Analysis of Write-back Caches under Fixed-priority Preemptive and Non-preemptive Scheduling

Robert I. Davis
University of York, UK
INRIA, France
rob.davis@york.ac.uk

Sebastian Altmeyer
University of Amsterdam,
Netherlands
altmeyer@uva.nl

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-sb.de

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.

1. INTRODUCTION

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.

RTNS '16, October 19 - 21, 2016, Brest, France
DOI: http://dx.doi.org/10.1145/2997465.2997476

Early work on accounting for scheduling overheads in fixed-priority preemptive systems by Katcher et al. [23] and Burns et al. [15] 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 section 2.1 of [6] for a detailed description). A number of methods have been developed for computing CRPD under fixed-priority preemptive scheduling. In 1996, Busquets et al. introduced the ECB-Only approach [16], which considers just the preempting task; while in 1998, Lee et al. developed the UCB-Only approach [24], which considers just the preempted task(s). Both the UCB-Union approach [30] developed by Tan and Mooney in 2007, and the ECB-Union approach [2] derived by Altmeyer et al. in 2011 consider both the preempted and preempting tasks. As does an alternative approach [29] developed by Staschulat et al. in 2005. These approaches were later superseded by multiset based methods (ECB-Union Multiset and UCB-Union Multiset) which dominate them [3].

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. In 2014, Altmeyer et al. [4], [5] 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 [32, 27] provide an alternative means of reducing CRPD by making certain groups of tasks non-preemptable with respect to each other. In 2014, Bril et al. [12] integrated CRPD into analysis for fixed-priority scheduling with preemption thresholds. Further work in this area by Wang et al. [31] in 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. [4].

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 section 2). In future, we aim to extend the techniques to set-associative caches and replacement policies such as LRU using the methodology given in [2].

Ferdinand and Wilhelm [19] 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 [28] implement a similar idea in the context of multi-level cache analysis, where the write-back behavior 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 [19, 28] 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, Malik, and Wolfe [25] proposed to capture both the software and the microarchitectural behavior 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.

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 blocks 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 [21], i.e. the timing contribution of cache misses and write backs can be analyzed separately from other architectural features such as the pipeline behavior.

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 in 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 these 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.

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. 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 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 jobs that are finished.

Consider Figure 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.

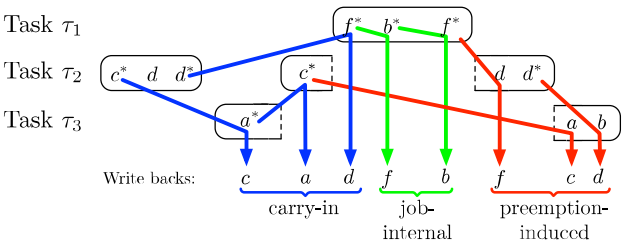


Figure 1: Example illustrating different kinds of write backs.

The first write to memory block a of task τ_3, causes the eviction of c, which was written to by a finished job of task τ_2, thus it causes a *finished-carry-in* write-back. On the other hand, the access to c in the second job of τ_2, causes an *lp-carry-in* write back of a. The first access to b within task τ_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 τ_2 causes a *preemption-induced* write back of f which was previously written to by task τ_1. Similarly, the reads of a and b in task τ_3 result in preemption-induced write backs of c and d, previously written to by task τ_2.

2.3 Characterizing a Task's Write Backs

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

An *Evicting Cache Block* (ECB) of task τ_i is a memory block that may be accessed by task τ_i. We denote the set of cache lines that evicting cache blocks of task τ_i map to by ECB_i. Note ECBs have previously been considered in the analysis of the cache-related preemption delays [3].

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

A *Final Dirty Cache Block* (FDCB) of task τ_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 τ_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 analysis 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 Γ comprises a static set of n tasks $\{\tau_1, \tau_2, ...\tau_n\}$. Each task has a unique priority, which without loss of generality is given by its index. Thus task τ_1 has the highest priority and task τ_n the lowest. Each task τ_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 τ_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 τ_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 τ_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 τ_i (including τ_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 τ_i.

