand for time predictable cache coherence solutions in the hard-real time domain to overcome the performance loss entailed by the bypassing of cache memory.
In this paper, we illustrate how todays cache coherence protocols negatively affect the timing behaviour of a cache and thereby impede a reasonable static worst-case execution time (WCET) analysis. Using the OTAWA toolbox, we then perform static WCET estimations applying the On-Demand Coherent Cache (ODC^2). The elimination of unpredictable inter-cache communication with ODC^2 leads to a reasonable static cache analysis and a tight WCET estimation. We present WCET results of parallelised benchmarks using the ODC^2 on a multi-core platform and compare them with other time predictable approaches to access shared data.

1 Introduction

The trend towards multi-core architectures is increasingly visible in the hard real-time (HRT) domain. Concerning time analysability, the issues of a multi-core environment like task scheduling, network topology and of course memory hierarchy pose novel challenges. In hard real-time systems, the need for time predictability dominates performance considerations. In particular, the impact of cache memory is crucial on time predictability. Predicting the behaviour of caches (content and access latency) is a complex issue where the usage of caches generally increases the overestimation of a static WCET analysis. Therefore, a single level of cache memory is the maximum used, if any.

In case of multi-core systems where parallelised applications may share the same data, a cache coherence mechanism is required to guarantee a correct system behaviour. Research on cache coherence techniques has been going on for several decades. Generally, two classes of common hardware coherence protocols can be

distinguished, snooping-based and directory-based approaches [20]. All these approaches rely to some extent on the MESI paradigm [8], in which a cache line is extended by a state information that indicates if the corresponding data is locally *Modified*, *Exclusively* available in the local cache, unmodified *Shared* with other caches, or *Invalid*. The aim of these protocols is to achieve a high average-case execution time. Time predictability is not considered.

The way state-of-the-art cache coherence techniques are designed greatly impedes the predictability of cache content and access latencies (as described in Section 3). Computing a tight WCET estimation is nearly impossible. To meet the demands for high performance as well as real-time demands, todays multi/many-core architectures provide a bilateral approach to handle shared data accesses. Beside cache coherence techniques based on the MESI paradigm, uncached access to shared data can be performed. Some many-core architectures rely on a directory-based coherence protocol like Tilera's TILEPro/Gx or CaviumOpteron. Others implement snooping-based techniques as in LEON 3/4 and Freescale P4080. These systems regularly allow accesses to bypass the cache to fulfil the needs for time predictability. Multi-core architectures specified for hard real-time applications like Infineon's Aurix or the Kalray MPPA completely abstain from a hardware cache coherence solution. Accordingly, in hard real-time domains like avionics and automotive, the use of caches is not an option for parallelised applications. The potential performance gain that comes along with caches is wasted because of the missing time predictability.

The real-time capable On-Demand Coherent cache (ODC^2) presented by Pyka et al. [14] offers time analysable coherence operations and an average-case performance similar to standard MESI based protocols. In this paper, we apply the OTAWA toolbox to perform static WCET estimations of several parallel benchmarks using ODC^2 to demonstrate that the mechanism allows the estimation of tight WCET bounds on real-time multi-core systems.

The rest of the paper is organised as follows: Section 2 discusses related work, followed by a timing analysis of common cache coherence techniques in Section 3. After a short summary of the ODC^2 mechanism in Section 4 the static analysis of the ODC^2 is illustrated in Section 5. Section 6 presents the evaluation followed by the results in Section 7. This work is concluded with Section 8.

2 Related Work

Multicore architectures and parallelised execution are challenging to static timing analysis. Depending on the hardware and software under analysis, overestimated execution time bounds have to be tolerated, if the execution time can be bound at all [7][17]. Research on the time predictability of caches with the aim of supporting WCET estimation mainly focuses on instruction caches and single core data caches. Applications share parts of the instruction cache, with the result that unforeseen evictions corrupt the prediction of cache hits. To decrease the cache miss rate and consequently the worst-case execution time, several techniques were proposed based on the partitioning and locking of instruction caches [1][11].

For data caches, research on improving the WCET estimation has been conducted for different cache configurations [19][15], cache replacement policies [16] or the stack distance [10]. Furthermore, WCET-driven code allocation techniques can support the predictability of cache accesses [5].

