

# Response-time analysis for fixed-priority systems with a write-back cache

Robert I. Davis<sup>1,2</sup>  · Sebastian Altmeyer<sup>3</sup> · Jan Reineke<sup>4</sup>

Published online: 11 April 2018  
© The Author(s) 2018

**Abstract** This paper introduces analyses of write-back caches integrated into response-time analysis for fixed-priority preemptive and non-preemptive scheduling. For each scheduling paradigm, we derive four different approaches to computing the additional costs incurred due to write backs. We show the dominance relationships between these different approaches and note how they can be combined to form a single state-of-the-art approach in each case. The evaluation explores the relative performance of the different methods using a set of benchmarks, as well as making comparisons with no cache and a write-through cache. We also explore the effect of write buffers used to hide the latency of write-through caches. We show that depending upon the depth of the buffer used and the policies employed, such buffers can result in domino effects. Our evaluation shows that even ignoring domino effects, a substantial write buffer is needed to match the guaranteed performance of write-back caches.

**Keywords** Write-back cache · Cache-related preemption delays (CRPD) · Schedulability analysis · Fixed priority scheduling · Real-time

---

✉ Robert I. Davis  
rob.davis@york.ac.uk

Sebastian Altmeyer  
altmeyer@uva.nl

Jan Reineke  
reineke@cs.uni-saarland.de

<sup>1</sup> University of York, York, UK

<sup>2</sup> INRIA, Paris, France

<sup>3</sup> University of Amsterdam, Amsterdam, Netherlands

<sup>4</sup> Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

## Extended version

This paper builds upon and extends the RTNS 2016 paper *Analysis of Write-Back Caches under Fixed-Priority Preemptive and Non-preemptive Scheduling* (Davis et al. 2016) as follows:

- Worked examples have been added, in Sects. 4.3 and 5.4, illustrating the incomparability and dominance relationships between the different analysis methods.
- A brief discussion of the *sustainability* of the analysis is given in Sect. 6.
- Additional experiments have been added, in Sect. 7.1, exploring how the performance of the various analysis methods is impacted by changes in the number of tasks and by the memory delay.
- A discussion and evaluation of the impact of write buffers on the performance of write-through caches has been added in Sect. 8. Here we show that depending on the precise policies employed, write buffers may result in domino effects, severely affecting guaranteed performance.
- Finally, while data-cache analysis is not the main focus of the paper, we review related work in this area in the Appendix.

## 1 Introduction

During the last two decades, applications in aerospace and automotive electronics have progressed from deploying embedded microprocessors clocked in the 10's of MHz range to higher performance devices operating in the 100's of MHz to GHz range. The use of high-performance embedded microprocessors has meant that access times to main memory have become a significant bottleneck, necessitating the use of caches to tackle the increasing gap between processor and memory speeds.

Caches may be classified according to the type of information that they store, thus we have *data caches*, *instruction caches*, and *unified caches* which store both instructions and data. In this paper, we are interested in the behaviour of single-level data and unified caches. The behaviour of these caches is crucially dependent on the *write policy* used. Two policies are commonly employed: *write back* and *write through*. In caches using a write-through policy, writes immediately go to memory, thus multiple writes to the same location incur an unnecessarily high overhead. In caches using the write-back policy, writes are not immediately written back to memory. Instead, writes are performed in the cache and the affected cache lines are marked as *dirty*. Only upon eviction of a dirty cache line are its contents written back to main memory. This has the potential to greatly reduce the overall number of writes to main memory compared to a write-through policy, as multiple writes to the same location and multiple writes to different locations in the same cache line can be consolidated.

Evictions of dirty cache lines are a source of interference between different tasks sharing a cache. The execution of a task may leave dirty cache lines in the cache that will have to be written back during the execution of another task, delaying that task's execution. A read which is a cache miss and evicts a dirty cache line may incur approximately twice the delay compared to evicting a non-dirty line, since the former requires both a read from memory and an additional write back of the dirty line. This

may occur with non-preemptive as well as with preemptive scheduling, and dirty cache lines left by low priority tasks may impact the response time of higher priority tasks and vice-versa. This is in contrast to the impact of evictions with a write-through cache, which only affect other tasks under preemptive scheduling, and then only tasks of lower priority. In this paper, we discuss different ways of soundly accounting for write backs, and show how to integrate these costs into response-time analysis for both fixed-priority preemptive and non-preemptive scheduling. We also consider the use of *write buffers* as a way of hiding the write latency inherent in a write-through cache. We show that the use of such buffers can potentially lead to *domino effects*.

## 1.1 Related work

### 1.1.1 Accounting for overheads in schedulability analysis

Early work on accounting for scheduling overheads in fixed-priority preemptive systems by Katcher et al. (1993) and Burns (1994) focused on scheduler overheads and context switch costs. Subsequent work on the analysis of Cache Related Preemption Delays (CRPD) and their integration into schedulability analyses used the concepts of Useful Cache Blocks (UCBs) and Evicting Cache Blocks (ECBs), see Sect. 2.1 of (Altmeyer and Maiza 2011) for a detailed description. A number of methods have been developed for computing CRPD under fixed-priority preemptive scheduling. Busquets-Mataix et al. (1996) introduced the ECB-Only approach, which considers just the preempting task; while Lee (1998) developed the UCB-Only approach, which considers just the preempted task (s). Both the UCB-Union approach (Tan and Mooney 2007), and the ECB-Union approach (Altmeyer et al. 2011) consider both the preempted and preempting tasks. As does an alternative approach developed by Staschulat et al. (2005). These approaches were later superseded by multiset based methods (ECB-Union Multiset and UCB-Union Multiset) which dominate them (Altmeyer et al. 2012). These methods have been adapted by Lunniss et al. (2013, 2014a) to EDF scheduling and to hierarchical scheduling with local fixed-priority (Lunniss et al. 2014b) and EDF (Lunniss et al. 2016) schedulers. They have also been integrated into a response time analysis framework for multicore systems (Altmeyer et al. 2015).

Cache partitioning is one way of eliminating CRPD; however, this results in inflated worst-case execution times due to the reduced cache partition size available to each task. Altmeyer et al. (2014, 2016) derived an optimal cache partitioning algorithm for the case where each task has its own partition. They compared cache partitioning and cache sharing accounting for CRPD, concluding that the trade off between longer worst-case execution times and CRPD often favours sharing the cache rather than partitioning it.

Preemption thresholds (Wang and Saksena 1999; Saksena and Wang 2000) provide an alternative means of reducing CRPD by making certain groups of tasks non-preemptable with respect to each other. Bril et al. (2014) integrated CRPD into analysis for fixed-priority scheduling with preemption thresholds. Further work in this area by Wang et al. (2015) showed that by using preemption thresholds, groups of tasks

can share a partition while still avoiding CRPD. This results in a hybrid approach that can outperform the approach of Altmeyer et al. (2014).

As far as we are aware, all of the prior work on integrating CRPD into schedulability analysis assumes write-through caches. In this paper, we explore the impact of using write-back caches instead.

With write-through caches, non-preemptive scheduling provides a simple means of eliminating CRPD without increasing worst-case execution times, since each task can still utilise the entire cache. However, with write-back caches, non-preemptive scheduling is insufficient to eliminate all cache-related interference effects. In this paper, we therefore consider the effects of write-back caches under both fixed-priority preemptive scheduling and fixed-priority non-preemptive scheduling. As this is the *first* such study of the impact of write backs, we restrict our attention to direct-mapped caches (examples of microprocessors that implement such caches are given in Sect. 2). In future, we aim to extend the techniques to set-associative caches and replacement policies such as LRU using the methodology given by Altmeyer et al. (2011).

### 1.1.2 Write-back caches in worst-case execution time (WCET) analysis

Ferdinand and Wilhelm (1999) introduced an analysis of write-back caches to determine for each memory access, which cache lines may have to be written back. The basic idea is to track for each potentially dirty memory block whether it must or may be cached; however, this analysis has neither been integrated into a WCET analysis nor has it been experimentally evaluated. Sondag and Rajan (2010) implement a similar idea in the context of multi-level cache analysis, where the write-back behaviour of the first-level cache influences the contents of the second-level cache. While potential write backs from the first- to the second-level cache are correctly accounted for, the *cost* of write backs to main memory does not seem to be taken into account within their WCET analysis. We note that both approaches (Ferdinand and Wilhelm 1999; Sondag and Rajan 2010) are not particularly suited to precisely bound the *number* of write backs, as imprecisions in the may- and must-analyses yield many potential write backs for a single write back in a concrete execution. To analyze a program's WCET, Li et al. (1996) proposed to capture both the software and the microarchitectural behaviour via integer linear programming (ILP). Their analysis is able to cover write-back caches, however, scalability is a major concern. The key distinction between the work presented in this paper and previous research is that our work focuses on the open problem of integrating write-back costs into schedulability analysis. Data cache analysis is not per-se the focus of the work in this paper, nevertheless we provide a discussion of related work in that area in the appendix. For readers interested in cache analysis techniques, a recent survey is given by Lv et al. (2016).

## 1.2 Organisation

The remainder of the paper is organized as follows. Section 2 discusses caches, different write policies and a classification of write backs, as well as how a task's write-back behaviour can be characterized. Section 3 sets out the task model used and recaps on

existing response-time analysis techniques. Sections 4 and 5 derive analyses bounding the cost of using write-back caches under fixed-priority non-preemptive and fixed-priority preemptive scheduling respectively. Section 6 discusses the *sustainability* of the analysis presented in those sections. Section 7 provides an evaluation of the performance of the different analyses for write-back caches, as compared to no cache and a write-through cache. Section 8 discusses the use of write buffers to improve the performance of write-through caches, and evaluates the effectiveness of different sized buffers. Section 9 discusses how information characterising write-back cache behaviour can be obtained. Finally, Sect. 10 concludes with a summary and a discussion of how the work in this paper may be extended. The appendix provides a brief review of related work on data cache analysis.

## 2 Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the difference in speed between the processor and main memory. To reduce management overhead and to profit from spatial locality, data is not cached at the granularity of words, but at the granularity of so-called *memory blocks*. To this end, main memory is logically partitioned into equally-sized memory blocks. Blocks are cached in *cache lines* of the same size. The size of a memory block varies from one processor to another, but is usually between 32 and 128 bytes.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache, a *cache hit*, or not, a *cache miss*. To enable an efficient look-up, each memory block can only be stored in a small number of cache lines referred to as a *cache set*. Thus caches are partitioned into a number of equally-sized cache sets. The size of a cache set is called the *associativity* of the cache.

The *placement policy* determines the cache set a memory block maps to. Typically, the number of cache sets is a power of two, and *modulo placement* is employed, where the least significant bits of the block number determine the cache set that a memory block maps to. Since caches are usually much smaller than main memory, a *replacement policy* is used to decide which memory block to replace on a cache miss. As stated earlier, we limit our attention to *direct-mapped* caches, where each cache set consists of exactly one cache line. In this case, the only possible action on a cache miss is to replace the memory block currently stored in the cache line that the accessed memory block maps to.

In this paper, we assume a timing-compositional architecture (Hahn et al. 2013), i.e. the timing contribution of cache misses and write backs can be analyzed separately from other architectural features such as the pipeline behaviour.

### 2.1 Write policies

Data written to the cache needs to eventually also be written to main memory. When exactly the data is written to main memory is determined by the *write policy*. There are two basic write policies: With a *write-through* policy, the write to main memory is requested at the same time as the write to the cache. With a *write-back* policy, the write

to main memory is postponed until the memory block containing the data is evicted from the cache, it is then written back to main memory in its entirety.

Write through is simpler to implement than write back, but may result in a significantly larger number of accesses to main memory. If a cached memory block is written to multiple times before being evicted, under write back only the final write needs to be performed to main memory. The drawback of write-back caches is that additional *dirty* bits are required to keep track of which cache lines have been modified since they were fetched from main memory, the writes are delayed, and the logic required to implement the cache is more complex.

Due to the potential performance advantages of write-back caches they are often preferred in embedded microprocessor designs. Alternatively, caches may be configurable as write back or write through. Examples include: Infineon Tricore TC1M (separate data and instruction caches, LRU replacement policy, write back); Freescale MPC740 (separate data and instruction caches, PLRU replacement policy, configurable for write back or write through); Renesas SH7705 (unified data and instruction cache, LRU replacement policy, configurable for write back or write through); Renesas SH7750 (separate instruction and data caches, direct mapped, configurable for write back or write through); NEC VR4181 and VR4121 (separate instruction and data caches, direct mapped, write back).

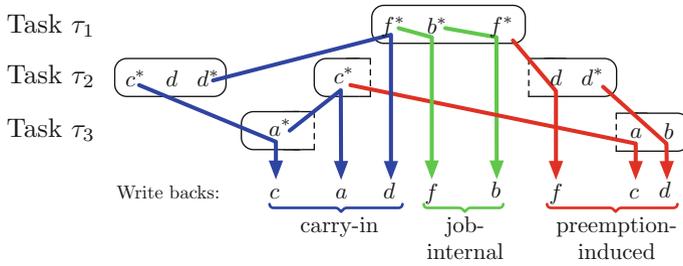
A second question to answer when designing a cache is what happens on a write to a memory block that is not cached. There are two *write-miss policies*: With *write allocate* a cache line is allocated to the memory block containing the word that is being written, which is fetched from main memory, then the write is performed in the cache. With *no-write allocate* the write is performed only in main memory, and no cache line is allocated. In principle, each write policy can be used in conjunction with each write-miss policy; however, usually, write through is combined with no-write allocate, and write back is combined with write allocate. In this paper we assume a cache employing *write back* and *write allocate*, which minimizes the total number of accesses to main memory.

## 2.2 Classification of write backs

For analysis purposes, it is useful to classify write backs into three categories:

*Job-internal* write backs. These are write backs of dirty cache lines previously written to by the same job. We assume that the cost of job-internal write backs is included in the WCET of a task, since it does not depend on the scheduling policy used.

*Carry-in* write backs. These are write backs of dirty cache lines that were not written to by the job itself and that were present in the cache when the job was dispatched. We assume that the cost of carry-in write backs is not included in the WCET of a task, since it depends on the scheduling policy used. (The WCET is instead determined assuming an arbitrary, but clean initial cache state). Carry-in write backs can be further distinguished depending on whether they emanate from a job that is still active or not: Carry-in write backs from jobs that are still active can only come from lower priority preempted tasks. We refer to these as “lp-carry-in” write backs. Carry-in write backs



**Fig. 1** Example illustrating different kinds of write backs

from finished jobs can emanate from both lower and higher priority tasks. We refer to these as “finished-carry-in” write backs.

*Preemption-induced* write backs. These are write backs of dirty cache lines that were not written to by the task itself and that were introduced by a preempting task. Preemption-induced write backs can only come from higher priority jobs that are finished.

Consider Fig. 1 for an example schedule of three tasks containing the three types of write backs described above. In the example,  $x^*$  denotes a write to memory block  $x$ , whereas just  $x$  denotes a read from memory block  $x$ . Memory blocks  $a, c$  and  $b, d, f$  map to the same cache sets, and hence cache lines, respectively.

The first write to memory block  $a$  of task  $\tau_3$ , causes the eviction of  $c$ , which was written to by a finished job of task  $\tau_2$ , thus it causes a *finished-carry-in* write-back. On the other hand, the access to  $c$  in the second job of  $\tau_2$ , causes an *lp-carry-in* write back of  $a$ . The first access to  $b$  within task  $\tau_1$  evicts  $f$ , which was previously modified in the same job, thus causing a *job-internal* write back. Finally, the read of  $d$  in the second job of task  $\tau_2$  causes a *preemption-induced* write back of  $f$  which was previously written to by task  $\tau_1$ . Similarly, the reads of  $a$  and  $b$  in task  $\tau_3$  result in preemption-induced write backs of  $c$  and  $d$ , previously written to by task  $\tau_2$ .

### 2.3 Characterizing a task’s write backs

We assume that job-internal write backs are accounted for within WCET analysis. To bound carry-in write backs, and in the case of preemptive scheduling, preemption-induced write backs, we need to characterize the memory-access behaviour of each task. To do so, we introduce the following concepts:

An *Evicting Cache Block* (ECB) of task  $\tau_i$  is a memory block that may be accessed by task  $\tau_i$ . We denote the set of cache lines that evicting cache blocks of task  $\tau_i$  map to by  $ECB_i$ . Note ECBs have previously been considered in the analysis of cache-related preemption delays (Altmeyer et al. 2012).

A *Dirty Cache Block* (DCB) of task  $\tau_i$  is a memory block that may be written to by task  $\tau_i$ . We denote the set of cache lines that dirty cache blocks of task  $\tau_i$  map to by  $DCB_i$ .