3.2 Schedulability Analysis for FPPS

For task sets with constrained deadlines scheduled using FPPS, the exact response time of task τ_i may be computed according to the following recurrence relation [8], [22]:

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

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 τ_i under FPNS requires checking all of the jobs of task τ_i within the worst-case priority level-i busy period [13]. (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 τ_i. Just after that job starts to execute,

jobs of task τ_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) [18]. This schedulability test considers two scenarios. Either the worst-case response time for task τ_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 τ_i in the worst-case priority level-i busy period can be computed using the following recurrence relation:

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

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

$$R_{i,0}^{NP} = W_{i,0}^{NP} + C_i \quad (3)$$

Subsequent jobs of task τ_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 τ_i be indexed by values of $q = 0, 1, \ldots$, 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:

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

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:

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

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 τ_i, under FPNS. This upper bound may be compared with the task's deadline to determine schedulability.

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

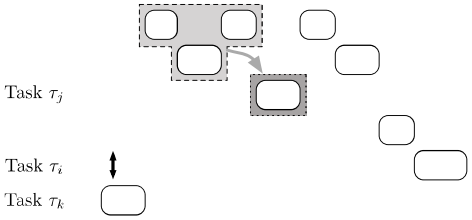
$$R_i^{NP} = W_i^{NP} + C_i \quad (7)$$

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).

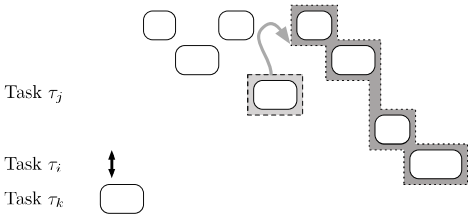
4. WRITE BACKS UNDER FPNS

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 Figure 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 Section 4.1. By contrast, in the second method we 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 Section 4.2.



(a) Write backs within the job itself.



(b) Write backs in subsequent jobs.

Figure 2: Carry-in write backs may be accounted for either (a) within the job of the task τ_i under analysis, or (b) in subsequent jobs of both higher (e.g. τ_j) and lower (e.g. τ_k) priority tasks.

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 it suffers[1]. Thus, assuming timing compositionality [21], the WCET of task τ_i, including the cost of write backs, is bounded by

$$C_i' = C_i + WBT \cdot |ECB_i| \quad (8)$$

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.

[1]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 [14].

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

Each final dirty cache line of a task with priority lower than or equal to that of task τ_i may result in at most one write back during τ_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 τ_i if the cache lines are accessed by (i.e. in the ECB_k) some task τ_k of priority i or higher. The term δ_i accounts for these write backs. Note that we exclude from δ_i cache lines that may be dirty due to higher priority tasks as such cache lines are accounted for by the $\gamma^{\text{wb}}_{i,j}$ term introduced next, thus:

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

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

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

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

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

$$R^{NP}_{i,WB} = W^{NP}_{i,WB} + (C_i + \gamma^{\text{wb}}_{i,i}) \quad (12)$$

In the $\gamma^{\text{wb}}_{n+1,b}$ term, $n+1$ denotes a priority that is lower than that of any task, thus $\gamma^{\text{wb}}_{n+1,b}$ accounts for all carry-in write backs that may occur during the execution of a blocking task τ_b due to cache lines left dirty by previous jobs of any task. In contrast, $\gamma^{\text{wb}}_{i,j}$ and $\gamma^{\text{wb}}_{i,i}$ need only cover write backs due to dirty cache lines from tasks of higher priority than τ_i, since all other write backs are accounted for in δ_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 $C_j + \gamma^{\text{wb}}_{i,j}$ terms in (11) and (12). From (10), it follows that $C_j + \gamma^{\text{wb}}_{x,j}$ cannot be greater than the value of C'_j used in (8) for any task τ_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 δ_i term. For an FDCB to contribute to δ_i, then from (9), that FDCB cannot be in $FDCB_k$ of any task τ_k with a priority higher than that of task τ_i. Also, it must be in the ECB_i of task τ_i or the ECB_k of some higher priority task τ_k. If it is in ECB_i and contributes to δ_i then from (10) it is not included in the $\gamma^{\text{wb}}_{i,i}$ term in (12), thus the inflated WCET C'_i in the ECB-Only approach covers both this contribution to δ_i and the $\gamma^{\text{wb}}_{i,i}$ term in (12). Similarly, if the FDCB is in ECB_j and contributes to δ_i then it is not included in the $\gamma^{\text{wb}}_{i,j}$ term in (11), thus the inflated WCET C'_j in the ECB-Only approach again covers both this contribution to δ and $\gamma^{\text{wb}}_{i,j}$. 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 $\gamma^{\text{wb}}_{i,j}$ to mean the cost of carry-in write backs that occur *within* the execution of a job of task τ_j, we can redefine $\gamma^{\text{wb}}_{i,j}$ to cover the write backs that occur in *subsequent jobs* due to dirty cache lines left by a job of task τ_j. This is achieved by assuming that all of these cache lines may be evicted by the subsequent jobs:

$$\gamma^{\text{wb}}_{i,j} = WBT \cdot |FDCB_j| \quad (13)$$

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

$$\delta = WBT \cdot \left| \bigcup_k FDCB_k \right| \quad (14)$$

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

$$R^{NP}_{i,WB} = W^{NP}_{i,WB} + C_i \quad (15)$$

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 τ_i's response time (i.e. by also considering the cache lines (ECB_k) accessed by each task τ_k of priority i or higher).

$$\gamma^{\text{wb}}_{i,j} = WBT \cdot \left| FDCB_j \cap \bigcup_{k \in hep(i)} ECB_k \right| \quad (16)$$

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

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

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

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

The ECB-Union approach strictly dominates the FDCB-Only approach. This can be seen by comparing the $\gamma^{\text{wb}}_{i,j}$ terms and the $\delta_{b,i}$ terms. Comparing the $\gamma^{\text{wb}}_{i,j}$ 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 τ_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. A worked example that illustrates these relationships is given in Appendix B of the technical report [17].

5. WRITE BACKS UNDER FPPS

Response-time analysis for FPPS has previously been extended to account for preemption-related cache misses [2], [3] by introducing a term $\gamma_{i,j}$ into the response-time equation for task τ_i as follows:

$$R^P_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R^P_i}{T_j} \right\rceil (C_j + \gamma_{i,j}) \quad (19)$$

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

$$R^P_i = \delta_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + \gamma^{\text{miss}}_{i,j} + \gamma^{\text{wb}}_{i,j}) \quad (20)$$

Here, δ_i is used to account for write backs due to cache lines that were already dirty on release of τ_i and are written back within its response time. Additional cache misses due to preemptions are captured by $\gamma^{\text{miss}}_{i,j}$. Any of the existing techniques, for example those introduced in [3] can be used to account for such misses. Finally, $\gamma^{\text{wb}}_{i,j}$ is used to account for carry-in and preemption-induced write backs of cache lines that were written to after τ_i's release.

We further subdivide $\gamma^{\text{wb}}_{i,j}$ into $\gamma^{\text{wb-lp}}_{i,j}$ and $\gamma^{\text{wb-fin}}_{i,j}$, such that $\gamma^{\text{wb}}_{i,j} = \gamma^{\text{wb-lp}}_{i,j} + \gamma^{\text{wb-fin}}_{i,j}$, where $\gamma^{\text{wb-lp}}_{i,j}$ accounts for lp-carry-in write backs and $\gamma^{\text{wb-fin}}_{i,j}$ accounts for finished-carry-in and preemption-induced write backs (see Section 2.2 for their definitions). In the following we introduce four different ways of computing $\gamma^{\text{wb-lp}}_{i,j}$. These combine with the analysis derived for δ_i and $\gamma^{\text{wb-fin}}_{i,j}$ 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 τ_i. Only tasks of lower priority than τ_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 τ_i's response time. As only τ_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 δ_i:

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

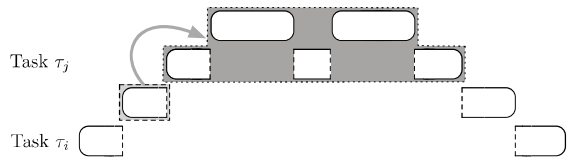
5.2 Lower-Priority Carry-in Write Backs

To bound lp-carry-in write backs ($\gamma^{\text{wb-lp}}_{i,j}$) due to preempted tasks, we identify two methods, both illustrated in Figure 3.