Since time predictability is gaining importance in future embedded systems, the development of comprehensively time analysable architectures [18][9] is pushed forward. Scratchpad memories are often used in such architectures and a lot of research has been done in this area. A key aspect here is the WCET-aware allocation of code [4][12] as well as data [21][3] for scratchpads to allow reliable prediction of access latencies. Local scratchpads clearly do not permit coherent access to shared data.

3 Time analysability of cache coherence techniques

A tight worst-case execution time estimation demands a predictable behaviour of all relevant system components [23]. In case of the cache memory, the prediction of cache hits as well as bounded cache access latencies are crucial for a reasonable access time estimation. A static cache analysis benefits from an extensive knowledge of the cache content, which allows some of the cache accesses to be treated as cache hits with a low access latency. If a cache miss has to be assumed, the access latency increases by the time needed to transfer the data from the main memory to the cache. Applying a cache coherence protocol introduces additional delays for coherence transactions.

Common cache coherence techniques are either based on snooping-based protocols following the MESI paradigm or directory-based protocols, combining MESI with the *Globally Owned - Locally Shared* principle. All applied techniques, regardless of specific implementation details, rely on coherence transactions between the caches. This inter-cache communication negatively affects the time analysability of cache accesses in multiple ways and makes a tight WCET estimation generally impossible. Prediction of a cache content in a static cache analysis is based on the assumption that solely internal accesses can change a cache's state and content. The mechanism of common cache coherence protocols violates that assumption. In the following, selected unpredictable interferences are illustrated:

1. **External Invalidations:**
   Coherence protocols which belong to the class of write-invalidate protocols rely on external invalidation of shared cache lines to avoid outdated copies of shared data. The possibility of invalidating a cache line via a coherence message, as given in many architectures like Tilera's TilePro/TileGx, Cavium Opteron, LEON 3/4 and Freescale P4080, harms the predictability of the content of a cache line. A static cache analysis performs a must-/may-analysis to estimate if a cache line is located in a cache [6]. A cache access can be regarded as a cache hit only if the analysis guarantees the existence of the given cache line in the cache. From the perspective of a static cache analysis, an external invalidation can not be predicted, neither the arrival time, nor the specific cache block that becomes invalid is known. In a worst-case analysis, this results in two serious consequences:

Assuming that the analysis cannot predict possible invalidations of the programs running in parallel, it has to assume that any of the existing cache lines may become invalid. Hence, subsequent cache accesses cannot be predicted as a cache hit. Second, since the arrival time is not known, an external invalidation must be assumed at any possible point in time. This forces the cache analysis to expect a cache miss as the worst-case on every single cache access to shared data. The result is a strongly increased worst-case execution time estimation that makes caching of shared data useless.

2. **Cache misses with write-back caches:**
   In architectures using the write-back policy for write accesses to cached data, the requested data can be outdated in the main memory with a modified copy of the data present in another core's cache. In that case, when a cache access to that data generates a cache miss, the core holding the updated copy has to interrupt the loading of the cache line and write back the modified data to the shared memory first. For the latter core, this intervention causes a serious increase in access latency. In a worst-case analysis the maximum possible value of this increased latency will be assumed for any cache miss and hence, strongly increase the complete WCET.
3. **Write accesses in directory-based write-invalidate protocols:**
   In caches applying a write-back strategy combined with a directory-based write-invalidate protocol, a write access to a shared cache line generates an invalidation message. To complete this operation (switching the written cache line to an exclusive state), the accessing cache has to wait for all cores to acknowledge the invalidation. Otherwise it could happen that two caches switch the same cache line into an exclusive state, which is a forbidden situation in a write-invalidate protocol. Thus, instead of assuming a minimal latency for a cache write hit to shared data, a latency similar to a cache miss must be assumed.

The mentioned examples of interferences and interdependencies of cache accesses that come along with common cache coherence protocols hinder the prediction of cache access latencies required for hard real time systems. For this reason, current multi-core systems typically allow accesses to bypass the cache, so that hard real-time applications can be executed. Alternatively, they delegate the handling of coherent accesses to shared data to the user or to the system software. These approaches allow time predictability but induce a serious overhead to the execution time. An efficient cache coherence technique suitable for hard real-time systems has to be free of unpredictable interferences between caches and has to permit caching of shared data as efficiently as possible.