A *Final Dirty Cache Block* (FDCB) of task  $\tau_i$  is a DCB that may still be cached at completion of the task. We denote the set of cache lines that final dirty cache blocks of task  $\tau_i$  map to by  $FDCB_i$ . (By definition,  $FDCB_i \subseteq DCB_i \subseteq ECB_i$ ).

By evicting dirty cache lines, ECBs may cause both carry-in and preemption-induced write backs. In preemptive scheduling, lp-carry-in write backs may occur due to DCBs, while preemption-induced and finished-carry-in write backs can only be due to FDCBs. In non-preemptive scheduling, preemption-induced write backs do not occur, and carry-in write backs are necessarily finished-carry-in write backs, and can thus only be due to FDCBs. With both scheduling paradigms, job-internal write backs can occur and carry-in write backs can occur due to jobs of all tasks, including the previous job of the same task.

### 3 Task model and basic analysis

In this section, we set out the basic task model used in the rest of the paper, and recapitulate existing response-time analyses for fixed-priority preemptive scheduling (FPPS) and fixed-priority non-preemptive scheduling (FPNS).

#### 3.1 Task model

We consider a set of sporadic tasks scheduled on a uniprocessor under either FPPS or FPNS. A task set  $\Gamma$  comprises a static set of  $n$  tasks  $\{\tau_1, \tau_2, \dots, \tau_n\}$ . Each task has a unique priority, which without loss of generality is given by its index. Thus task  $\tau_1$  has the highest priority and task  $\tau_n$  the lowest. Each task  $\tau_i$  gives rise to a potentially unbounded sequence of jobs separated by a minimum inter-arrival time or period  $T_i$ . Each job of task  $\tau_i$  has a bounded worst-case execution time  $C_i$ , and relative deadline  $D_i$ . Deadlines are assumed to be *constrained*, i.e.  $D_i \leq T_i$ . Note  $C_i$  is the worst-case execution time in the non-preemptive case, starting from an arbitrary clean cache. Thus  $C_i$  does not include the cost of reloading cache lines evicted due to preemption, or additional write backs that may be required when loading memory blocks into dirty cache lines. On the other hand, it does include the cost of job-internal write backs.

The worst-case response time  $R_i$  of task  $\tau_i$  is given by the longest time from the release of a job of the task until it completes execution. If the worst-case response time is not greater than the deadline ( $R_i \leq D_i$ ), then the task is said to be schedulable. The utilization  $U_i$  of a task  $\tau_i$  is given by  $U_i = \frac{C_i}{T_i}$  and the utilization of the task set is the sum of the utilizations of the individual tasks  $U = \sum_{i=1}^n U_i$ .

We use  $hp(i)$  and  $hep(i)$  to denote respectively the set of indices of tasks with priorities higher than, and higher than or equal to that of task  $\tau_i$  (including  $\tau_i$  itself). Similarly, we use  $lp(i)$  and  $lep(i)$  to denote respectively the set of indices of tasks with priorities lower than, and lower than or equal to that of task  $\tau_i$ .

### 3.2 Schedulability analysis for FPPS

For task sets with constrained deadlines scheduled using FPPS, the exact response time of task  $\tau_i$  may be computed according to the following recurrence relation (Audsley et al. 1993; Joseph and Pandya 1986):

$$R_i^P = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^P}{T_j} \right\rceil C_j \tag{1}$$

Iteration starts with  $R_i^P = C_i$  and ends either on convergence or when  $R_i^P > D_i$  in which case the task is unschedulable.

### 3.3 Schedulability analysis for FPNS

Determining exact schedulability of a task  $\tau_i$  under FPNS requires checking all of the jobs of task  $\tau_i$  within the worst-case priority level- $i$  busy period (Bril et al. 2009). (This is the case even when all tasks have constrained deadlines).

The worst-case priority level- $i$  busy period starts with an interval of blocking due to a job of the longest task of lower priority than  $\tau_i$ . Just after that job starts to execute, jobs of task  $\tau_i$  and all higher priority tasks are released simultaneously, and then re-released as soon as possible. Finally, the busy period ends at some time  $t$  when there are no ready jobs of priority  $i$  or higher that were not released strictly before time  $t$ .

In this paper, we make use of the following *sufficient* schedulability test for FPNS, applicable only to constrained-deadline task sets. It is based on a test originally given for non-preemptive scheduling on Controller Area Network (CAN) (Davis et al. 2007). This schedulability test considers two scenarios. Either the worst-case response time for task  $\tau_i$  occurs for the first job in the priority level- $i$  busy period, or for a subsequent job. The start time  $W_{i,0}^{NP}$  of the first job  $q = 0$  of task  $\tau_i$  in the worst-case priority level- $i$  busy period can be computed using the following recurrence relation:

$$W_{i,0}^{NP} = \max_{k \in lp(i)} C_k + \sum_{j \in hp(i)} \left( \left\lceil \frac{W_{i,0}^{NP}}{T_j} \right\rceil + 1 \right) C_j \tag{2}$$

and hence its worst-case response time is given by:

$$R_{i,0}^{NP} = W_{i,0}^{NP} + C_i \tag{3}$$

Subsequent jobs of task  $\tau_i$  may be subject to *push-through* blocking due to non-preemptive execution of the previous job of the same task. Let the jobs of task  $\tau_i$  be indexed by values of  $q = 0, 1, \dots$ , where  $q = 0$  is the first job in the busy period. We consider job  $q + 1$ , assuming that job  $q$  is schedulable (we return to this point later). Since job  $q$  is schedulable it completes by its deadline at the latest and therefore also by the release of job  $q + 1$ . Consider the length of the time interval from when job  $q$  starts executing to when job  $q + 1$  starts executing. Note when job  $q$  starts executing

there can be no jobs of higher priority tasks that are ready to execute. In the worst-case, jobs of all higher priority tasks may be released immediately after job  $q$  starts to execute. Thus an upper bound on the length  $W_{i,q+1}^{NP}$  of this interval can be computed using the following recurrence relation:

$$W_{i,q+1}^{NP} = C_i + \sum_{j \in hp(i)} \left( \left\lfloor \frac{W_{i,q+1}^{NP}}{T_j} \right\rfloor + 1 \right) C_j \tag{4}$$

Since we assume that job  $q$  completes by its deadline and deadlines are constrained ( $D_i \leq T_i$ ), then the interval  $W_{i,q+1}^{NP}$  must also upper bound the time from the release of job  $q + 1$  until it starts to execute. As job  $q + 1$  takes time  $C_i$  to execute, an upper bound on its worst-case response time is given by:

$$R_{i,q+1}^{NP} = W_{i,q+1}^{NP} + C_i \tag{5}$$

Assuming that job  $q = 0$  is schedulable according to (2) then schedulability of the second and subsequent jobs in the busy period can be determined by induction using (5).

We note the similarity between (2) and (4), and also between (3) and (5). Thus we may combine them obtaining an upper bound for the response time of task  $\tau_i$ , under FPNS. This upper bound may be compared with the task’s deadline to determine schedulability.

$$W_i^{NP} = \max_{k \in lep(i)} C_k + \sum_{j \in hp(i)} \left( \left\lfloor \frac{W_i^{NP}}{T_j} \right\rfloor + 1 \right) C_j \tag{6}$$

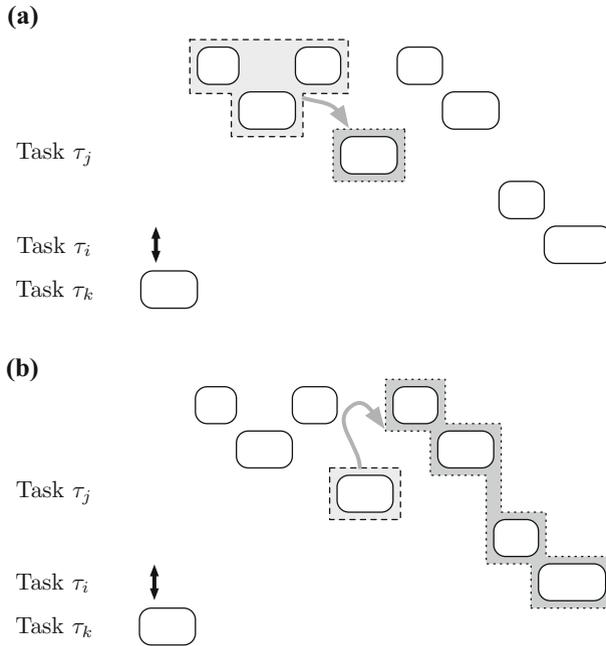
$$R_i^{NP} = W_i^{NP} + C_i \tag{7}$$

The analysis expressed in (5) can be improved by noting that the start time of job  $q$  must be at least  $C_i$  before the release of job  $q + 1$ , hence the response time upper bound given in (5) may be reduced by  $C_i$ . In this paper, for ease of presentation, we make use of the simpler test embodied in (6) and (7).

### 4 Write backs under FPNS

In this section, we extend the sufficient schedulability test for FPNS for constrained-deadline task sets given in (6) and (7) to account for carry-in write backs. In non-preemptive scheduling only job-internal and finished-carry-in write backs may occur. As discussed earlier, we assume that job-internal write backs are accounted for within WCET analysis.

We identify two methods of accounting for finished-carry-in write backs, which are illustrated in Fig. 2. In the first method, we associate with each job of a task, the carry-in write backs that occur *within the job*. This method is used in the *ECB-Only* and *FDCB-Union* approaches described in Sect. 4.1. By contrast, in the second method we



**Fig. 2** Carry-in write backs may be accounted for either, **a** within the job of the task  $\tau_i$  under analysis, or **b** in subsequent jobs of both higher (e.g.  $\tau_j$ ) and lower (e.g.  $\tau_k$ ) priority tasks

associate with each job of a task the carry-in write backs that occur in *subsequent jobs* due to dirty cache lines left by the job itself. This method is used in the *FDCB-Only* and *ECB-Union* approaches described in Sect. 4.2.

### 4.1 Carry-in write backs within the job

#### 4.1.1 ECB-Only approach

The number of ECBs provides an upper bound on the number of carry-in write backs a task suffers.<sup>1</sup> Thus, assuming timing compositionality (Hahn et al. 2013), the WCET of task  $\tau_i$ , including the cost of write backs, is bounded by

$$C'_i = C_i + WBT \cdot |ECB_i| \tag{8}$$

where *WBT* is an upper bound on the time to perform one write back. Replacing  $C_i$  by  $C'_i$  as defined above (and similarly  $C_k$  and  $C_j$ ), (6) and (7) can be used to derive worst-case response times accounting for write backs.

<sup>1</sup> Note that this holds for direct-mapped caches as well as for set-associative caches with LRU replacement. This is different from additional cache misses, which are not directly bounded by the number of ECBs (Burguière et al. 2009).

### 4.1.2 FDCB-Union approach

The ECB-Only approach can be improved upon by taking into account which cache lines may be dirty when a job is started. In non-preemptive execution, dirty cache lines at a job’s start are the final dirty cache lines left by other jobs.

When analyzing  $\tau_i$ ’s response time, we distinguish two types of finished-carry-in write backs: Those that are due to dirty cache lines introduced before  $\tau_i$ ’s release by tasks with lower or equal priority to  $\tau_i$ , represented by  $\delta_i$ , and those that are due to dirty cache lines introduced before and after  $\tau_i$ ’s release by tasks of higher priority than  $\tau_i$ , represented by  $\gamma_{i,j}^{wb}$ .

Each final dirty cache line of a task with priority lower than or equal to that of task  $\tau_i$  may result in at most one write back during  $\tau_i$ ’s response time, excluding write backs that occur during the blocking time. Write backs of these dirty cache lines can only occur within the response time of task  $\tau_i$  if the cache lines are accessed by (i.e. in the  $ECB_k$ ) some task  $\tau_k$  of priority  $i$  or higher. The term  $\delta_i$  accounts for these write backs. Note that we exclude from  $\delta_i$  cache lines that may be dirty due to higher priority tasks as such cache lines are accounted for by the  $\gamma_{i,j}^{wb}$  term introduced next, thus:

$$\delta_i = WBT \cdot \left| \left( \bigcup_{k \in lep(i)} FDCB_k \setminus \bigcup_{k \in hp(i)} FDCB_k \right) \cap \left( \bigcup_{k \in hp(i)} ECB_k \right) \right| \quad (9)$$

The number of finished-carry-in write backs that can be made during the execution of one job of task  $\tau_j$  due to dirty cache lines introduced by tasks of higher priority than  $\tau_i$  is upper bounded by  $\gamma_{i,j}^{wb}$ . Note that only cache lines accessed by task  $\tau_j$  (i.e. in  $ECB_j$ ) can be written back during the execution of a job of  $\tau_j$ .

$$\gamma_{i,j}^{wb} = WBT \cdot \left| \left( \bigcup_{k \in hp(i)} FDCB_k \right) \cap ECB_j \right| \quad (10)$$

We now adapt (6) and (7) to include the write backs ( $\gamma_{n+1,b}^{wb}$ ) that can occur within one job of a blocking task  $\tau_b$ ; the write backs ( $\delta_i$ ) that can occur during jobs other than that of a blocking task, due to dirty cache lines left by tasks of lower priority than  $\tau_i$  before the start of the busy period; and finally, the write backs ( $\gamma_{i,j}^{wb}$  and  $\gamma_{i,i}^{wb}$ ) that can occur within each of the other jobs that contribute to the response time of task  $\tau_i$ , due to dirty cache lines introduced by tasks of higher priority than  $\tau_i$ .

$$W_{i,WB}^{NP} = \max_{b \in lep(i)} (C_b + \gamma_{n+1,b}^{wb}) + \delta_i + \sum_{j \in hp(i)} \left( \left\lfloor \frac{W_{i,WB}^{NP}}{T_j} \right\rfloor + 1 \right) (C_j + \gamma_{i,j}^{wb}) \quad (11)$$

$$R_{i,WB}^{NP} = W_{i,WB}^{NP} + (C_i + \gamma_{i,i}^{wb}) \quad (12)$$

In the  $\gamma_{n+1,b}^{wb}$  term,  $n + 1$  denotes a priority that is lower than that of any task, thus  $\gamma_{n+1,b}^{wb}$  accounts for all carry-in write backs that may occur during the execution of a blocking task  $\tau_b$  due to cache lines left dirty by previous jobs of any task. In contrast,  $\gamma_{i,j}^{wb}$  and  $\gamma_{i,i}^{wb}$  need only cover write backs due to dirty cache lines from tasks of higher priority than  $\tau_i$ , since all other write backs are accounted for in  $\delta_i$ .

The ECB-Only approach pessimistically assumes that each time a task is executed the cache is full of dirty cache lines. The FDCB-Union approach improves upon this by more precisely modeling which cache lines could actually be dirty. FDCB-Union strictly dominates ECB-Only, meaning that any task set that is deemed schedulable according to the ECB-Only approach is guaranteed to be deemed schedulable using the FDCB-Union approach. This can be seen by first considering the  $C_j + \gamma_{i,j}^{wb}$  terms in (11) and (12). From (10), it follows that  $C_j + \gamma_{x,j}^{wb}$  cannot be greater than the value of  $C'_j$  used in (8) for any task  $\tau_j$  and index  $x$ , and hence cannot exceed the inflated WCET values used in the ECB-Only approach. Second, we must consider the additional contributions in the  $\delta_i$  term. For an FDCB to contribute to  $\delta_i$ , then from (9), that FDCB cannot be in  $FDCB_k$  of any task  $\tau_k$  with a priority higher than that of task  $\tau_i$ . Also, it must be in the  $ECB_i$  of task  $\tau_i$  or the  $ECB_k$  of some higher priority task  $\tau_k$ . If it is in  $ECB_i$  and contributes to  $\delta_i$  then from (10) it is not included in the  $\gamma_{i,i}^{wb}$  term in (12), thus the inflated WCET  $C'_i$  in the ECB-Only approach covers both this contribution to  $\delta_i$  and the  $\gamma_{i,i}^{wb}$  term in (12). Similarly, if the FDCB is in  $ECB_j$  and contributes to  $\delta_i$  then it is not included in the  $\gamma_{i,j}^{wb}$  term in (11), thus the inflated WCET  $C'_j$  in the ECB-Only approach again covers both this contribution to  $\delta$  and  $\gamma_{i,j}^{wb}$ . Finally, it serves only to consider a system with no FDCBs to see that FDCB-Union strictly dominates ECB-Only. At the other extreme, if all ECBs are also FDCBs, then FDCB-Union reduces to ECB-Only (with  $\delta_i = 0$ ).