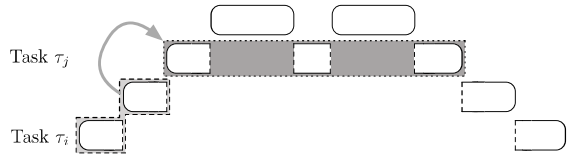
(a) the lp-carry-in write backs of dirty cache lines introduced by the job *immediately-preempted* by a job of τ_j that occur within the response time of τ_j, i.e. either executing τ_j or a higher-priority task.

(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 τ_j.

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



(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).

Figure 3: Methods of accounting for lp-carry-in write backs.

5.2.1 DCB-Only Approach

Using method (a), any task that could be active during the response time of task τ_i and has a lower priority than task τ_j (i.e. a task in the set $aff(i,j) = hep(i) \cap lp(j)$) could

be immediately preempted by task τ_j, thus we obtain the following upper bound on the cost of write backs $\gamma_{i,j}^{\text{wb-lp}}$ associated with jobs of task τ_j:

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

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 τ_j is active, i.e. due to execution of τ_j or a higher-priority task (see Figure 3) thus:

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

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 τ_j are upper bounded by the ECBs of τ_j:

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

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 Figure 3). 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 τ_i as these are already accounted for in δ_i.

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

5.3 Finished-carry-in Write Backs

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

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

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 τ_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.

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. A worked example that illustrates these relationships is given in Appendix B of the technical report [17].

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.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the different analyses introduced in Sections 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 [20] and the EEMBC Benchmark suite [1] (Section 7 explains how this was done). Table 1 shows the number of UCBs, ECBs, DCBs, and FDCBs for selected benchmarks, as well as 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}). 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 1 where the stand-alone WCETs were in the range $[7000 : 70000]$ 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

Table 1: Data from the Mälardalen and EEMBC benchmarks used for evaluation

Name	C^{wb}	C^{wt}	C^{wt}/C^{wb}	C^{nc}	C^{nc}/C^{wb}	$\|UCB^I\|$	$\|ECB^I\|$	$\|UCB^D\|$	$\|ECB^D\|$	$\|DCB\|$	$\|FDCB\|$
cnt	9325	13485	1.44	24565	2.63	12	82	21	68	28	28
compress	10673	18713	1.75	43443	4.07	21	71	53	103	60	60
countneg	36180	57250	1.58	114340	3.16	15	77	59	103	66	66
crc	68889	133909	1.94	272859	3.96	19	89	25	73	40	39
expint	9268	15208	1.64	31098	3.35	16	76	11	42	13	13
fdct	7883	16793	2.13	38423	4.87	52	144	15	48	19	19
fir	8328	18998	2.28	43668	5.24	22	83	17	57	17	16
jfdctint	9711	18621	1.91	39181	4.03	46	145	17	53	23	23
loop3	14189	28729	2.02	57929	4.08	7	309	9	42	12	12
ludcmp	10058	15948	1.58	39668	3.94	38	128	21	61	28	28
minver	18976	30616	1.61	54746	2.88	103	213	18	71	33	33
ns	27464	37674	1.37	98634	3.59	14	70	9	116	13	11
nsichneu	18988	24458	1.28	66808	3.51	345	494	52	95	54	53
qurt	10473	16003	1.52	23573	2.25	61	132	14	49	17	17
select	8981	17031	1.89	30331	3.37	47	124	10	49	16	16
sqrt	27667	40537	1.46	59117	2.13	51	102	11	48	16	16
statemate	64638	195778	3.02	581908	9.00	92	167	25	68	21	20
a2time	12655	22975	1.81	53815	4.25	16	122	8	100	69	67
aifirf	44898	86768	1.93	181698	4.04	25	141	33	188	161	54
basefp	50491	92221	1.82	213771	4.23	11	88	15	512	507	467
canrdr	32641	65211	1.99	156611	4.79	8	40	9	371	195	186
iirflt	29995	56995	1.90	127605	4.25	35	288	28	259	147	138
pntrch	23887	43137	1.80	109257	4.57	24	38	20	237	176	70
puwmod	48782	97072	1.98	239752	4.91	3	50	5	512	307	275
rspeed	10913	21393	1.96	51713	4.73	8	53	7	122	71	70
tblook	12533	25493	2.03	58813	4.69	12	115	14	125	71	71