4 Summary of the ODC^2 mechanism

The On-Demand Coherent Cache specifies a cache coherence protocol based on an extended cache module and a minimal software code extension. Hence it implements a hardware/software co-approach. The mechanism completely abstains from coherence interactions between the cores and external modifications to a

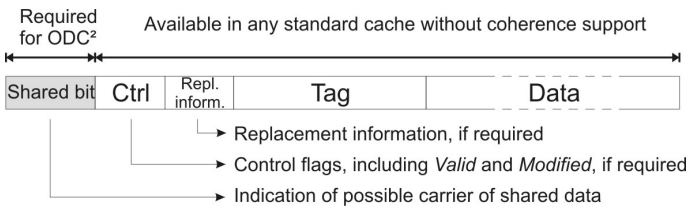

Fig. 1. Block frame with ODC^2 status bits.

cache. Operations to maintain the cache coherence are triggered by special control instructions which are attached to synchronisation points in the program.

These two control instructions encapsulate a specific code region in which shared data is accessed, e.g. a critical section. The commands are used to activate and deactivate a special handling of cache accesses in the ODC^2. Doing so, accesses to shared data are kept coherent inside the code region. The mechanism is not restricted to a special synchronisation technique. Common techniques like mutexes, semaphores, barriers etc. can be applied. In general, the synchronisation must guarantee that multiple cores do not access/modify the same shared variable at the same time; simultaneous read-only accesses are allowed.

By default, the ODC^2 operates in *private mode*, acting as a normal incoherent cache without any additional functionality. Thus, in *private mode*, only accesses to private data are allowed. When the control instruction switches the cache into *shared mode*, all newly loaded cache lines will be marked as *shared*. Using an additional shared bit information inside the block frame (see Figure 1), the content is treated as possibly shared. To prevent private data from being marked as well, the ODC^2 can check the target address of cache accesses and restrict marking to a specific memory section that contains shared data only. Write accesses to shared data while in *shared mode* are performed in write-through policy to prevent false sharing (unintentional loading of foreign data together with requested data). When the critical section is left the control instruction switches the cache back to *private mode* and the *restore procedure* starts. All cache lines marked with the shared bit become invalid. At the end of the procedure, no shared data remains in the cache and the shared data in the main memory is up to date.

With the ODC^2 mechanism, cache coherence is only maintained when it is needed, when shared data is accessed, and accesses to private data are performed uninterrupted without coherence overhead. Due to the frequent invalidation of all cache lines containing shared data, an increase of the cache miss rate must be assumed. But the absence of inter-cache communication still allows a tight static WCET analysis and makes the technique suitable for hard-real time systems.

5 Time analysability of the ODC^2

Although the On-Demand Coherent Cache does not rely on unpredictable interferences between the caches, the mechanism applies coherence operations that affect the timing of the cache and, consequently, the estimated worst-case latencies. This

section analyses the influence of the ODC^2 mechanism on time predictability in detail. As described in Section 4, the ODC^2 maintains coherent accesses only when shared data is accessed. When the ODC^2 remains in private mode no special handling occurs. In private mode, it acts as a standard incoherent cache with the same level of time predictability provided by its architectural characteristics, such as associativity or replacement strategy. This can be seen as an upper bound for the possible analysability. In other words, the level of predictability of the timing behaviour the ODC^2 is limited by the underlying cache architecture. Therefore, it is advisable to use a cache that encourages static analysability, for example using LRU as the replacement strategy [16].

Based on this, although the coherence operations performed in shared mode may affect the timing behaviour in certain ways, they do not constrain its predictability. The different influences are illustrated in detail below:

- **ODC^2 control instructions**
  The commands to enter and exit the shared mode can be implemented as standard accesses to cache control registers. Regardless of the functionality that these instructions perform, the execution time of these instructions is static and does not affect predictability.
- **Marking of shared cache lines**
  Once the shared mode is entered, newly loaded cache lines will be marked with the shared bit. This is done by modifying a status bit in the block frame of the cache line (see Figure 1) and does not increase the latency of the access.
- **Restore procedure**
  The only operation that has an influence on the timing behaviour is the restore procedure, triggered on leaving the shared mode. As a first step, cache lines which are marked with the shared bit become invalid. This operation can be implemented as a global clearing of the valid bit hooked on the shared bit, and can be done in parallel for all shared cache lines. Doing so, the latency of the operation is independent of the number of marked cache lines. The additional required time is fixed and thus does not harm predictability. Of course, the invalidation of cache lines indirectly increases the execution time, since it results in future cache misses. These will, however, be accounted in the remaining cache analysis.