## 4.2 Carry-in write backs in subsequent jobs

### 4.2.1 FDCB-Only approach

Instead of using  $\gamma_{i,j}^{wb}$  to mean the cost of carry-in write backs that occur *within* the execution of a job of task  $\tau_j$ , we can re-define  $\gamma_{i,j}^{wb}$  to cover the write backs that occur in *subsequent jobs* due to dirty cache lines left by a job of task  $\tau_j$ . This is achieved by assuming that all of these cache lines may be evicted by the subsequent jobs:

$$\gamma_{i,j}^{wb} = WBT \cdot |FDCB_j| \tag{13}$$

With this approach,  $\delta$  needs to account for *all* carry-in write backs due to cache lines that were dirty prior to  $\tau_i$ 's release:

$$\delta = WBT \cdot \left| \bigcup_k FDCB_k \right| \tag{14}$$

Finally, the final dirty cache lines that  $\tau_i$  leaves do not affect its own response time. As a consequence (12) can be simplified as follows (with (11) unchanged):

$$R_{i,WB}^{NP} = W_{i,WB}^{NP} + C_i \tag{15}$$

### 4.2.2 ECB-Union approach

The above approach can be improved by taking into account which of the dirty cache lines may actually be evicted by subsequent jobs of tasks which may execute within  $\tau_i$ 's response time (i.e. by also considering the cache lines ( $ECB_k$ ) accessed by each task  $\tau_k$  of priority  $i$  or higher).

$$\gamma_{i,j}^{wb} = WBT \cdot \left| FDCB_j \cap \left( \bigcup_{k \in lep(i)} ECB_k \right) \right| \tag{16}$$

Similarly, in the  $\delta_{b,i}$  term, we need only account for those dirty cache lines that may be evicted during  $\tau_i$ 's response time. This depends on the blocking task  $\tau_b$ :

$$\delta_{b,i} = WBT \cdot \left| \left( \bigcup_k FDCB_k \right) \cap \left( \bigcup_{j \in lep(i) \cup \{b\}} ECB_j \right) \right| \tag{17}$$

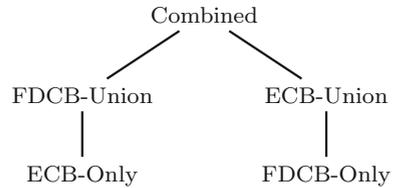
Hence we include  $\delta_{b,i}$  in the blocking term resulting in the following adaptation of (11):

$$W_{i,WB}^{NP} = \max_{b \in lep(i)} (C_b + \gamma_{i,b}^{wb} + \delta_{b,i}) + \sum_{j \in hp(i)} \left( \left\lfloor \frac{W_{i,WB}^{NP}}{T_j} \right\rfloor + 1 \right) (C_j + \gamma_{i,j}^{wb}) \tag{18}$$

The ECB-Union approach strictly dominates the FDCB-Only approach. This can be seen by comparing the  $\gamma_{i,j}^{wb}$  terms and the  $\delta_{b,i}$  terms. Comparing the  $\gamma_{i,j}^{wb}$  terms in (13) and (16) we note that surprisingly there is no advantage gained by ECB-Union, since  $FDCB_j \subseteq ECB_j$  and  $i \in lep(j)$  in all uses of this term, hence (16) effectively reduces to (13). Considering the  $\delta_{b,i}$  terms, if there are a number of lower priority tasks with FDCBs that are not present in the ECBs of tasks with priorities higher than or equal to  $\tau_i$  then (17) can improve upon (14), with dominance apparent from the set intersection.

We note that the ECB-Union and FDCB-Union approaches are incomparable, and hence we may form a combined approach by taking the minimum response time computed by either approach. By construction, this combined approach dominates both ECB-Union and FDCB-Union. Since it can be applied on a per task basis, the combined approach classifies more task sets as schedulable than can be found by using the ECB-Union and FDCB-Union approaches individually on each task set. Figure 3 illustrates these relationships via a Hasse diagram.

**Fig. 3** Hasse diagram illustrating the dominance relationships between different approaches to account for write backs under fixed-priority non-preemptive scheduling



**Table 1** Example task set

Task	C	T	ECB	DCB	FDCB
$\tau_1$	100	1000	{1, 4, 5}	{1}	{1}
$\tau_2$	100	1000	{2, 3, 4, 5}	{2, 3, 4}	{2, 3}
$\tau_3$	100	1000	{2, 3, 5}	{2, 3, 5}	{2, 3}
$\tau_4$	100	1000	{1, 2, 3, 4, 5, 6}	{1, 2, 3, 4, 5, 6}	{1}

### 4.3 Worked example

Below, we present a worked example illustrating the various approaches to analysing write backs under fixed-priority non-preemptive scheduling and their differences in performance. Table 1 gives the task set parameters.

For ease of presentation, we assume a write-back delay of 1 and choose task parameters so that only one job of each task may be released during another task’s response time. The sets of UCBs are assumed to be empty (i.e. we focus on write backs and do not consider CRPD due to cache misses).

#### 4.3.1 ECB-Only

The ECB-Only approach (8) effectively increases the tasks’ execution times by  $WBT \cdot |ECB|$ :  $C'_1 = 103$ ,  $C'_2 = 104$ ,  $C'_3 = 103$ ,  $C'_4 = 106$ . The response time is then computed using (6) and (7) giving  $R_1 = 209$ ,  $R_2 = 313$ ,  $R_3 = 416$ , and  $R_4 = 522$ .

#### 4.3.2 FDCB-Union

The FDCB-Union approach extends the ECB-Only approach by taking into account which cache lines may be dirty when a job is started. The term  $\delta_i$  accounts for dirty cache lines of lower or equal priority tasks and is computed using (9):  $\delta_1 = 1$ ,  $\delta_2 = 2$ ,  $\delta_3 = 0$ ,  $\delta_4 = 0$ . The term  $\gamma_{i,j}^{wb}$  accounts for write backs due to higher priority tasks and is computed using (10):

$$\gamma_{i,j}^{wb} \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 2 & 1 & - & - \\ 3 & 1 & 2 & - \\ 4 & 1 & 2 & 2 \end{array}$$

Further, we have:  $\gamma_{5,1}^{wb} = 1, \gamma_{5,2}^{wb} = 2, \gamma_{5,3}^{wb} = 2, \gamma_{5,4}^{wb} = 3$  and  $\gamma_{1,1}^{wb} = 0, \gamma_{2,2}^{wb} = 0, \gamma_{3,3}^{wb} = 2, \gamma_{4,4}^{wb} = 3$ . The response time is then computed using (11) and (12) giving  $R_1 = 204, R_2 = 306, R_3 = 408, \text{ and } R_4 = 511$ .

Note that the FDCB-Union approach dominates ECB-Only and results in shorter response times in this example.

### 4.3.3 FDCB-Only

The FDCB-Only approach accounts for write backs in subsequent jobs, instead of write backs in the execution of the job itself. Hence, the term  $\delta$  accounts for dirty cache lines prior to the release of the task under analysis and is given by (14) thus  $\delta_i = 3$ . The  $\gamma_{i,j}^{wb}$  term is given by (13):

$\gamma_{i,j}^{wb}$	1	2	3
2	1	–	–
3	1	2	–
4	1	2	2

Further,  $\gamma_{5,1}^{wb} = 1, \gamma_{5,2}^{wb} = 2, \gamma_{5,3}^{wb} = 2, \gamma_{5,4}^{wb} = 1$ . The response time is then computed using (11) and (15) giving  $R_1 = 205, R_2 = 306, R_3 = 408, \text{ and } R_4 = 509$ .

### 4.3.4 ECB-Union

The ECB-Union approach only differs from FDCB-Only in the  $\delta$  terms. This difference, although technically possible, is neither visible in this example, nor in the evaluation. Instead, the ECB-Union approach results in the same response times as the FDCB-Only approach.

We observe that this example suffices to highlight the incomparability between ECB-Union and FDCB-Union. The response time for task  $\tau_1$  is smaller with FDCB-Union (204 vs. 205), while the response time for task  $\tau_4$  is smaller with ECB-Union (509 vs. 511). The combined approach, taking the minimum response times (204, 306, 408, 509) thus dominates all others.

## 5 Write backs under FPPS

Response-time analysis for FPPS has previously been extended to account for preemption-related cache misses (Altmeyer et al. 2011, 2012) by introducing a term  $\gamma_{i,j}$  into the response-time equation for task  $\tau_i$  as follows:

$$R_i^P = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j^P}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{19}$$

To also account for additional write backs in preemptive scheduling, we extend the recurrence relation as follows:

$$R_i^P = \delta_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + \gamma_{i,j}^{miss} + \gamma_{i,j}^{wb}) \tag{20}$$

Here,  $\delta_i$  is used to account for write backs due to cache lines that were already dirty on release of  $\tau_i$  and are written back within its response time. Additional cache misses due to preemptions are captured by  $\gamma_{i,j}^{miss}$ . Any of the existing techniques, for example those introduced by Altmeyer et al. (2012), can be used to account for such misses. Finally,  $\gamma_{i,j}^{wb}$  is used to account for carry-in and preemption-induced write backs of cache lines that were written to after  $\tau_i$ 's release.

We further sub-divide  $\gamma_{i,j}^{wb}$  into  $\gamma_{i,j}^{wb-lp}$  and  $\gamma_{i,j}^{wb-fin}$ , such that  $\gamma_{i,j}^{wb} = \gamma_{i,j}^{wb-lp} + \gamma_{i,j}^{wb-fin}$ , where  $\gamma_{i,j}^{wb-lp}$  accounts for lp-carry-in write backs and  $\gamma_{i,j}^{wb-fin}$  accounts for finished-carry-in and preemption-induced write backs (see Sect. 2.2 for their definitions). In the following we introduce four different ways of computing  $\gamma_{i,j}^{wb-lp}$ . These combine with the analysis derived for  $\delta_i$  and  $\gamma_{i,j}^{wb-fin}$  to give the *DCB-Only*, *ECB-Union*, *ECB-Only* and *DCB-Union* approaches for analysing write backs under FPPS.

### 5.1 Initially dirty cache line write backs

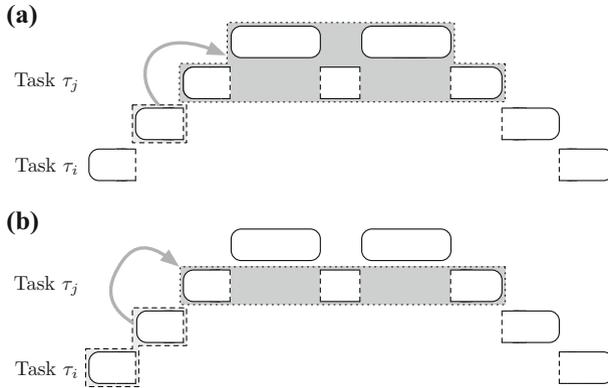
We first consider which cache lines may be dirty when the priority level- $i$  busy period starts that leads to the worst-case response time of a job of task  $\tau_i$ . Only tasks of lower priority than  $\tau_i$  may be active immediately before the start of this busy period, so the cache lines in  $\bigcup_{j \in lp(i)} DCB_j$  may all be in the cache and dirty. Further, the cache lines in  $\bigcup_{k \in hep(i)} FDCB_k$  may have been left dirty by finished jobs of higher priority tasks. Among all the dirty cache lines, we need only account for those that may be evicted within  $\tau_i$ 's response time. As only  $\tau_i$  and higher priority tasks can run during this interval, these are  $\bigcup_{k \in hep(i)} ECB_k$ , hence we obtain the following formula for  $\delta_i$ :

$$\delta_i = WBT \cdot \left| \left( \bigcup_{j \in lp(i)} DCB_j \cup \bigcup_{k \in hep(i)} FDCB_k \right) \cap \left( \bigcup_{k \in hep(i)} ECB_k \right) \right| \tag{21}$$

### 5.2 Lower priority carry-in write backs

To bound lp-carry-in write backs ( $\gamma_{i,j}^{wb-lp}$ ) due to preempted tasks, we identify two methods, both illustrated in Fig. 4.

- (a) the lp-carry-in write backs of dirty cache lines introduced by the job *immediately-preempted* by a job of  $\tau_j$  that occur within the response time of  $\tau_j$ , i.e. either executing  $\tau_j$  or a higher priority task.



**Fig. 4** Methods of accounting for lp-carry-in write backs. **a** Effect of immediately-preempted task (light grey) on all preempting tasks (dark grey). **b** Effect of preempted tasks (light grey) on immediately preempting task (dark grey)

(b) the lp-carry-in write backs of dirty cache lines introduced by *any preempted lower priority tasks* that occur within the execution of a job of  $\tau_j$ .

Using method (a), we define the DCB-Only and ECB-Union approaches, and with method (b), the ECB-Only and DCB-Union approaches.

### 5.2.1 DCB-Only approach

Using method (a), any task that could be active during the response time of task  $\tau_i$  and has a lower priority than task  $\tau_j$  (i.e. a task in the set  $aff(i, j) = hep(i) \cap lp(j)$ ) could be immediately preempted by task  $\tau_j$ , thus we obtain the following upper bound on the cost of write backs  $\gamma_{i,j}^{wb-lp}$  associated with jobs of task  $\tau_j$ :

$$\gamma_{i,j}^{wb-lp} = WBT \cdot \max_{h \in aff(i,j)} |DCB_h| \tag{22}$$

Note, when using this DCB-Only approach we assume that (21) is simplified ignoring the ECBs.

$$\delta_i = WBT \cdot \left| \bigcup_{j \in lp(i)} DCB_j \cup \bigcup_{k \in hep(i)} FDCB_k \right| \tag{23}$$

### 5.2.2 ECB-Union approach

The DCB-Only approach can be refined by noting that we are only interested in write backs of these dirty cache lines due to execution of tasks while the job of task  $\tau_j$  is

active, i.e. due to execution of  $\tau_j$  or a higher priority task (see Fig. 4) thus:

$$\gamma_{i,j}^{wb-lp} = WBT \cdot \max_{h \in aff(i,j)} \left| DCB_h \cap \bigcup_{l \in hep(j)} ECB_l \right| \tag{24}$$

### 5.2.3 ECB-Only approach

Using method (b), the lp-carry-in write backs of dirty cache lines introduced by *any preempted lower priority tasks* that occur within the execution of  $\tau_j$  are upper bounded by the ECBs of  $\tau_j$ :

$$\gamma_{i,j}^{wb-lp} = WBT \cdot |ECB_j| \tag{25}$$

Note, when using this ECB-Only approach we assume that (21) is simplified ignoring the DCBs.

$$\delta_i = WBT \cdot \left| \bigcup_{k \in hep(i)} ECB_k \right| \tag{26}$$

### 5.2.4 DCB-Union approach

The ECB-Only approach can be refined by noting that we are only interested in write backs of dirty cache lines introduced by *preempted lower priority tasks* (see Fig. 4). Note, that we do not need to account for lp-carry-in write backs due to dirty cache lines of tasks of lower priority than  $\tau_i$  as these are already accounted for in  $\delta_i$ .

$$\gamma_{i,j}^{wb-lp} = WBT \cdot \left| \left( \bigcup_{h \in aff(i,j)} DCB_h \right) \cap ECB_j \right| \tag{27}$$

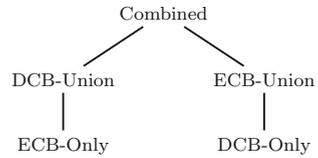
## 5.3 Finished-carry-in write backs

A job of task  $\tau_j$  can leave  $|FDCB_j|$  dirty cache lines, which may have to be written back within  $\tau_i$ 's response time. This yields the following simple bound on the cost of finished-carry-in and preemption-induced write backs:

$$\gamma_{i,j}^{wb-fin} = WBT \cdot |FDCB_j|. \tag{28}$$

One might assume that this bound can be improved by taking into account the evicting cache blocks of other tasks; however, as  $FDCB_j \subseteq ECB_j$ , then without further information, we must assume that the next job of task  $\tau_j$  will have to clean up the final dirty cache lines left by the previous job of the same task, thus no improvement is possible.

**Fig. 5** Hasse diagram illustrating the dominance relationships between approaches to account for write backs under fixed-priority preemptive scheduling



By construction, the ECB-Union approach dominates DCB-Only, and the DCB-Union approach dominates ECB-Only. Further, since ECB-Union and DCB-Union are incomparable we may form a combined approach that takes the smallest response time computed by either approach, and hence dominates both. Figure 5 illustrates these relationships via a Hasse diagram.