the results for FPPS, while also providing task sets that can actually be scheduled using FPNS. We used the same set of benchmarks for FPPS and FPNS to facilitate direct comparison. Results using all of the benchmarks are shown in Appendix D of the technical report [17], along with a complete table of values.

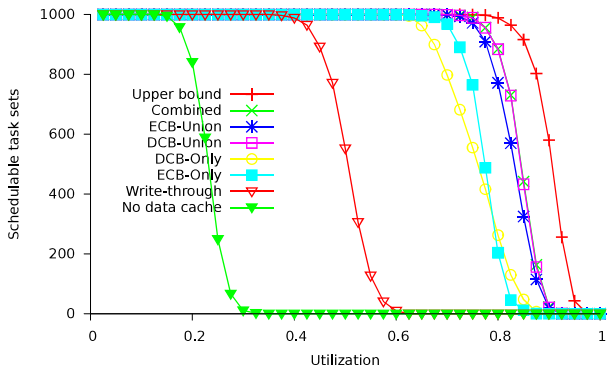


Figure 4: Number of schedulable task sets (FPPS).

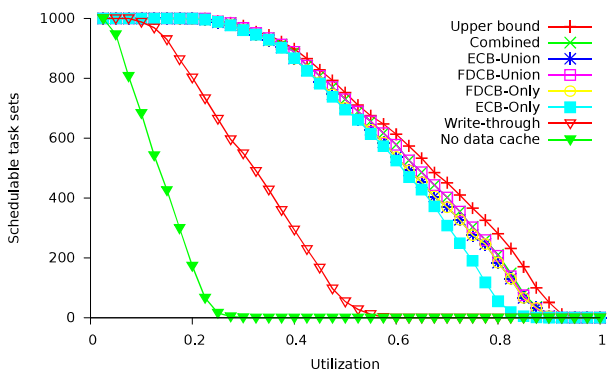


Figure 5: Number of schedulable task sets (FPNS).

We evaluated the guaranteed performance of the various approaches on a large number of randomly generated task sets (10000 per utilization level for the baseline experiments, and 100 per level for the weighted schedulability [9]) 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 1, corresponding to code from the benchmarks.
- The task utilizations (U_i) were generated using UUnifast [10].
- Task periods were set based on utilization and the stand-alone WCET for a write-back cache, i.e., $T_i = C_i^{wb}/U_i$.
- 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 4 and 5 show the baseline results for FPPS and FPNS respectively. Table 2 summarises these results using the weighted schedulability measure [9].

Table 2: Weighted Schedulability 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-through	0.249231	0.112666
No data cache	0.052548	0.021463

Additional experimental results showing how this measure varies with the number of tasks and with the memory latency are given in Appendix A of the technical report [17]. 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 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. For preemptive scheduling, in *all* cases, we include the cost of additional cache misses due to CRPD using the UCB-Union approach [3].

The results shown in Figures 4 and 5 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. Further, the *upper bound* line indicates that the combined approaches used to analyse write-back cache offer a high degree of precision.