6 WCET estimation

We evaluate different parallel benchmarks using the ODC^2 to estimate the influences on the worst-case timing behaviour, as analysed in the prior section. Since extensive performance evaluation has been performed before [14][13], we focus on estimations of the worst-case execution time. During the evaluation, upper bounds on the WCET, as executed on a multi/many-core platform applying the ODC^2, are estimated. Furthermore, we examine the number of different access types to the cache and shared memory.

Three parallel benchmarks, representing different shared data access patterns are employed in this evaluation: in **Matrix Multiplication (Matrix)**, two matrices of 40x76 cells are multiplied in parallel by the available cores. Each core

computes several complete rows of the target matrix and the synchronisation is done via mutexes. The **Fast Fourier Transform (FFT)** is applied to an array of 512 complex numbers. The calculation of the Fourier algorithm of the array is done in parallel on disjunct parts of the array, synchronized via mutexes and barriers. **Dijkstra (Dijkstra)** computes the shortest path between two nodes in a set of 100 nodes using the Dijkstra algorithm. Barriers are used to organize parallel accesses to the graph and other shared data structures.

6.1 Evaluation platform

The WCET estimations are calculated using the OTAWA (Open Tool for Adaptive WCET Analyses) [2] toolbox developed by the Institut de Recherche en Informatique (IRIT) in Toulouse. In the context of the FP7 parMERASA project [22], a model of the real-time capable parMERASA many-core architecture has been integrated into OTAWA. The architecture is composed of PowerPC-based cores and a shared on-chip memory, linked through a tree-type on-chip interconnect. The shared memory contains private and shared application data. Moreover, each processor core uses a local data cache and a perfect instruction scratchpad, allowing one cycle delays for all instruction fetches. Hence, we eliminate the memory interference caused by instruction cache misses. The size of the local data caches is 16 kB, organised in 512 2-way associative sets.

Four different platforms are examined in this evaluation, implementing different methods to access shared data in a coherent way. The predictable ODC^2 mechanism is implemented inside the *ODC^2* platform. With the *uncached* platform, all accesses to shared data bypass the cache and result in an access to the shared memory. The data cache is used for private data only. This is a common state-of-the-art approach in many HRT systems. The *cache flush* platform implements a software-based coherence approach. While shared as well as private data is cached with this platform, the complete cache is invalidated at synchronisation points (critical sections, barriers). Note that the synchronisation variables shall not be kept in the local caches. To avoid incoherent memory situations caused by false sharing, the cache must operate in write-through mode. Lastly, the theoretical Magic platform represents a reference for the maximum possible performance where coherent accesses are performed at no overhead.

6.2 OTAWA ODC^2 model

A model of the ODC^2 is included in the OTAWA toolbox. It models the behaviour of the cache including the specific coherence operation, as described in Section 4. Applying the ODC^2 in a static analysis requires special handling.

Data cache support in WCET computation relies on several static analyses that attempt to model as precisely as possible the behaviour of the cache. As usual, static analyses cannot cope with all possible program behaviours and, to maintain safety, may fall back to overestimation. To handle this variability, the involved analyses are doubled with a MUST analysis and a MAY analysis that determine a property, respectively, for all execution paths or only for some paths.

This allows deriving upper and lower bounds on the property in question, in this case, the number of cache misses.

The following analyses are applied for a classic data cache:

1. *Reference Analysis* — defines the pattern of accessed cache blocks for each memory access instruction in the program,
2. *Data Cache State Analysis* — determines for each program point an abstraction of the cache state,
3. *Miss Count Analysis* — counts the number of misses from the abstract states of the cache for each memory access instruction,
4. *Dirty Analysis* — only used for write-back caches, it determines if a block has been modified and therefore requires a write-back operation before its replacement.