In some cases there could be pessimism in the analysis for FPPS as a result of write backs that are counted as both job-internal write backs in the WCET of a task, and also as carry-in write backs that occur when a task is preempted and a cache line is written back by the preempting task. As an example consider the sequence of accesses  $c^*$ ,  $c^*$ ,  $c^*$ ,  $d$  where memory blocks  $c$  and  $d$  are mapped to the same cache line, and  $*$  indicates a write. Here the read of  $d$  causes a job-internal write back of  $c$ . Preemption between the final write to  $c$  and the read of  $d$  could result in the preempting task writing back  $c$  (a carry-in write back), but no job-internal write back. In this case the analysis would over-approximate the total number of write backs. However, preemptions between the writes to  $c$  could induce a further carry-in write back in addition to the job-internal one. While there is some over-approximation in the analysis, our evaluations, in the next section, show that this over-approximation is small, with the combined approach close to the upper bound computed without write-back costs.

### 5.4 Worked example

Below, we present a worked example illustrating the various approaches to analysing write backs under fixed-priority preemptive scheduling and their differences in performance. Table 2 gives the task set parameters. (Note the example task set is the same as that used in Sect. 4.3. It is repeated here for ease of reference).

For ease of presentation, we again assume a write-back delay of 1 and choose task parameters so that only one job of each task may be released during another task’s response time. The sets of UCBs are assumed to be empty.

In the case of fixed-priority preemptive scheduling, all four approaches use the same response time equation (20), and the same  $\gamma_{i,j}^{wb-fin}$  terms to account for the finished carry-in write backs (28):  $\gamma_{-,1}^{wb-fin} = 1$ ,  $\gamma_{-,2}^{wb-fin} = 2$ ,  $\gamma_{-,3}^{wb-fin} = 2$ ,  $\gamma_{-,4}^{wb-fin} = 1$ . The approaches only differ in the  $\delta_i$  terms to account for initially dirty cache lines and the  $\gamma_{i,j}^{wb-lp}$  terms.

**Table 2** Example task set

Task	C	T	ECB	DCB	FDCB
$\tau_1$	100	1000	{1, 4, 5}	{1}	{1}
$\tau_2$	100	1000	{2, 3, 4, 5}	{2, 3, 4}	{2, 3}
$\tau_3$	100	1000	{2, 3, 5}	{2, 3, 5}	{2, 3}
$\tau_4$	100	1000	{1, 2, 3, 4, 5, 6}	{1, 2, 3, 4, 5, 6}	{1}

5.4.1 DCB-Only

Uses (23) to compute:  $\delta_1 = 6, \delta_2 = 6, \delta_3 = 6, \delta_4 = 3$ .

$\gamma_{i,j}^{wb-lp}$	1	2	3
2	3	–	–
3	3	3	–
4	6	6	6

$$R_1 = 106, R_2 = 210, R_3 = 315, R_4 = 426.$$

5.4.2 ECB-Union

Uses (21) to compute:  $\delta_1 = 3, \delta_2 = 5, \delta_3 = 5, \delta_4 = 3$ .

$\gamma_{i,j}^{wb-lp}$	1	2	3
2	1	–	–
3	1	3	–
4	3	5	5

$$R_1 = 103, R_2 = 207, R_3 = 312, R_4 = 421.$$

5.4.3 ECB-Only

Uses (26) to compute:  $\delta_1 = 3, \delta_2 = 5, \delta_3 = 5, \delta_4 = 6$ .

$\gamma_{i,j}^{wb-lp}$	1	2	3
2	3	–	–
3	3	4	–
4	3	4	3

$$R_1 = 103, R_2 = 209, R_3 = 315, R_4 = 421.$$

### 5.4.4 DCB-Union

Uses (21) to compute:  $\delta_1 = 3, \delta_2 = 5, \delta_3 = 5, \delta_4 = 3$ .

$\gamma_{i,j}^{\text{wb-lp}}$	1	2	3
2	1	–	–
3	2	3	–
4	3	4	3

$$R_1 = 103, R_2 = 207, R_3 = 313, R_4 = 418.$$

The example shows the dominance relationships of ECB-Union over DCB-Only and DCB-Union over ECB-Only, as well as the incomparability between ECB-Union and DCB-Union. The response time for task  $\tau_3$  is smaller with ECB-Union than with DCB-Union (312 vs. 313). Vice versa, the response time for task  $\tau_4$  is smaller with DCB-Union than with ECB-Union (418 vs. 421). The combined approach, taking the minimum response times (103, 207, 312, 418) thus dominates all others.

## 6 Sustainability of the analysis

The analysis given in this paper builds upon response-time analyses for FPPS and FPNS (see Sects. 3.2 and 3.3 respectively), integrating the effects of write-back costs. The response-time analyses used for FPPS and FPNS are both *sustainable* (Baruah and Burns 2006), meaning that a system that is deemed schedulable by the schedulability test used will not become unschedulable or be deemed unschedulable by the test if the task parameters are *improved*. These improvements include (i) reduced execution times, (ii) increased periods or minimum inter-arrival times, and (iii) increased deadlines.

We note that with the integration of write-back costs and CRPD given in Sects. 4 and 5, sustainability still holds with respect to the above parameters. Further, the analysis is sustainable with respect to improvements in the sets of cache lines considered, i.e. ECBs, DCBs, and FDCBs. (Here, by improvement we mean removal of one or more elements from a set, such that the new set is a subset of the old). This can be seen from the formulaes involved, since the response times computed are monotonically non-decreasing with respect to increases (addition of elements) to any of these sets. In all of the equations given in Sects. 4 and 5, for the overheads of write backs, the ECBs, DCBs, and FDCBs are combined using union, intersection, and cardinality operators. Thus the overheads are monotonically non-decreasing with respect to the content of those sets, and so any response time  $R'_i$  computed using  $ECB'_j, DCB'_j, FDCB'_j$  is no smaller than  $R_i$  computed using  $ECB_j, DCB_j, FDCB_j$  where  $ECB'_j \supseteq ECB_j, DCB'_j \supseteq DCB_j,$  and  $FDCB'_j \supseteq FDCB_j$ . The only exception that requires further consideration occurs in the FDCB-Union approach (Sect. 4.1) where (9) makes use of the set subtraction operator. Here, any reduction in the value of  $\delta_i$  due to an additional element in  $FDCB_k$  where  $k \in hp(i)$  is matched by an increase in  $\gamma_{i,j}^{\text{wb}}$  given by (10) for at least one of the higher priority tasks in  $hp(i)$ . Since each  $\gamma_{i,j}^{\text{wb}}$  term for a higher

priority task is included in the response time equation (11) at least once, the computed response time cannot decrease with the addition of any element to  $FDCB_j$ .

## 7 Experimental evaluation

In this section, we evaluate the performance of the different analyses introduced in Sects. 4 and 5 for write-back caches under fixed-priority preemptive and non-preemptive scheduling, as compared to no cache and a write-through cache. For both write-back and write-through caches, we assumed a write-allocate policy. Preliminary experiments showed that the difference between write allocate and no-write allocate for a write-through cache were minimal, with the former giving slightly better performance on the benchmarks studied.

We assume a timing-compositional processor with separate instruction and data caches. Each cache is direct-mapped and has 512 cache lines of size 32 bytes. Thus both caches have a capacity of 16 KB. Further, we assume a write-back latency  $WBT$  of 10 cycles. Cache misses also take 10 cycles, while non-memory instructions and cache hits take 1 cycle.

As a *proof-of-concept* for the analysis techniques, we obtained realistic estimates for WCETs and the sets of DCBs and ECBs, from the Mälardalen benchmark suite (Gustafsson et al. 2010) and the EEMBC Benchmark suite (EEMBC 2016) (Sect. 9 explains how this was done). Table 3 shows the WCETs (without inter-task interference) assuming a write-back cache ( $C^{wb}$ ), a write-through cache ( $C^{wt}$ ), and no data cache ( $C^{nc}$ ) for the selected benchmarks. Table 4 shows the number of UCBs, ECBs, DCBs, and FDCBs. We note that these stand-alone WCETs are a substantial factor of 1.4 to 3.0 times lower with a write-back cache than with write through, and 2 to 9 times lower than with no data cache. Since we assume a separate instruction and data cache, the UCB and ECB values are shown separately for each cache.

We note that fixed-priority non-preemptive scheduling suffers from the long task problem, whereby task sets that contain some tasks with short deadlines and others with long WCETs are trivially unschedulable due to blocking. To ameliorate this problem, we only selected benchmarks for Table 4 where the stand-alone WCETs were in the range [7000:70,000] cycles. This interval corresponds to the most populated range where the smallest and largest WCETs differ by a factor of 10. This restriction has little effect on the results for FPPS, while also providing task sets that can actually be scheduled using FPNS.

We evaluated the guaranteed performance of the various approaches on a large number of randomly generated task sets (10,000 per utilization level for the baseline experiments, and 200 per level for the weighted schedulability (Bastoni et al. 2010)) experiments. The task set parameters were generated as follows:

- The default task set size was 10.
- Each task was assigned data from a randomly chosen row of Table 4, corresponding to code from the benchmarks.
- The task utilizations ( $U_i$ ) were generated using UUnifast (Bini and Buttazzo 2005).
- Task periods were set based on utilization and the stand-alone WCET for a write-back cache, i.e.,  $T_i = C_i^{wb}/U_i$ .

**Table 3** Data from the Mälardalen and EEMBC benchmarks used for evaluation

Name	$ UCB^I $	$ ECB^I $	$ UCB^D $	$ ECB^D $	$ DCB $	$ FDCB $
cnt	12	82	21	68	28	28
compress	21	71	53	103	60	60
countneg	15	77	59	103	66	66
crc	19	89	25	73	40	39
expint	16	76	11	42	13	13
fdct	52	144	15	48	19	19
fir	22	83	17	57	17	16
jfdctint	46	145	17	53	23	23
loop3	7	309	9	42	12	12
ludcmp	38	128	21	61	28	28
minver	103	213	18	71	33	33
ns	14	70	9	116	13	11
nsichneu	345	494	52	95	54	53
qurt	61	132	14	49	17	17
select	47	124	10	49	16	16
sqrt	51	102	11	48	16	16
statemate	92	167	25	68	21	20
a2time	16	122	8	100	69	67
aifirf	25	141	33	188	161	54
basefp	11	88	15	512	507	467
canrdr	8	40	9	371	195	186
iirflt	35	288	28	259	147	138
pntrch	24	38	20	237	176	70
puwmod	3	50	5	512	307	275
rspeed	8	53	7	122	71	70
tblock	12	115	14	125	71	71

- Task deadlines were implicit  $D_i = T_i$ .
- Task priorities were in deadline-monotonic order.
- Tasks were placed in memory sequentially in priority order, thus determining the direct mapping to cache.

Figures 6 and 7 show the baseline results for FPPS and FPNS respectively (the graphs are best viewed online in colour). Table 5 summarises these results using the weighted schedulability measure (Bastoni et al. 2010).

Additional experimental results showing how this measure varies with the number of tasks and with the memory latency are given in the next subsection on weighted schedulability.

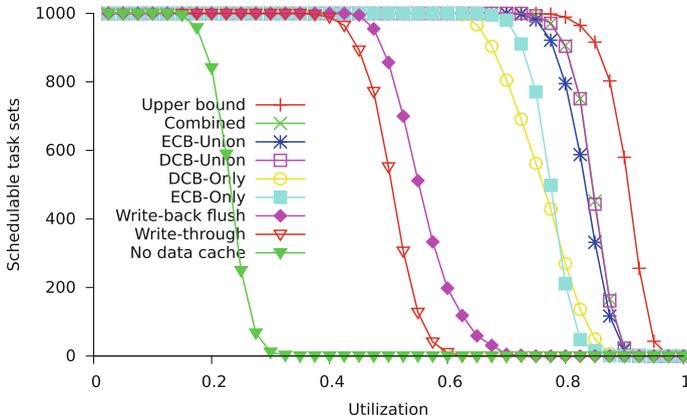
The lines in the figures correspond to the four different approaches, plus the combined approach, along with results for a write-through data cache and a system with no data cache. The first line refers to an optimistic *upper bound* where we assumed

**Table 4** WCETs from the Mälardalen and EEMBC benchmarks used for evaluation

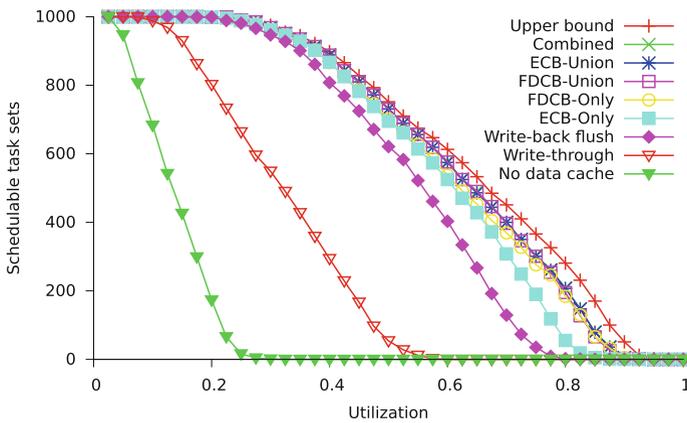
Name	$C^{wb}$	$C^{wt}$	$C^{wt}/C^{wb}$	$C^{nc}$	$C^{nc}/C^{wb}$
cnt	9325	13485	1.44	24565	2.63
compress	10673	18713	1.75	43443	4.07
countneg	36180	57250	1.58	114340	3.16
crc	68889	133909	1.94	272859	3.96
expint	9268	15208	1.64	31098	3.35
fdct	7883	16793	2.13	38423	4.87
fir	8328	18998	2.28	43668	5.24
jfdctint	9711	18621	1.91	39181	4.03
loop3	14189	28729	2.02	57929	4.08
ludcmp	10058	15948	1.58	39668	3.94
minver	18976	30616	1.61	54746	2.88
ns	27464	37674	1.37	98634	3.59
nsichneu	18988	24458	1.28	66808	3.51
qurt	10473	16003	1.52	23573	2.25
select	8981	17031	1.89	30331	3.37
sqrt	27667	40537	1.46	59117	2.13
statemate	64638	195778	3.02	581908	9.00
a2time	12655	22975	1.81	53815	4.25
aifirf	44898	86768	1.93	181698	4.04
basefp	50491	92221	1.82	213771	4.23
canldr	32641	65211	1.99	156611	4.79
iirflt	29995	56995	1.90	127605	4.25
pnrch	23887	43137	1.80	109257	4.57
puwmod	48782	97072	1.98	239752	4.91
rspeed	10913	21393	1.96	51713	4.73
tblock	12533	25493	2.03	58813	4.69

the stand-alone WCETs for write-back caches, but without any cost for write backs. This line upper bounds the performance of any sound analysis for write-back caches, and thus gives an indication of the precision of the analyses introduced in this paper.

The line *write-back flush* corresponds to a pessimistic analysis for write-back caches. In the case of FPNS, this analysis assumes that the entire cache is dirty and is flushed (written back) at the start of each task. To account for this, the WCET for each task is increased by  $N \cdot WBT$ , where  $N$  is the number of caches lines (e.g. 512) and  $WBT$  is the time to write back one cache line (e.g. 10 cycles). In the case of FPPS, not only could the entire cache be dirty and require writing back at the start of each preempting task, it could also be dirty at the end of each preemption and so also require writing back by the preempted task. The *write-back flush* analysis for FPPS therefore assumes that the entire cache is dirty and is flushed (written back) at both the start and at the end of each task. To account for this, the WCET for each task is increased



**Fig. 6** Number of schedulable task sets (FPPS)



**Fig. 7** Number of schedulable task sets (FPNS)

**Table 5** Weighted schedulability measure for FPNS and FPPS

Approach	FPPS	FPNS
Write-back (upper bound)	0.793458	0.445750
Combined	0.693003	0.412270
(F)DCB-Union	0.692087	0.411087
ECB-Union	0.672489	0.396159
(F)DCB-Only	0.561542	0.396159
ECB-Only	0.581876	0.365523
Write-back (flush)	0.304987	0.305039
Write-through	0.249231	0.112666
No data cache	0.052548	0.021463

by  $2N \cdot WBT$ . The results for *write-back flush* lower bound any useful analysis for a write-back cache.

For preemptive scheduling, in *all* cases, we include the cost of additional cache misses due to CRPD using the UCB-Union approach (Altmeyer et al. 2012).

The results shown in Figs. 6 and 7 indicate that the guaranteed performance obtained for write-back caches using the analyses introduced in this paper exceeds that which can be obtained for write-through caches. The new methods also provide a substantial improvement over the pessimistic *write-back flush* analysis, which in turn has an advantage over analysis for write-through caches. This shows that the gain from using a write-back cache comes from a combination of reduced WCETs and accurate analysis.

Further, the *upper bound* line indicates that the combined approaches used to analyse write-back cache offer a high degree of precision.