The two latter analyses compute the time spent in a memory access: the number of times write-backs are performed (when a block has been modified), the number of times a memory access is performed (misses).

Adaptation to ODC^2 caches requires modifing analyses 2 to 4: the *Reference Analysis* only depends on semantics of the program and not on the structure of the hardware. The main changes come from the dual behaviour of the ODC^2 cache: inside a shared area, the cache handles memory stores in write-through mode; in a non-shared (private) area, depending on the base policy of the cache, a write-through or write-back policy is applied. Hence, an analysis of cache mode for each memory access instruction is needed. This so-called *Shared Analysis* is relatively easy to perform: a shared area starts with a specific *enter* instruction and ends with another dedicated *exit* instruction. Although *enter* and *exit* should be paired to discriminate shared and private areas, it is easy to imagine Control Flow Graphs (CFG) containing areas that are sometimes private and sometimes shared.

For the sake of completeness, the *Shared Analysis* is performed in both MUST and MAY contexts and, consequently, an instruction may be classified as Always Shared (AS), Never Shared (NS) or Sometimes Shared (SS). These categories are then used in analyses 2 to 4. It is clear that categories AS and NS are precise and do not alter the efficiency of the data cache analysis. In contrast, SS causes an approximation because each subsequent analysis needs to revert to the worst-case of shared and private modes. As such cases should be rare and may even denote suspicious code, the *Shared Analysis* required by ODC^2 should not induce overestimation.

Once the shared areas have been identified, it is straightforward to adapt analyses 2 to 4. *Data Cache State Analysis* has only to compose existing write-through and private policy cache analyses: they work on the same abstract state of the cache, the load operation is the same between two modes and only the store operation may choose one of the two possible write behaviours of the cache. The only trick concerns the flushing of shared blocks when an *exit* instruction is reached: in fact, OTAWA already supports this operation to implement cache flush instructions found in some instruction sets. Only the *Miss Count Analysis* and *Dirty Analysis*, handle stores in shared areas either as always causing a miss or ignoring them, respectively.

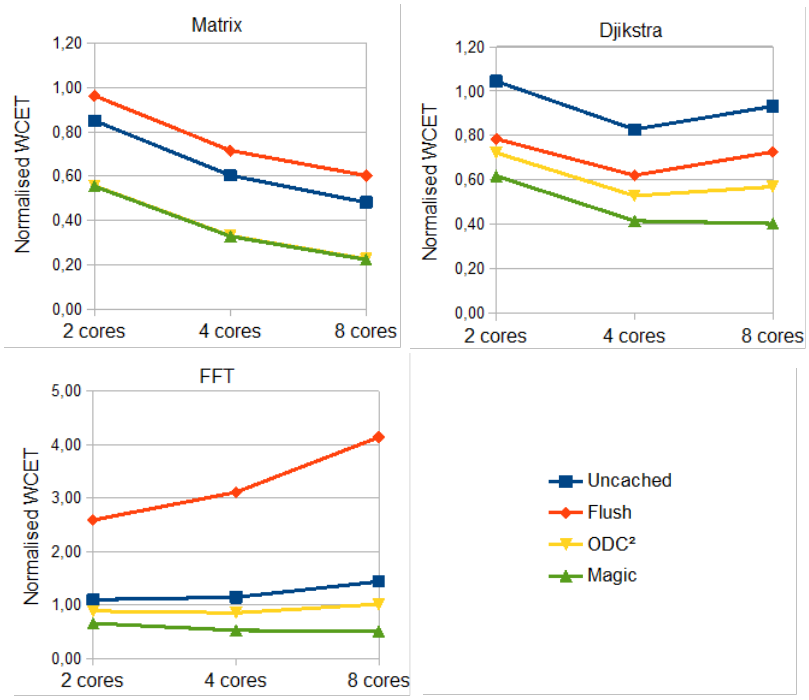

Fig. 2. Estimated WCET of benchmarks executed with 2 - 8 cores, normalised to a single core execution.

A second important feature of ODC^2 is the protected memory region that is never shared. As soon as the *Reference Analysis* provides precise results, it is quite simple to know whether the accessed data is in the protected region. In the former case, the shared property is ignored and the private write policy of the cache applies. From the point of view of static analyses, ODC^2 caches cause very little overestimation. In the rare case where shared analysis causes approximation, it is usually due to a fault in the program. Any remaining imprecision is mainly due to the data cache analysis itself and not from the ODC^2.