In Figs. 6 and 7 the ECB-Union approaches are outperformed by DCB-Union and FDCB-Union respectively. We note that this is not always the case as shown by the worked example in Sects. 5.4 and 4.3. In our experiments, the performance of the DCB-Union approach for FPPS and the FDCB-Union approach for FPNS is close to that of the associated combined approach. The reason for this is the relatively weak performance of the ECB-Union approach in each case. This occurs because the sets of ECBs for the benchmark tasks are substantially larger than the sets of DCBs and FDCBs. This degrades the relative performance of the ECB-Union approaches, particularly for low priority tasks which are the most critical to task set schedulability. (For a low priority task, the union of ECBs over all higher priority tasks may well cover all of the cache).

## 7.1 Weighted schedulability

The weighted schedulability measure  $W_y(p)$  for a schedulability test  $y$  and parameter  $p$ , combines results for all task sets generated for a set of equally spaced utilization levels (e.g. from 0.025 to 0.975 in steps of 0.025). Let  $S_y(\tau, p)$  be the binary result (1 or 0) of schedulability test  $y$  for a task set  $\tau$  assuming parameter  $p$ .

$$W_y(p) = \left( \sum_{\forall \tau} u(\tau) \cdot S_y(\tau, p) \right) / \sum_{\forall \tau} u(\tau) \quad (29)$$

where  $u(\tau)$  is the utilization of task set  $\tau$ . Weighting the results by task set utilization reflects the higher value placed on being able to schedule higher utilization task sets.

Figures 8 and 9 show how the weighted schedulability measure varies with task set size for FPPS and FPNS respectively. With preemptive scheduling, the relative performance of the different approaches remains consistent, with an overall gradual decline in schedulability as the number of tasks increases. This is due to an increase in the number of tasks increasing the number of preemptions and to some degree also their cost. (With FPPS, it is also simply harder to schedule task sets with increasing numbers of tasks, even without considering overheads).

With FPNS, as the number of tasks increases, the WCET of each task in relation to its period and deadline tends to decrease. This enables an overall increase in schedulability

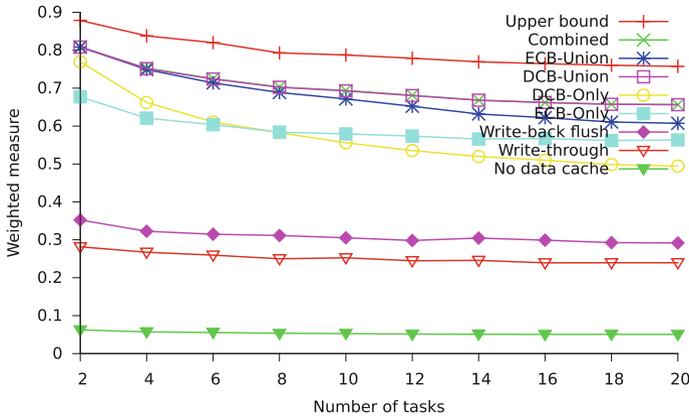


Fig. 8 Weighted schedulability versus number of tasks (FPPS)

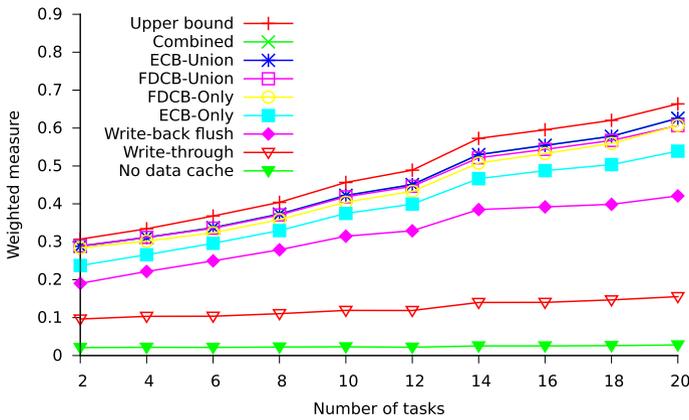
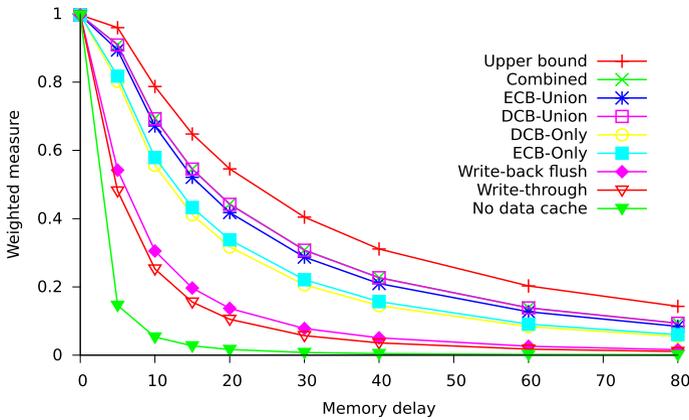


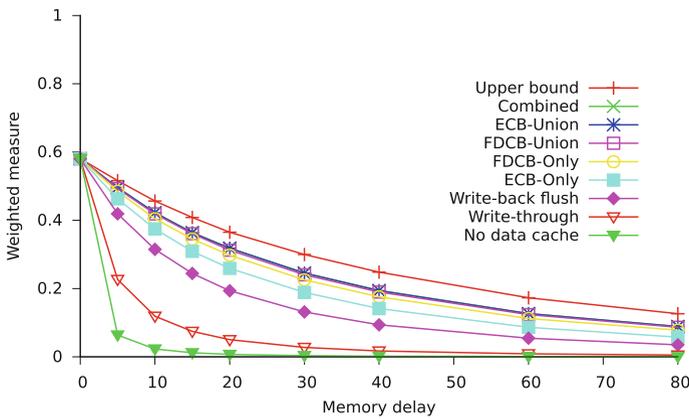
Fig. 9 Weighted schedulability versus number of tasks (FPNS)

with a write-back cache; however, at the very low level of schedulability achieved by write-through cache and no cache, schedulability is more dependent on a random choice of tasks with similar WCETs and hence similar deadlines, which avoid the long task problem. This becomes rarer with more tasks counteracting the previous effect.

Figures 10 and 11 show how the weighted schedulability measure varies with memory delay (time for write back or write through) for FPPS and FPNS respectively. Both figures show that as expected, increasing the memory delay has a detrimental effect on schedulability. As the memory delay increases, the larger number of writes to memory with a write-through cache becomes more heavily penalized and the relative performance of that approach (and no cache) deteriorates rapidly. We observe that for the benchmarks studied in our experiments, the guaranteed performance obtained with a write-back cache under FPPS was similar to that for a write-through cache when the latter was used on a higher performance system with one quarter of the memory delay



**Fig. 10** Weighted schedulability versus memory latency (FPNS)



**Fig. 11** Weighted schedulability versus memory latency (FPNS)

(e.g. 20 vs. 5 cycles, 40 vs. 10 cycles, or 80 vs. 20 cycles). For FPNS, where long task execution times have an increased impact on schedulability, the difference was even more stark, with the guaranteed performance obtained with a write-back cache with a memory delay of 40 cycles similar to that with a write-through cache with a delay of 5 cycles.

### 8 Write buffers

In this section, we discuss *write buffers* and their use, predominantly in improving the performance of write-through caches. At the end of the section we discuss the use of write buffers for write-back caches.

The key performance issue with a write-through cache is that the processor can potentially stall each time there is a write access, i.e. it may have to wait until the write to memory completes before continuing with subsequent instructions. This problem

can, to a large extent, be remedied via the use of a *write buffer*. A write buffer is a small buffer that operates between the cache and main memory. It holds data that is waiting to be written to memory. When a write occurs, the address and data (block) are placed in the write buffer. This allows the processor to continue with subsequent instructions, while the write to memory occurs in parallel via the write buffer.

Write buffers are characterized by a *depth* (indicating the number of entries), and a *width* (which is typically the same as a cache line), as well as the policies defining their operation. These policies include: (i) the *local-hazard* policy, which determines what happens when a read access occurs to an address that is currently in the write buffer; (ii) the *coalescence policy*, which determines what happens when a write access occurs to an address that is currently in the write buffer, and finally (iii) the *retirement policy*, which determines when write buffer entries are retired, i.e. written to memory. We discuss these policies in more detail below. (The interested reader is also referred to the work of Skadron and Clark (1997), which discusses write buffer design from the perspective of improving average-case performance).

### 8.1 Local hazard policy

Care is needed in the design of a write buffer, since a naive design could potentially result in data inconsistency, termed a *local hazard*, as follows: If a read occurs which is a cache miss, but the data is in the write buffer waiting to be written to memory, then reading from memory could result in an inconsistent value being obtained. To avoid this hazard there are two possible options that we consider (i) *read from the write buffer* or (ii) *full flush of the write buffer* and then read from memory. (More complex schemes are possible, such as flushing the write buffer only as far as necessary to write the required data to memory, or flushing only the specific item. They are not considered here).

### 8.2 Coalescence policy

Entries in a write buffer consist of an address and a block of data. The latter is typically the same size as a cache line. When a write occurs and there are no entries in the write buffer, then the block of data is copied to the write buffer and the specific word that is being written is marked as valid via a flag bit. The flag bit indicates that the word should later be written to memory.

If a write occurs to an address that is already in an entry in the write buffer then it could potentially be coalesced. In this case the entry containing the address is found in the buffer and the appropriate word of data is updated and marked as valid. We refer to this mechanism as *write merge*. Merging writes in this way has the advantage that it enables multiple writes to the same address or to the same block to be coalesced, resulting in fewer writes to memory. The write-merge mechanism has similarities to a write-back cache, in that it takes advantage of the spacial locality of writes. Merging writes also makes better use of the limited capacity of the write buffer.

The alternative to merging writes is to simply add a new entry to the write buffer on each write. This still facilitates latency hiding, since the processor is able to continue

with other instructions while writes to memory take place; however, it does not take advantage of spatial locality. We refer to this approach as *no-write merge*. While the average-case performance of no-write merge is typically worse than write merge, it has some advantages in terms of timing composition and guaranteed worst-case performance.

### 8.3 Retirement policies

The *retirement policy* determines when entries are retired from the write buffer, i.e. written to memory. Entries in a write buffer are typically processed in FIFO order.

The two main approaches are: (i) *eager retirement* where write-buffer entries are written to memory as soon as possible, and (ii) *lazy retirement* where write-buffer entries are written to memory as late as possible. (With lazy retirement, no entries are written back until the buffer becomes full and a write occurs that needs a new entry in the buffer). Eager retirement has the advantage that it keeps the buffer as empty as possible, with the aim of avoiding processor stalls due to a write to a full buffer. However, it has the disadvantage that, as data stays in the buffer for the minimum amount of time, there is little opportunity to take advantage of reads from the write buffer or write merging. Lazy retirement has the advantage that it keeps entries in the buffer as long as possible, maximising the potential for write merging and reads from the buffer, assuming that those mechanisms are employed. Lazy retirement has the disadvantage that once the buffer is full it stalls the processor on every write that requires a new buffer entry. In short, lazy retirement makes the write buffer behave in a similar way to a small FIFO cache.

There are a number of more complex options that are possible. For example, only retiring the oldest entry in the buffer when the number of entries exceeds half the buffer size. This approach aims to avoid the buffer becoming full and stalling writes, while also allowing entries to persist in the buffer with the advantages that brings. Other mechanisms retire entries when they get to a certain age measured in processor cycles. In this paper we only consider eager and lazy retirement.

### 8.4 Timing composition

It is important when analysing the worst-case performance of caches and associated buffering mechanisms that the results obtained are *timing compositional*, that is the local worst-case behaviours can be summed up to give a bound on the overall worst-case performance. It is known that certain designs, for example FIFO and PLRU caches, exhibit behaviours whereby a small change in cache contents due to preemption can result in an unbounded increase in the number of cache misses (see pp. 56–57 of (Altmeyer 2013) for worked examples). Such designs are not timing compositional and present a substantial challenge in terms of analysing their worst-case performance. They have performance that is dependent on the initial state, with an empty cache not necessarily representing the worst-case.

In this subsection, we explore how certain combinations of the policies defining write-buffer operation can result in domino effects. These effects mean that it is not

possible to bound the write buffer related preemption delay with a constant value. This effectively prevents an integrated analysis with fixed-priority preemptive scheduling (Schneider 2000).

## 8.5 Domino effects

We now show that some combinations of policies can result in domino effects.

### 8.5.1 Reading from the write buffer combined with lazy retirement

Let the write buffer comprise just one slot (entry) that can hold an address and a word of data. Further, lazy retirement is used. Note, the write merge / no-write merge policy is irrelevant to this example.

Consider the following sequence of memory accesses, where  $*$  indicates a write:  $a^*, b, a, b, a, b, a, b, a, \dots$ , and  $a$  and  $b$  are mapped to the same set in a direct mapped cache. Executing this sequence of accesses results in the following behaviour. On the first access,  $a$  is placed in the cache and copied to the write buffer. Due to lazy retirement  $a$  remains in the write buffer for the rest of the sequence. Each read access to  $b$  then evicts  $a$  from the cache. Each subsequent read access to  $a$  is then serviced from the write buffer, but nevertheless evicts  $b$  from the cache. The result is that all accesses to  $b$  are misses and need to be serviced from memory, while all accesses to  $a$  except the first are hits, serviced from the write buffer.

Now consider what happens if there is a preemption between accesses  $a^*$  and  $b$ , assume that the preemption makes a write access  $c^*$ . This write stalls the processor while  $a$ , which is in the write buffer, is written to memory. The write buffer now contains  $c$ . Returning to the preempted sequence, we see that every access to  $a$  has now become a miss. It can no longer be serviced from the write buffer, and instead must be serviced from memory.

Trivially, this domino effect extends to buffers of size 1 or more. We note that with eager retirement the effect cannot persist indefinitely, since entries are written to memory as soon as possible.

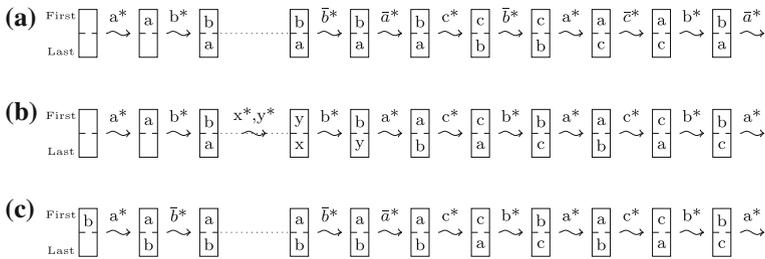
### 8.5.2 Write merge combined with lazy retirement

Let the write buffer be of depth 2, with each slot able to hold an address and a word of data. Further, lazy retirement is used. The local hazard policy is irrelevant to this example, since there are no reads.

Consider the following sequence of memory accesses all of which are writes:  $a^*, b^*, b^*, a^*, c^*, b^*, a^*, c^*, b^*, a^*, \dots$  (i.e. repeating with further sub-sequences of  $c^*, b^*, a^*$ ).

The write buffer contents are depicted in Fig. 12a for execution without preemption, Fig. 12b for execution with preemption and Fig. 12c for execution without preemption, starting from a non-empty buffer.

Without preemption (see Fig. 12a), every second write in the final repeating sub-sequence  $c^*, b^*, a^*$  is merged into the write buffer and therefore does not cause a stall.



**Fig. 12** Domino effect with write merge and lazy retirement. Note  $\bar{b}$  indicates a write merge to address  $b$  in the write-buffer. **a** Execution without preemption. **b** Execution with preemption. **c** Execution without preemption, starting with a non-empty write buffer

Such merged writes are marked with a bar in the figure, e.g.  $\bar{b}^*$ . However, if preemption occurs after the initial writes  $a^*, b^*$  (see Fig. 12b), altering the write buffer contents to  $x$  and  $y$  then this results in every write in the final repeating sub-sequence  $c^*, b^*, a^*$  causing a stall, since it is not to an address in the buffer. This effect persists indefinitely, giving a potentially unbounded increase in execution time.

Figure 12c illustrates what happens when execution is not preempted, but starts from a non-empty write buffer containing  $b$ . This has the effect of switching the order of  $a$  and  $b$  in the write buffer, which causes every write in the final repeating sub-sequence  $c^*, b^*, a^*$  to cause a stall. This example illustrates that an empty write buffer does not necessarily result in the worst-case behaviour.

We note that these domino effects extend to buffers of size 2 or more by using longer sub-sequences.

### 8.5.3 Write merge combined with eager retirement

Let the write buffer be of depth 3, with each slot able to hold an address and a word of data. Further, the buffer operates as follows: Writes are retired from the buffer as soon as possible in FIFO order. The time to retire an entry from the buffer is substantially longer than the time for an access that does not go to memory. While an entry is being retired, it cannot be merged into by another write, for example if the write buffer contains entries  $a, b$ , and  $d$ , then while  $d$  is being retired,  $a$  and  $b$  can be merged into, but  $d$  cannot. The local hazard policy is irrelevant to this example, since there are no reads.