7 Evaluation results

The evaluation results confirm the theoretical analysis of the ODC^2 mechanism. Compared to the Uncached platform, as well as the Cache Flush platform, a significantly reduced WCET estimation can be observed with ODC^2, as illustrated in Figure 2. A decrease of the execution time estimation using ODC^2 compared to Uncached varies from 16,5% to 53,0%, depending on the benchmark and number of participating cores. An even larger reduction of up to 75,0% (FFT with 8 cores) is observed compared to Cache Flush. The Magic platform predictably achieves the best performance in all cases. Compared to the Magic platform, the overhead of coherence operations with ODC^2 results in minor WCET increase of 0,3% -

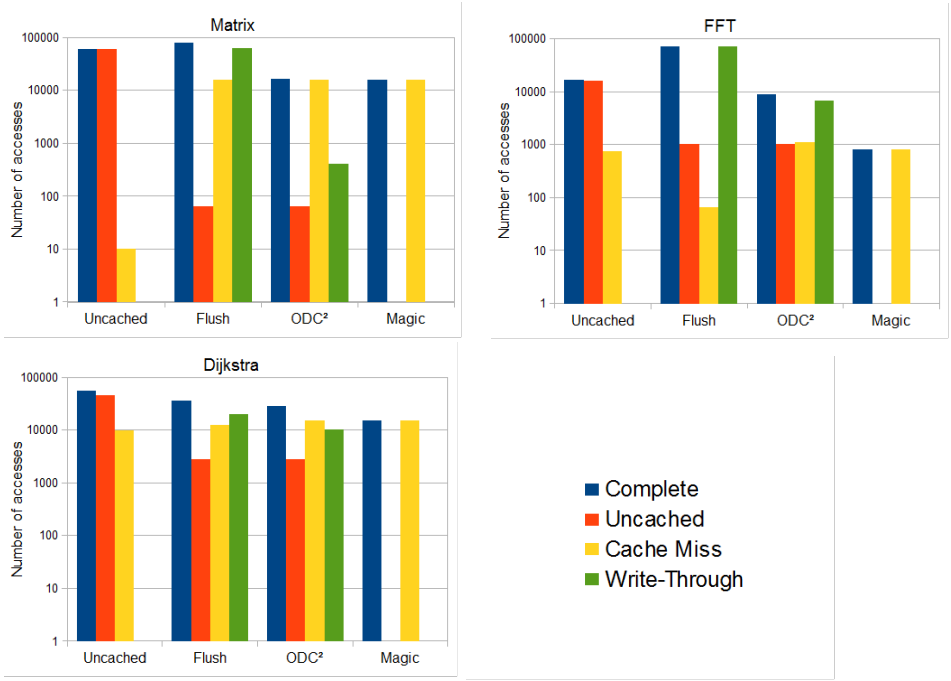

Fig. 3. Estimated worst-case access numbers on execution with 4 cores.

1,4% for the Matrix benchmark. On the other hand, an increase of 35,0% - 99,8% is stated for the FFT benchmark.

The evident discrepancy between the three benchmarks is caused by their distinct memory access patterns. This distinction reveals the advantages and disadvantages of the different approaches to access shared memory. The Matrix benchmark is an example of an algorithm that can be parallelised in a highly efficient manner. Accesses to shared data are dominated by read accesses, while write accesses are relatively rare. Thus, the ODC^2 can perform a large part of shared data accesses locally in the cache. In Figure 3 different access type numbers are presented for the parallel execution of Matrix (and other benchmarks) using 4 cores. To be able to illustrate high and low numbers at once, a logarithmic scale is used. The number of *Write-Through* accesses with ODC^2 is notably low compared to the execution with Cache Flush, in which all write accesses result in Write-Through accesses. For all cache-based platforms (ODC^2, Cache Flush and Magic), the amount of *Cache Misses* is similar. Here, expected variation is covered by a pessimism in the estimation of cache hits, caused by complex array addressing.

In contrast to Matrix, the FFT benchmark is characterised by frequent write accesses to private and shared data. None of the platforms scale adequately with an increased core number because the FFT implementation allows parallelisation in a less efficient manner. While the Cache Flush suffers from the large number of Write-Through accesses again, it benefits from its non-write-allocate strategy

which significantly decreases the cache miss rate (see Figure 3). ODC^2 mechanism induces the highest number of cache misses, which shows the overhead of the invalidation. Nevertheless, the reduced amount of accesses to the shared memory (*Complete*) is the key ingredient to a low WCET. Executing Dijkstra benchmark with the Cache Flush platform, the additional overhead in memory accesses compared to ODC^2 is less significant but still sufficient to create an increase in the WCET estimation of 8,9% - 21,6%.

The results show that the ODC^2 mechanism achieves a significant performance gain compared to the other platform . Even compared to a theoretical "'best-case"' solution (Magic platform), the ODC^2 entails little overhead and thus allows a tight WCET estimation.

8 Conclusion

Cache coherence mechanisms implemented in today's multicore architectures include operations to achieve coherent accesses to shared data. Relying on frequent interaction between the caches, these operations impede a feasible worst-case execution time estimation. Common time predictable approaches induce significant overhead by increasing the rate of memory accesses. Our study demonstrates that compared to common approaches, a feasible performance gain can be achieved when using the On-Demand Coherent Cache for parallelised applications.

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. Asaduzzaman, N. Limbachiya, I. Mahgoub, and F. Sibai, "Evaluation of i-cache locking technique for real-time embedded systems," in *Innovations in Information Technology, 2007. IIT '07. 4th International Conference on*, nov. 2007, pp. 342 –346.
2. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive WCET analysis," in *Software Technologies for Embedded and Ubiquitous Systems*. Springer Berlin Heidelberg, 2011, pp. 35–46.
3. 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*, July 2007, pp. 179–190.
4. H. Falk and J. C. Kleinsorge, "Optimal static WCET-aware scratchpad allocation of program code," in *The 46th Design Automation Conference (DAC)*, San Francisco / USA, jul 2009, pp. 732–737.
5. H. Falk and H. Kotthaus, "WCET-driven cache-aware code positioning," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Taipei, Taiwan, oct 2011, pp. 145–154.
6. C. Ferdinand, D. Kastner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm, "Run-time guarantees for real-time systems – the USES approach," in *GI Jahrestagung*, 1999, pp. 410–419.
7. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," in *IEEE*, vol. 91, no. 7, 2003, pp. 1038–1054.

8. J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.
9. I. Liu, J. Reineke, and E. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, nov. 2010, pp. 2111 –2115.
10. Y. Liu and W. Zhang, "Exploiting stack distance to estimate worst-case data cache performance," in *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, S. Y. Shin and S. Ossowski, Eds. ACM, 2009, pp. 1979–1983.
11. S. Plazar, J. Kleinsorge, H. Falk, and P. Marwedel, "WCET-aware static locking of instruction caches," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, apr 2012, pp. 44–52.
12. A. Prakash and H. Patel, "An instruction scratchpad memory allocation for the precision timed architecture," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 11, pp. 1819–1823, Nov 2013.
13. A. Pyka, M. Rohde, and S. Uhrig, "Performance evaluation of the time analysable on-demand coherent cache," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, July 2013, pp. 1887–1892.
14. ——, "A real-time capable coherent data cache for multicores," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 6, pp. 1342–1354, 2014.
15. H. Ramaprasad and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, ser. RTAS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 148–157.
16. 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.
17. C. Rochange, "An Overview of Approaches Towards the Timing Analysability of Parallel Architecture," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, ser. OpenAccess Series in Informatics (OASIcs), vol. 18, Dagstuhl, Germany, 2011, pp. 32–41.
18. M. Schoeberl, "Time-predictable chip-multiprocessor design," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, Nov 2010, pp. 2116–2120.
19. R. Sen and Y. N. Srikant, "Wcet estimation for executables in the presence of data caches," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*. New York, NY, USA: ACM, 2007, pp. 203–212.
20. P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, Jun. 1990.
21. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, Dec 2005, pp. 10 pp.–232.
22. T. Ungerer *et al.*, "Experiences and results of parallelisation of industrial hard real-time applications for the parmerasa multi-core," in *Submitted to the 3nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*, jan 2015.
23. R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.