We modify the example given in Sect. 8.5.2 for lazy retirement by adding a further write  $d^*$  at the beginning of the sequence. Now in the case without preemption, shown in Fig. 13a,  $d$  immediately starts being retired, meanwhile, writes  $a^*$ , and  $b^*$  fill the rest of the buffer. At this point,  $a$  and  $b$  can be merged into but  $d$  cannot. The next two writes  $b^*$  and  $a^*$  both merge into the buffer. The write access to  $c^*$  then stalls until retirement of  $d$  to memory is complete. Once that happens, the buffer contains  $c, b$ , and  $a$ , and  $a$  starts being retired and so becomes unavailable for merges. It is easy to see that as the sequence progresses, the entries available for merging are identical to those shown for a buffer of size 2 in Fig. 12a, with a further entry that is in the process of being retired and is therefore not available for merging. As before, every second



of each job. This junk will need to be written to memory before a further write access can make use of the buffer. Since the address of the junk is assumed to be unknown, any read is also assumed to flush the junk to memory. With FPPS, we also need to account for an additional preemption cost that equates to flushing the buffer. Since the write buffer only holds one entry, this additional per-job overhead has very little impact on schedulability.

We now explain why this additional preemption cost provides a sound upper bound on the write buffer related preemption delay (WBRPD) that needs to be included, per preempting job, in the response time computation for the preempted task. The only benefit that can be obtained with a write buffer of depth 1 is when one (or more) subsequent writes merge into the buffer. Since the buffer is of depth 1, this can only happen when consecutive writes occur to the same block. For example in the sequence:  $r1, r2, w1^*, r3, w1^*, r4, w1^*, w2^*$ , consecutive accesses to the same block are labelled  $w1^*$ ,  $r_x$  represent reads to different addresses, and  $w2$  is a write to a different address. Preemption between the writes to  $w1$  either does not replace  $w1$  in the buffer, in which case the bound on the preemption cost trivially holds, or it does replace  $w1$ . If it replaces  $w1$ , then the preempting job has already paid for the write of  $w1$  to memory within its own execution time (as the junk assumed to be in the buffer when it started to execute). The preempted job has an additional cost of writing the junk left in the buffer by the preempting task so that the second write access to  $w1$  can reside in the buffer. Any subsequent write access to  $w1$  would then merge into the buffer as before. Note the additional cost may be incurred either via a read to the same address as the junk, thus flushing the buffer, or by the write access  $w1$  itself. Either way, the extra cost is at most a single write to memory, and is covered by the WBRPD.

*An alternative for FPNS:* with FPNS and a write buffer of depth  $> 1$ , domino effects relating to initial buffer contents can be avoided by ensuring that each new job starts execution with an empty write buffer. This can be achieved by each job flushing the write buffer on completion. We note that some architectures, including the ARM 9, provide an instruction that flushes the write buffer. Sound analysis for FPNS could in this case be obtained by assuming such an instruction at the end of the code for each task. This would enable analysis of FPNS with a write buffer of arbitrary but known depth.

We note that write merge with a buffer of depth greater than 1 is challenging for analysis of real systems, since a single write to an unknown address has the potential to set up a domino effect.

## 8.7 Analysis of no-write merge

In this subsection, we provide analysis for no-write merge and full flush on local hazard, a combination which does not suffer from domino effects. Since the application of these two policies means that there is nothing to be gained from entries that are in the write buffer, the optimum retirement policy to use in conjunction with them is eager retirement.

With its operation defined by the above policies, the only advantage that the write buffer conveys is to hide the latency of writes. It follows that the maximum write

buffer related preemption delay (WBRPD) that can occur is when a preempting job delivers a full write buffer back to the preempted job. The preempted job then suffers a delay while these entries are retired, for example as a consequence of making a read to one of them. The WBRPD thus equates to the product of the buffer depth  $M$  and the Write Back Time  $WBT$  for one entry in the buffer. The WBRPD can be modeled by inflating the execution time of each preempting task by this amount. Since we assume that there may also be junk in the write buffer at the start of each job, the baseline execution times also need to include the time to flush the write buffer at the start of each job. Thus a simple analysis for no-write merge can be obtained by inflating all execution times by  $2M \cdot WBT$ .

We note that since the write buffer depth is typically only 1–4 entries, this overhead has only limited impact on schedulability. (A more detailed analysis is left for future work; however, it is unlikely that substantial improvement can be obtained, since the number of DCBs and FDCBs is typically much larger than the depth of the write buffer).

## 8.8 Write buffers and write-back caches

While write buffers are most useful in improving the performance of write-through caches, they can also be used to improve the performance of write-back caches.

In theory, the domino effects noted previously with a write buffer and a write-through cache also apply in the case of write-back caches. The precise sequences needed to show this behaviour differ however, since writes first have to be evicted from the cache before they are written to the buffer. This can be achieved by interspersing reads to other addresses that share the same cache set as the write that needs to be evicted.

In practice, since a write-back cache already captures the spacial locality of writes, there is little advantage to be gained from using a write buffer with a depth of more than 1, since that is already sufficient to provide latency hiding. For example, the Renesas SH7705, SH7750, and the AM1806 ARM low power microprocessor (based on the ARM926EJ-S) all have a write buffer of depth 1 to improve performance in write-back cache configurations. In the following, we therefore only consider write buffers of depths either 0 or 1 for a write-back caches.

## 8.9 Evaluation with write buffers

In this subsection, we examine the analysable performance of write-back caches with write buffers of depths 0 and 1, and write-through caches with write buffers of depths 0, 1, 2, and 4. (The Renesas SH7750 and AM1806 ARM have write buffers of depths 2 and 4 respectively for write-through cache configurations). In the case of write-back caches, we assume full flush on local hazards, and eager retirement. We assume that the buffer contents immediately start to be retired, and that a write cannot be merged while the contents are being retired, hence no-write merge. In the case of write-through caches, we assume write merge, full flush on local hazards, and lazy retirement. Worst-case execution time estimates for the EEMBC and Mälardalen benchmarks are given

**Table 6** Execution times estimates for the Mälardalen and EEMBC benchmarks used for evaluation

Name	$C^{wb-0}$	$C^{wb-1}$	$C^{wt-0}$	$C^{wt-1}$	$C^{wt-2}$	$C^{wt-4}$	$C^{nc}$
cnt	9325	9325	13485	9815	9745	9695	24565
compress	10673	10673	18713	16963	16403	14863	43443
countneg	36180	36180	57250	56500	48450	37260	114340
crc	68889	68869	133909	79469	79419	69759	272859
expint	9268	9268	15208	12548	9508	9448	31098
fdct	7883	7883	16793	11403	10203	9253	38423
fir	8328	8318	18998	13718	8858	8548	43668
jfdctint	9711	9711	18621	14141	12291	11601	39181
loop3	14189	14189	28729	26909	14369	14349	57929
ludcmp	10058	10048	15948	13178	11628	10828	39668
minver	18976	18976	30616	23226	22276	20026	54746
ns	27464	27444	37674	27704	27644	27624	98634
nsichneu	18988	18954	24458	20068	20028	19988	66808
qurt	10473	10473	16003	12293	11483	10873	23573
select	8981	8971	17031	12181	11251	9961	30331
sqr	27667	27667	40537	34607	31037	28147	59117
statemate	64638	64628	195778	120958	102918	96858	581908
a2time	12655	12468	22975	22825	12645	12635	53815
aifirf	44898	41638	86768	77528	41508	41508	181698
basefp	50491	49822	92221	91421	50801	50651	213771
canldr	32641	32372	65211	64811	33261	33141	156611
iirflt	29995	29845	56995	54845	34445	32865	127605
pntrch	23887	22519	43137	42447	22627	22627	109257
puwmod	48782	48184	97072	96702	48642	48592	239752
rspeed	10913	10893	21393	21213	12103	11933	51713
tbody	12533	12503	25493	22383	19923	13573	58813

in Table 6 assuming these policies. The ratios between the worst-case execution time estimates for the different policies are given in Table 7.

To recap, analysis for write merge can be obtained by assuming (i) the write buffer is full of junk at the start of each job, and (ii) the WBRPD equates to a full flush of the buffer. While this analysis is sound for a write buffer of depth 1, it is potentially optimistic due to domino effects for larger buffers. Nevertheless, given that the main focus of this paper is on the analysis and evaluation of the guaranteed performance of write-back caches, it is interesting to use this potentially optimistic analysis to make indicative comparisons with write-through caches with larger write buffers.

In Figs. 14, 15, 16, 17, 18, and 19, we examine the performance of write-through caches with a write buffer of depths 0 (= no write buffer) and 1 with sound analysis, as well as depths 2 and 4 with potentially optimistic analysis; the latter indicated by

**Table 7** Ratios of execution time estimates for the Mälardalen and EEMBC benchmarks used for evaluation

Name	$\frac{C^{wb-1}}{C^{wb-0}}$	$\frac{C^{wt-0}}{C^{wb-0}}$	$\frac{C^{wt-1}}{C^{wb-0}}$	$\frac{C^{wt-2}}{C^{wb-0}}$	$\frac{C^{wt-4}}{C^{wb-0}}$	$\frac{C^{nc}}{C^{wb-0}}$
cnt	1.00	1.44	1.05	1.04	1.03	2.63
compress	1.00	1.75	1.58	1.53	1.39	4.07
countneg	1.00	1.58	1.56	1.33	1.02	3.16
crc	.99	1.94	1.15	1.15	1.01	3.96
expint	1.00	1.64	1.35	1.02	1.01	3.35
fdct	1.00	2.13	1.44	1.29	1.17	4.87
fir	.99	2.28	1.64	1.06	1.02	5.24
jfdctint	1.00	1.91	1.45	1.26	1.19	4.03
loop3	1.00	2.02	1.89	1.01	1.01	4.08
ludcmp	.99	1.58	1.31	1.15	1.07	3.94
minver	1.00	1.61	1.22	1.17	1.05	2.88
ns	.99	1.37	1.00	1.00	1.00	3.59
nsichneu	.99	1.28	1.05	1.05	1.05	3.51
qurt	1.00	1.52	1.17	1.09	1.03	2.25
select	.99	1.89	1.35	1.25	1.10	3.37
sqrt	1.00	1.46	1.25	1.12	1.01	2.13
statemate	.99	3.02	1.87	1.59	1.49	9.00
a2time	.98	1.81	1.80	.99	.99	4.25
aifirf	.92	1.93	1.72	.92	.92	4.04
basefp	.98	1.82	1.81	1.00	1.00	4.23
canrdr	.99	1.99	1.98	1.01	1.01	4.79
iirflt	.99	1.90	1.82	1.14	1.09	4.25
pntrch	.94	1.80	1.77	.94	.94	4.57
puwmod	.98	1.98	1.98	.99	.99	4.91
rspeed	.99	1.96	1.94	1.10	1.09	4.73
tblock	.99	2.03	1.78	1.58	1.08	4.69

dashed lines. These results are compared to sound analysis for write-back caches with write buffers of depths 0 (= no write buffer) and 1. The experimental configurations used are otherwise identical to those presented in Figs. 6, 7, 8, 10, 9, and 11. For write-back caches, the analysis used is the combined approach which is the most effective of all the methods presented in this paper. Note the graphs are best viewed online in colour.

We observe in Table 6 that the worst-case execution time estimates for the benchmarks used in the evaluation are substantially better with a write-back cache than with a write-through cache when no write buffers are employed. Adding a write buffer of depth 4 to the write-through cache appears to be sufficient to close the performance gap, relative to a write-back cache with no write buffer. This observation is born out by the schedulability analysis results. Figures 14, 15, 16, 17, 18, and 19, show that while adding a write buffer improves the performance of the write-through cache configuration, the guaranteed performance with a buffer of depth 1 is well below that of a

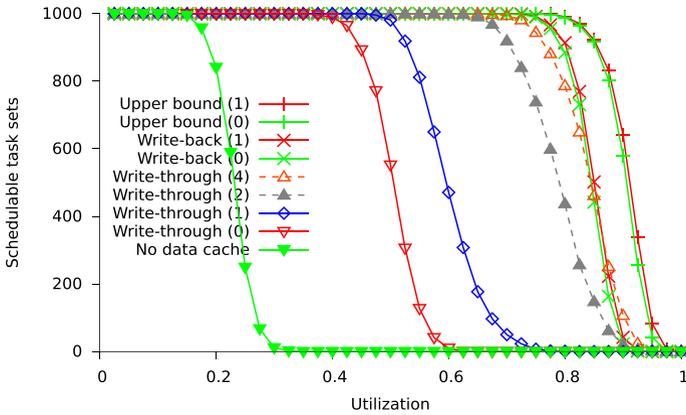


Fig. 14 Number of schedulable task sets (FPPS)

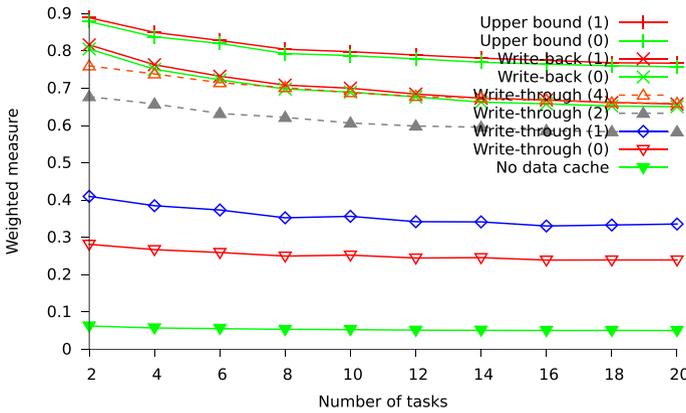


Fig. 15 Weighted schedulability versus number of tasks (FPPS)

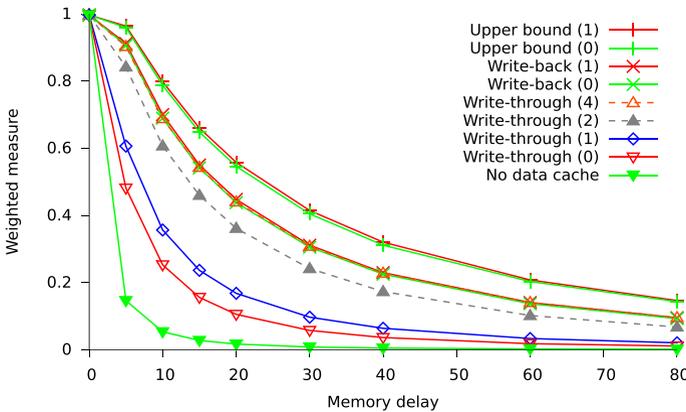


Fig. 16 Weighted schedulability versus memory latency (FPPS)

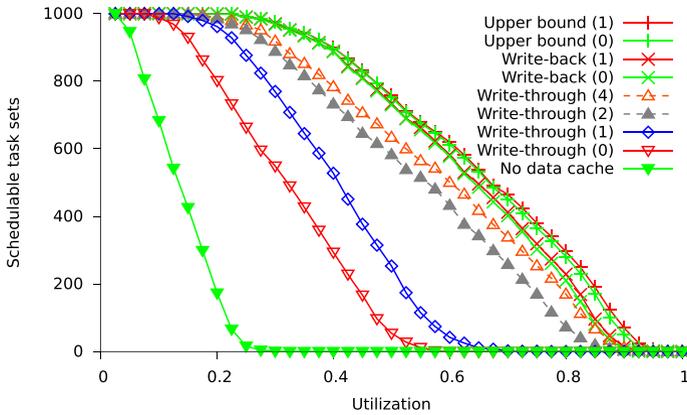


Fig. 17 Number of schedulable task sets (FPNS)

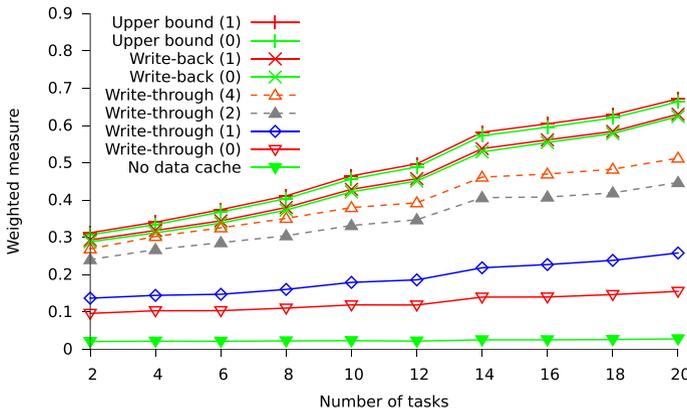


Fig. 18 Weighted schedulability versus number of tasks (FPNS)

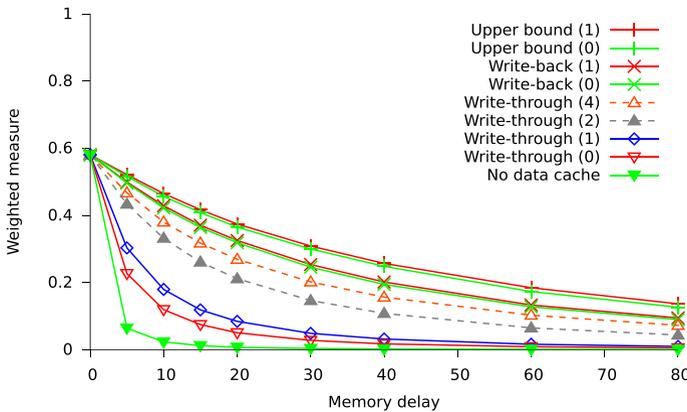


Fig. 19 Weighted schedulability versus memory latency (FPNS)

write-back cache. This gap is reduced with a write buffer of depth 2 and closed with a write buffer of depth of 4; however, we caution that the results given for buffer depths of 2 and 4 (dashed lines) are potentially optimistic due to domino effects, which have been ignored in computing those *indicative-only* results.

By contrast, the addition of a write buffer of depth 1 has little effect on the results for a write-back cache. Here, the ratio of worst-case execution time estimates with/without a write buffer vary from 0.92 to 1.0 for the different benchmarks (see Table 7). This is because the write-back cache is already effective at hiding the write latency.

Note, two upper bounds are shown on the graphs, these are for a write-back cache with/without a write buffer, but ignoring the overheads of all write backs except for job internal ones. Thus the difference between these lines reflects the difference in the worst-case execution time estimates shown in Table 6.

Recall that our simple analysis of write buffers assumes that the write buffer is full of junk that has to be written back at the start of each job. This assumption of a dirty write buffer has a negligible impact on task execution times, since the differences in the number of write-backs between an initially dirty buffer and an initially empty buffer are bounded by the buffer size. We validated this assumption by repeating our experiments comparing an empty and a dirty write buffer, both of size 4. There was no observable difference in the results, hence we only show the results for a dirty write buffer in the graphs.

## 9 WCET, ECB, DCB, and FDCB analysis for write-back caches

This paper focusses on the integration, into response-time analysis, of the overheads due to write backs. As a proof-of-concept, to evaluate the response-time analysis techniques, we obtained the WCETs and the sets of DCBs and ECBs from a trace of accesses obtained for each of the programs in the Mälardalen (Gustafsson et al. 2010) and EEMBC (EEMBC 2016) benchmark suites. Due to the simplicity of the benchmarks, and the provision of input data, this was possible for both single-path and multi-path examples. The code for each benchmark was first compiled using the GCC ARM cross-compiler, and included statically-linked library calls. A single trace for each benchmark was then generated, using the gem5 instruction set simulator (Binkert 2011), using the input data specified as part of the benchmark. For each benchmark, the trace was used to obtain the sets of UCBs, ECBs, DCBs and FDCBs via cache simulation. These values and the WCET bounds obtained were therefore exact. Obtaining the sets of values in this way enables a like-for-like comparison between the different analyses for write-back, write-through, and no cache. More complex programs would require the use of static analysis techniques to generate these sets.

Write-back caches are a popular choice in embedded microprocessors as they promise higher performance than write-through caches. So far, however, their use in hard real-time systems has been prohibited by the lack of adequate worst-case execution time (WCET) analysis support. The development and implementation of such techniques is the subject of ongoing work.

Blaß et al. (2017) introduced an effective method of statically analysing write-back caches.<sup>3</sup> Previous work in this area looked at the problem from an *eviction-focussed* perspective, analysing if a cache miss could result in a write back. Blaß et al. (2017) complemented this with analysis from a *store-focussed* perspective, considering if a write could dirty a clean cache line and thus result in a write back occurring later on. Their evaluation showed that large improvements in precision can be obtained by combining analyses from the two different perspectives. Blaß et al. (2017) thus showed that for most of the Malardalen and Scade benchmarks considered, the WCET bounds are smaller for write-back than for write-through caches. This was the case for 34 out of 36 benchmarks studied, and 32 out of 34 when a write buffer was added to the write-through cache. The ratio of WCETs for write-back versus write-through caches varied from 58 to 114%. Note, since these figures assume an LRU replacement policy, they are not directly comparable with the figures in this paper, which assume a direct-mapped cache. From the work of Blaß et al. (2017), we conclude that write-back caches provide a substantial opportunity to improve performance when static analysis is used to obtain WCETs.

We now sketch how to derive the set of evicting cache blocks (ECB), dirty cache blocks (DCB), and final dirty cache blocks (FDCB) using static analysis techniques. In all cases, we are interested in conservative approximations in the sense that the sets may only be over- but never be under-approximated. For the set of ECBs, it is sufficient to accumulate all cache lines accessed across all paths during program execution, and for the set of DCBs, it is sufficient to accumulate all cache lines written to during program execution. This can be accomplished by a simple data-flow analysis. In the case of data caches, a challenge is to precisely determine which cache lines may be accessed at a particular program point. Since by construction, the set of FDCBs is a subset of the set of DCBs, a DCB analysis provides a sound but pessimistic approximation of the set of FDCBs. A more precise approximation can be obtained using *may-cache analysis* (Ferdinand and Wilhelm 1999). This computes for each program point an over-approximation of the cache contents, i.e., the memory blocks that may be cached in each cache set. May-cache analysis can be extended to keep track of the dirty state of each cache line, as shown by Ferdinand and Wilhelm (1999), again in a conservative fashion: each potentially dirty cache line is considered to be dirty. The set of FDCBs is then given by the set of dirty cache lines in the may cache at the final program point.

We assume that the software programs being analysed are designed for use in critical real-time systems. Thus they make minimal use of pointers, do not include recursion, and statically allocate all data structures. (This is inline with design principles set out in ISO26262 that must be complied with: no dynamic objects, no recursion, limited use of pointers, single entry and single exit point for functions, no hidden data flow or control flow). Further, we assume that the operating system uses a separate fixed stack location for each task, thus stack variables created in every function calling context can have their addresses fully resolved at compilation / linking time, along with all global variables and other data structures. Difficulties remain in resolving precisely which memory locations are accessed inside loops; however, loop unrolling provides a

<sup>3</sup> Note publication of (Blaß et al. 2017) followed 6 months after this paper was submitted.

potential solution to this problem. Nevertheless, we recognise that there are a number of sources of pessimism that can potentially impact the accuracy of a static cache analysis leading to imprecision in the sets of DCBs and FDCBs, examples include accesses to locations that are dependent on input data. Refining the representation of ECBs, DCBs, and FDCBs to capture this uncertainty while avoiding undue pessimism in the analysis is the subject of our ongoing research.

## 10 Conclusions and future work

In this paper, we showed how to account for the costs of using a write-back cache in response-time analysis for fixed-priority preemptive and fixed-priority non-preemptive scheduling. Thus we introduced, for the first time, an effective method of bounding these overheads and therefore guaranteeing schedulability in fixed-priority systems using write-back caches. We introduced the concepts of Dirty Cache Blocks (DCBs), and Final Dirty Cache Blocks (FDCBs) and classified the different types of write back which can occur due to a task's internal behaviour, carry-in effects from previously executing tasks, and preemption effects. For each scheduling paradigm, we derived four approaches to analysing the worst-case number of write backs that can occur within the response time of a task. We showed the dominance relationships that hold between these different approaches and formed state-of-the-art combined approaches for both fixed-priority preemptive and non-preemptive scheduling based on them.

Our evaluation using data from the Mälardalen and EEMBC benchmark suites showed that the approaches derived are highly effective, resulting in guaranteed performance with a write-back cache which significantly exceeds that obtained using a write-through cache. These results show that the commercial preference for write-back caches due to their better average case performance extends to their analysable real-time performance. This conclusion is backed up by recent research (Blaß et al. 2017) providing an effective WCET analysis for systems with write-back caches.

We also extended our work to consider *write buffers* which can be used to improve efficiency, particularly with write-through caches. Here we showed that with write-through caches, large write buffers are necessary to achieve comparable performance to write-back caches. Further, compositional analysis for write-buffers of size  $> 1$  may incur timing anomalies (domino effects) and result in unsafe bounds.

This paper represents an important first step in the integration of analysis for write-back caches into schedulability analysis. It necessarily makes some simplifications, most notable of which is the focus on direct-mapped caches. We intend to extend our work in this area to include the analysis of set-associative caches, with the least-recently-used (LRU) policy, and a resilience-like (Altmeyer et al. 2010) notion for dirty cache blocks.

Other avenues we aim to explore include; the effect of bypassing the cache on stores where there is no re-use, i.e. *streaming* stores; the effect of flushing the cache (forcing write backs) at certain points in the code to improve predictability, for example by forcing write backs at job termination; and the effect of memory layout on performance, similar to what has previously been done to reduce cache-related preemption delays (Lunniss et al. 2012).

Our proof-of-concept evaluation relies on measurements and simulation to determine the WCET and the sets of ECBs, UCBs, DCBs, and FDCBs for each benchmark task. For the analysis techniques introduced in this paper to be used on real systems, this information needs to be provided via static analysis. Subsequent research (Blaß et al. 2017) (published after this paper was submitted) has now provided an effective method of WCET analysis for write-back caches with an LRU replacement policy. In future we aim to build upon this analysis. We note that uncertainty / imprecision in the sets of ECBs, UCBs, DCBs, and FDCBs challenges the precision of analysis for both write-through and write-back caches. This is an area that requires further study, and is the subject of our ongoing work. In this paper, we have shown that pessimistic analysis (write-back flush) which assumes the entire cache is written back at the start of each task still provides a substantial improvement over analysis for write through caches, hence even an imprecise static analysis can reasonably be expected to provide considerable improvements over the guaranteed performance obtained with a write through cache. Finally, another interesting area to explore is the analysis of multi-level write-back caches (Zhang et al. 2017).

**Acknowledgements** This work was supported by the the COST Action IC1202 TACLe, the UK EPSRC Projects MCC (EP/K011626/1) and MCCps (EP/P003664/1), the INRIA International Chair program, by the Deutsche Forschungsgemeinschaft (DFG) as part of the Project PEP, and by the NWO Veni Project 'The time is now: Timing Verification for Safety-Critical Multi-Cores'. EPSRC Research Data Management: No new primary data was created during this study. Collaboration was sparked by the Dagstuhl Seminar on Mixed Criticality Systems <http://www.dagstuhl.de/15121>. Finally, we would like to thank Benjamin Lesage for his comments on an earlier draft.

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## Appendix: Related work on data cache analysis

Data cache analysis is more difficult than instruction cache analysis. There are two reasons for this. Firstly, absolute data addresses are more difficult to obtain via static analysis than instruction addresses. *Data flow analysis* is required, and the program is restricted to having no dynamic data structures. In some cases, it may be impossible to determine the data address due to dependence on input values, or it may be determined only within a certain range. Secondly, the address accessed by a particular read/write instruction in the code may change during execution of the program, for example the elements of an array that are accessed sequentially. The problem of determining stack frames can be resolved if each function call is considered as a separate instance, and there are no recursive calls. Access to scalar global variables can also be resolved.

Early work on static analysis for data caches by Kim et al. (1996) categorized read/write instructions as either *static* or *dynamic*, the latter meaning that the address may change. For dynamic accesses, they assumed a cost of two cache misses, equating to the access itself and the eviction of another block that might otherwise have resulted in a cache hit. For array accesses in a loop, Kim et al. (1996) used a method based

on the pigeonhole principle. For each loop, they compute (i) the maximum number of accesses from each instruction, and (ii) the maximum number of distinct memory locations accessed. Subtracting (ii) from (i) gives the number of data cache hits for the loop, assuming there are no other conflicting accesses, and the data accesses within a loop fit in the cache. This method assumes that the size of each data access is the same size as a cache block.

Li et al. (1996) proposed a data cache analysis divided into two stages *data flow analysis* which determines the absolute addresses of the read/write instructions (single addresses, a range of possible values, or a set of addresses accessed sequentially), and *data cache conflict analysis* which involves building Cache Conflict Graphs (CCG) for direct mapped caches, or Cache State Transition Graphs (CSTG) for set associative caches, deriving constraints and then solving an ILP. Li et al. show how ranges of possible address values can be modelled in the constraints; however, a separate constraint is needed for each possible access in a given range, which produces a large number of constraints for large arrays.

White et al. (1997) presented an approach that uses data flow analysis within a compiler to determine a range of addresses for each access. Categorisation of the accesses is then done via a static cache simulator, providing Always Hit, Always Miss, First Hit, and First Miss classification. This information is then used as part of a pipeline WCET analysis to determine the WCET for each loop and function in the program. Experiments for a direct mapped data cache of 16 lines showed that the method is effective, with accurate results for many of the programs studied. The method was also extended to set associative caches.

Ferdinand and Wilhelm (1998) presented a *persistence* analysis for LRU caches, indicating which memory blocks are guaranteed to persist in the cache and therefore result in cache hits on their subsequent access. The persistence analysis is extended to cover the case where memory addresses are not fully resolved, and thus may take a range of values. Ferdinand and Wilhelm (1998) note that *Must* and *May* analysis can be used to determine the data cache behaviour if the addresses of access can be statically determined. They note that with array accesses, although in the general case it may not be possible to resolve the behaviour on each access, in many programs, the way in which array elements are accessed is very simple (affine in the loop variables) and a system of linear equations can be constructed that allows the cache behaviour to be determined. Solving these equations exactly can however be computationally very expensive.

Ghosh et al. (1999) introduced a *Cache Miss Equation (CME)* framework. This method generates a set of Diophantine equations<sup>4</sup> that describe the behaviour of the data cache for code in loops. Solving these equations is computationally complex; however, approximations and constrained methods can be used to reduce this complexity. The CME approach produces an estimate of the number of misses in nested loops. There are a number of restrictions on the code that can be analysed in this way: loops must be rectangular, and strictly nested, expressions for array indices and loop bounds must be affine combinations of loop variables known at compile time. Fur-

---

<sup>4</sup> Equation in two or more unknown values where only integer solutions are sought.

ther, no input dependent conditionals are permitted. Ramaprasad and Mueller (2005) extended the CME approach to handle more general loops via *forced loop fusion*, to handle data-dependent conditionals, and to accurately handle scalar variables. Their method produces exact data cache reference patterns, giving the position of misses in a sequence of references.

Lundqvist and Stenstrom (1999), proposed a method of improving precision in the analysis of data caches. Their method involves placing unpredictable data structures in uncached regions of memory. First, during WCET analysis, memory accesses are classified as *predictable* or *unpredictable* depending on whether the address referenced is known during the analysis. Data structures with unpredictable accesses are marked as unpredictable and subsequently allocated to memory areas that are not cached. For example, the linker can be used to place individual data structures in cached/uncached regions of memory. This method leaves only predictable accesses to the data cache, improving analysis precision. Uncached data structures incur a miss penalty on each access; however, that is lower than the potential double miss penalty that has to be conservatively assumed if the data structure were placed in cached memory.

Lundqvist and Stenstrom (1999) categorized accesses based on the storage type: global, stack or heap, and the access type: scalar, regular array, irregular but input independent, and input dependent. They observed that most accesses are in fact predictable, with only input-dependent accesses being always unpredictable. Accesses via the heap could be made predictable if the allocation policy always resulted in the same memory address for a given object. (We note that in many hard real-time systems only static memory allocation is permitted).

Chatterjee et al. (2001) developed an exact analysis of the cache behaviour of nested loops based on the use of Presburger formulas. This method classifies misses as either *interior misses* that do not depend on the cache state at the start of a program fragment (e.g. loop nest), and *boundary misses* which may be a miss or a hit depending on the cache state when the fragment starts to execute. This classification has the useful property that it is composable. The method determines the cache state at the start of each fragment and from that, the exact number of misses. The method handles imperfect nests, a variety of array layouts, and a modest level of associativity (examples are given for an associativity of 2). The computational complexity of the method, which relates to the static structure of the loop nests not their dynamic iteration count, is however very high. Quantifier elimination in the Presburger formulas is super-exponential with worst-case upper and lower bounds that are  $O(2^{2^n})$ . Nevertheless, the method is shown to be effective for a number of examples of loop nesting, with computation times from less than 1 second to 4 minutes. The method was validated against simulation and found to determine precisely the number of misses. The authors suggest that the method could be used in conjunction with cache simulation, allowing a simulator to rapidly skip over loop nests which would otherwise consume much of the running time. They note; however, that the handling of associativity is incomplete and does not scale.

Staschulat and Ernst (2006) investigated the problem of data cache analysis where there are dependencies on inputs. They identify Single Data Sequences (SDS) where the memory blocks accessed and the control flow are both independent of inputs. The cache behaviour for SDS can then be determined by a simple cache simulation. For

data accesses that are not in SDS, persistence analysis is used. This does not however capture array access patterns.

Sen and Srikant (2007) presented an approach that combines automatic executable analysis to determine the addresses accessed and a Must analysis for determining cache behaviour, both using Abstract Interpretation. The overall problem is divided into four sub-problems: address analysis, cache analysis, access sequencing, and worst-case path analysis. The latter is solved using an ILP formulation. Sen and Srikant (2007) use Circular Linear Progressions (CLPs) to provide a strided linear approximation of the discrete set of memory addresses that may be accessed by a particular instruction. CLPs enable more precise evaluation of the sequence of locations accessed. They are used by the cache analysis to determine bounds on the age of each block in the cache, and hence if an access should be classified as always hit or not classified. The access sequencing problem is handled via the partial unrolling of loops using an expansion mode (virtual unrolling) and a summary mode. Experimental results were obtained for an ARM7TDMI assuming LRU cache. These showed improvements in precision with more loop expansion. The virtual loop unrolling is however expensive in terms of analysis time.

Huynh et al. (2011) introduced a method which takes into account the scope in which certain memory accesses may occur. Instructions that access memory may access different memory locations in different temporal scopes. The method of Huynh et al. (2011) captures the temporal scope (i.e. loop iterations) where a particular memory block is accessed for a given read/write instruction. These temporal scopes are then used to provide more precise abstract cache state modelling. Persistence analysis is extended to determine, on a per scope basis, if a memory block persists in the cache. Further memory blocks accessed in mutually exclusive scopes do not conflict with each other. The authors showed that their method fully captures the temporal locality of array traversal made in row-major order (as the array is laid out in memory), achieving much tighter results than persistence analysis without temporal scope information. Huynh et al. (2011) also fixed a problem in the original persistence analysis.

Herter et al. (2011) introduced CAMA, a cache-aware dynamic memory allocator. The use of CAMA enables static analysis of the data-cache behaviour of programs using dynamic memory allocation.

Wegener (2012) described a method of determining the *same block relation* indicating whether two memory accesses target the same block and thus may result in a cache hit. The method focusses on establishing same block relations for array accesses within a loop. It uses loop peeling and loop unrolling to provide more information to the analysis. Results for the Malardalen benchmarks show that this relational analysis increases precision for most of the programs, with a few showing no improvement due to compiler optimisations splitting loops into nested ones or due to bit operations destroying the relational information.

Hahn and Grund (2012) introduced *relational cache analysis*. This approach does not require absolute address information, but rather reasons based on the relative addresses of different memory accesses. This enables cache hit predictions for some accesses that are dependent on unknown static pointers, or input values. Relational cache analysis uses *symbolic names* to abstract away from absolute addresses. A *congruence analysis* is used to reason about the relations between pairs of symbolic

names. *Relational Cache Analysis* is then used to classify memory references using the relational information about the symbolic names. (Cache information, for example age bounds are attached to the symbolic names). The congruence analysis establishes relations between symbolic names such as: same block, same cache set but different block, and different cache set. It also establishes approximations that are useful such as: same block or different set (which excludes evictions). Congruence information also includes interval analysis, global value numbering, octagon analysis, and value set analysis. In their evaluation, Hahn and Grund (2012) show how the relational cache analysis can provide significantly improved results for examples with stack-relative accesses, array accesses within a loop iteration, and input-dependent accesses. In fact the analysis is claimed to always dominate the abstract interpretation method of Ferdinand and Wilhelm (1998), since it is at least as precise. Hahn and Grund (2012) note that prior works that make use of address information as a description of memory blocks in abstract cache states cannot model imprecisely determined addresses. Also, there is excessive information loss, for example when  $m$  accesses occur to the same, but unknown memory block, cached blocks must be aged by  $m$ . Further, to regain precision, prior analyses have to be highly context dependent increasing analysis runtime.

Schoeberl et al. (2013) proposed splitting the data cache into different data areas, i.e. different small data caches. This ensures that accesses to unknown addresses, due for example to heap allocated data, do not pollute information about the cache for other simple, easy to predict areas e.g. for static data. The different data caches are optimized for their data area. A cache for the stack and constants is direct mapped, while the stack for heap allocated data has high associativity. For heap allocated data, Schoeberl et al. (2013) present a scope-based persistence analysis.

## References

- Altmeyer S (2013) Analysis of preemptively scheduled hard real-time systems. epubli GmbH, <https://books.google.co.uk/books?id=GdoZAY-Jv34C>
- Altmeyer S, Maiza C (2011) Cache-related preemption delay via useful cache blocks: survey and redefinition. *J Syst Archit* 57:707–719
- Altmeyer S, Maiza C, Reineke J (2010) Resilience analysis: tightening the CRPD bound for set-associative caches. In: Proceedings of the Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES), pp 153–162
- Altmeyer S, Davis R, Maiza C (2011) Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp 261–271
- Altmeyer S, Davis R, Maiza C (2012) Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Syst*. 48(5):499–526
- Altmeyer S, Douma R, Lunniss W, Davis R (2014) Evaluation of cache partitioning for hard real-time systems. In: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), pp 15–26
- Altmeyer S, Davis RI, Indrusiak L, Maiza C, Nelis V, Reineke J (2015) A generic and compositional framework for multicore response time analysis. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems, ACM, New York, NY, USA, RTNS '15, pp 129–138, <https://doi.org/10.1145/2834848.2834862>
- Altmeyer S, Douma R, Lunniss W, Davis R (2016) On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Syst*. <https://doi.org/10.1007/s11241-015-9246-8>
- Audley N, Burns A, Richardson M, Tindell K, Wellings A (1993) Applying new scheduling theory to static priority pre-emptive scheduling. *Softw Eng J* 8(5):285–292

- Baruah S, Burns A (2006) Sustainable scheduling analysis. In: IEEE Real-Time Systems Symposium (RTSS), pp 159–168, <https://doi.org/10.1109/RTSS.2006.47>
- Bastoni A et al (2010) Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of the workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT), pp 33–44
- Bini E, Buttazzo G (2005) Measuring the performance of schedulability tests. *Real-Time Syst* 30(1):129–154
- Binkert N et al (2011) The gem5 simulator. *SIGARCH Comput Archit News* 39(2):1–7. <https://doi.org/10.1145/2024716.2024718>
- Blaß T, Hahn S, Reineke J (2017) Write-back caches in WCET analysis. In: Bertogna M (ed) 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol 76, pp 26:1–26:22, <https://doi.org/10.4230/LIPIcs.ECRTS.2017.26>, <http://drops.dagstuhl.de/opus/volltexte/2017/7158>
- Bril R, Lukkien J, Verhaegh W (2009) Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Syst* 42(1):63–119
- Bril R, Altmeyer S, van den Heuvel M, Davis R, Behnam M (2014) Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp 161–172
- Burguière C, Reineke J, Altmeyer S (2009) Cache-related preemption delay computation for set-associative caches—pitfalls and solutions. In: Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET), pp 1–11
- Burns A (1994) Preemptive priority based scheduling: an appropriate engineering approach. In: Son S (ed) *Advances in real-time systems*. Prentice-Hall, Upper Saddle River, pp 225–248
- Busquets-Mataix JV, Serrano JJ, Ors R, Gil P, Wellings A (1996) Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In: Proceedings of the IEEE Real-Time Embedded Technology and Applications (RTAS), pp 204–212
- Chatterjee S, Parker E, Hanlon PJ, Lebeck AR (2001) Exact analysis of the cache behavior of nested loops. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '01, pp 286–297, <https://doi.org/10.1145/378795.378859>
- Davis R, Burns A, Bril R, Lukkien J (2007) Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst* 35(3):239–272
- Davis R, Altmeyer S, Reineke J (2016) Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In: Proceedings of the 24th International Conference on Real-Time and Network Systems (RTNS)
- EEMBC (2016) EEMBC Autobench industry-standard benchmarks for embedded systems. [http://www.eembc.org/benchmark/automotive\\_sl.php](http://www.eembc.org/benchmark/automotive_sl.php). Accessed 29 April 2016
- Ferdinand C, Wilhelm R (1998) On predicting data cache behavior for real-time systems. In: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, Springer-Verlag, London, UK, UK, LCTES '98, pp 16–30, <http://dl.acm.org/citation.cfm?id=646905.710485>
- Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst* 17(2–3):131–181. <https://doi.org/10.1023/A:1008186323068>
- Ghosh S, Martonosi M, Malik S (1999) Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans Program Lang Syst* 21(4):703–746. <https://doi.org/10.1145/325478.325479>
- Gustafsson J, et al (2010) The Mälardalen WCET benchmarks—past, present and future. In: Proceedings of the International workshop on Worst-Case Execution Time Analysis (WCET), pp 137–147
- Hahn S, Grund D (2012) Relational cache analysis for static timing analysis. In: Euromicro Conference on Real-Time Systems (ECRTS), pp 102–111, <https://doi.org/10.1109/ECRTS.2012.14>
- Hahn S, Reineke J, Wilhelm R (2013) Towards compositionality in execution time analysis – definition and challenges. In: Proceedings of the International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)
- Herter J, Backes P, Hauptenthal F, Reineke J (2011) CAMA: a predictable cache-aware memory allocator. In: 23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011, pp 23–32, <https://doi.org/10.1109/ECRTS.2011.11>

- Huynh BK, Ju L, Roychoudhury A (2011) Scope-aware data cache analysis for wcet estimation. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 203–212, <https://doi.org/10.1109/RTAS.2011.27>
- Joseph M, Pandya P (1986) Finding response times in a real-time system. *Comput J* 29(5):390–395
- Katcher D, Arakawa H, Strosnider J (1993) Engineering and analysis of fixed priority schedulers. *IEEE Trans Softw Eng* 19:920–934
- Kim SK, Min S, Ha R (1996) Efficient worst case timing analysis of data caching. In: Proceedings of Real-Time Technology and Applications Symposium. IEEE, pp 230–240, <https://doi.org/10.1109/RTTAS.1996.509540>
- Lee CG et al (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans Comput* 47(6):700–713
- Li YTS, Malik S, Wolfe A (1996) Cache modeling for real-time software: beyond direct mapped instruction caches. In: Proceedings of IEEE Real-Time Systems Symposium (RTSS), pp 254–263
- Lundqvist T, Stenstrom P (1999) A method to improve the estimated worst-case performance of data caching. In: RTCSA '99. Sixth International Conference on Real-Time Computing Systems and Applications, pp 255–262, <https://doi.org/10.1109/RTCSA.1999.811244>
- Lunniss W, Altmeyer S, Davis R (2012) Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In: Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS), pp 161–170, <https://doi.org/10.1145/2392987.2393008>.
- Lunniss W, Davis R, Maiza C, Altmeyer S (2013) Integrating cache related pre-emption delay analysis into edf scheduling. In: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
- Lunniss W, Altmeyer S, Davis RI (2014a) A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays. *Leibniz Trans Embed Syst* 1(1):01–01:24, <https://doi.org/10.4230/LITES-v001-i001-a001>, <http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v001-i001-a001>
- Lunniss W, Altmeyer S, Lipari G, Davis RI (2014b) Accounting for cache related pre-emption delays in hierarchical scheduling. In: Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS), ACM, New York, NY, USA, pp 183:183–183:192, <https://doi.org/10.1145/2659787.2659797>
- Lunniss W, Altmeyer S, Lipari G, Davis RI (2016) Cache related pre-emption delays in hierarchical scheduling. *Real-Time Syst* 52(2):201–238. <https://doi.org/10.1007/s11241-015-9228-x>
- Lv M, Guan N, Reineke J, Wilhelm R, Yi W (2016) A survey on static cache analysis for real-time systems. *Leibniz Trans Embed Syst* 3(1):05–1–05:48, <https://doi.org/10.4230/LITES-v003-i001-a005>, <http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v003-i001-a005>
- Ramaprasad H, Mueller F (2005) Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In: 11th IEEE Real Time and Embedded Technology and Applications Symposium, pp 148–157, <https://doi.org/10.1109/RTAS.2005.12>
- Saksena M, Wang Y (2000) Scalable real-time system design using preemption thresholds. In: Proceeding of the IEEE Real-Time Systems Symposium (RTSS), pp 25–34
- Schneider J (2000) Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In: The 21st IEEE, Proceedings of Real-Time Systems Symposium. pp 195–204, <https://doi.org/10.1109/REAL.2000.896009>
- Schoeberl M, Huber B, Puffitsch W (2013) Data cache organization for accurate timing analysis. *Real-Time Syst* 49(1):1–28. <https://doi.org/10.1007/s11241-012-9159-8>
- Sen R, Srikant YN (2007) Wcet estimation for executables in the presence of data caches. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, ACM, New York, NY, USA, EMSOFT '07, pp 203–212, <https://doi.org/10.1145/1289927.1289960>
- Skadron K, Clark DW (1997) Design issues and tradeoffs for write buffers. In: Third International Symposium on High-Performance Computer Architecture, pp 144–155, <https://doi.org/10.1109/HPCA.1997.569650>
- Sondag T, Rajan H (2010) A more precise abstract domain for multi-level caches for tighter WCET analysis. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp 395–404, <https://doi.org/10.1109/RTSS.2010.8>
- Staschulat J, Ernst R (2006) Worst case timing analysis of input dependent data cache behavior. In: 18th Euromicro Conference on Real-Time Systems (ECRTS'06), pp 210–236, <https://doi.org/10.1109/ECRTS.2006.33>

- Staschulat J, Schliecker S, Ernst R (2005) Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), pp 41–48
- Tan Y, Mooney VJ (2007) Timing analysis for preemptive multi-tasking real-time systems with caches. *ACM Trans Embed Comput Syst* 6(1):1–26
- Wang C, Gu Z, Zeng H (2015) Integration of cache partitioning and preemption threshold scheduling to improve schedulability of hard real-time systems. In: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), pp 69–79
- Wang Y, Saksena M (1999) Scheduling fixed-priority tasks with preemption threshold. In: Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA), pp 328–335
- Wegener S (2012) Computing same block relations for relational cache analysis. In: 12th International Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, OpenAccess Series in Informatics (OASICs), vol 23, pp 25–37, <http://dx.doi.org/10.4230/OASICs.WCET.2012.25>, <http://drops.dagstuhl.de/opus/volltexte/2012/3554>
- White RT, Mueller F, Healy CA, Whalley DB, Harmon MG (1997) Timing analysis for data caches and set-associative caches. In: Third IEEE, Proceedings of Real-Time Technology and Applications Symposium, pp 192–202, <https://doi.org/10.1109/RTTAS.1997.601358>
- Zhang Z, Guo Z, Koutsoukos X (2017) Handling write backs in multi-level cache analysis for wcet estimation. In: Proceedings of the 25th International Conference on Real-Time and Network Systems (RTNS)



**Robert I. Davis** is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, UK, and an INRIA International Chair at INRIA, Paris, France. Robert received his DPhil in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include the following aspects of real-time systems: scheduling algorithms and analysis for single processor, multiprocessor and networked systems; analysis of cache related preemption delays, mixed criticality systems, and probabilistic hard real-time systems.



**Sebastian Altmeyer** is Assistant Professor (Universitair Docent) at the University of Amsterdam. He has received his Ph.D. in Computer Science in 2012 from Saarland University, Germany with a thesis on the analysis of preemptively scheduled hard real-time systems. From 2013 to 2015 he has been a postdoctoral researcher at the University of Amsterdam, and from 2015 to 2016 at the University of Luxembourg. In 2015, he has received an NWO Veni grant on the timing verification of real-time multicore systems, and he is program chair of the Euromicro Conference on Real-Time Systems (ECRTS) 2018. His research targets various aspects of the design, analysis and verification of hard real-time systems, with a particular interest in timing verification and multicore architectures.



**Jan Reineke** is a Professor at Saarland University, where he has been on the faculty since 2012, and where he obtained his Ph.D. degree in 2008. From 2009 to 2011 he was a Postdoctoral Scholar at UC Berkeley. His research interests include static analysis by abstract interpretation with applications to static timing analysis, shape analysis, topology analysis, and side-channel analysis. He is also interested in automatic methods to obtain faithful models of microarchitectures and in the design of timing-predictable microarchitectures for use in hard real-time systems. In 2012, he was selected as an Intel Early Career Faculty Honor Program awardee. He was the program committee co-chair of EMSOFT 2014, the International Conference on Embedded Software, a topic co-chair at DATE 2016, and program chair of WCET 2017.