7. ECB, DCB, AND FDCB ANALYSES

This paper focusses on the integration of the overheads due to write backs into response time analysis. As a proof-of-concept of the analysis techniques, we obtained the WCETs and the sets of DCBs and ECBs used in the evaluation from a trace of accesses obtained for each of the programs in the Mälardalen [20] and EEMBC [1] 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. Traces for the benchmarks were then generated using the gem5 instruction set simulator [11]. Bounds on the sets of UCBs, ECBs, DCBs, and FDCBs for each benchmark were derived from the traces via cache simulation. (Note, we assumed that the location of code and data in memory was fixed for all runs of the program, as is common in simple embedded systems). 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. The development and implementation of such techniques is the subject of our ongoing work.

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 therefore provides a sound but pessimistic approximation of the set of FDCBs. A more precise approximation can be obtained using *may-cache analysis* [19]. This computes for each program point an over-approximation of the cache contents, i.e., of 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 in [19], 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. 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 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 analysis techniques presented in this paper to deal with such uncertainty is the subject of ongoing research.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced for the first time, analysis for a write-back cache which we integrated into response-time analysis for fixed-priority preemptive and fixed-priority non-preemptive scheduling. 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 behavior, 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 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 [7] notion for dirty cache blocks. We are also extending this work to consider *write buffers* which can be used to improve efficiency with write-through and write-back policies. Preliminary results in this area can be found in Appendix C of [17]. There we show 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.

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 [26]. In this work, we assume that job-internal write backs are accounted for in a task's WCET bound, in future, we aim to integrate a precise analysis of job-internal write

backs into WCET analysis. We also note that uncertainty / imprecision in determining the sets of ECBs, UCBs, DCBs, and FDCBs challenges the 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.

9. ACKNOWLEDGMENTS

This work was supported by the the COST Action IC1202 TACLe, the UK EPSRC Project MCC (EP/K011626/1), 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 http://www.dagstuhl.de/15121. Finally, we would like to thank Benjamin Lesage for his comments on an earlier draft.

10. REFERENCES

[1] EEMBC Autobench. http://www.eembc.org/benchmark/automotive_sl.php. Accessed: 2016-04-29.

[2] S. Altmeyer, R.I. Davis, and C. Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 261–271, December 2011.

[3] S. Altmeyer, R.I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[4] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26, July 2014.

[5] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems*, pages 1–46, Jan 2016.

[6] S. Altmeyer and C. Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57:707–719, August 2011.

[7] S. Altmeyer, C. Maiza, and J. Reineke. 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)*, pages 153–162, April 2010.

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993.

[9] A. Bastoni et al. 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)*, pages 33–44, July 2010.

[10] E. Bini and G.C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[11] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[12] R.J. Bril, S. Altmeyer, M. van den Heuvel, R.I. Davis, and M. Behnam. 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)*, pages 161–172, December 2014.

[13] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1):63–119, 2009.

[14] C. Burguière, J. Reineke, and S. Altmeyer. 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)*, pages 1–11, 2009.

[15] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S.H. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[16] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the IEEE Real-Time Embedded Technology and Applications (RTAS)*, pages 204–212, June 1996.

[17] R.I. Davis, S. Altmeyer, and J. Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. Technical report https://www.cs.york.ac.uk/ftpdir/reports/2016/YCS/502/YCS-2016-502.pdf, University of York, 2016.

[18] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[19] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Sys.*, 17(2-3):131–181, 1999.

[20] J. Gustafsson et al. The Mälardalen WCET benchmarks – past, present and future. In *Proceedings of the International workshop on Worst-Case Execution Time Analysis (WCET)*, pages 137–147, July 2010.

[21] S. Hahn, J. Reineke, and R. Wilhelm. 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)*, December 2013.

[22] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 1986.

[23] D.I. Katcher, H. Arakawa, and J.K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 19, 1993.

[24] Chang-Gun Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[25] Y-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 254–263, Dec 1996.

[26] W. Lunniss, S. Altmeyer, and R.I. Davis. 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)*, pages 161–170, 2012.

[27] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceeding of the IEEE Real-Time Systems Symposium (RTSS)*, pages 25–34, December 2000.

[28] T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 395–404, Nov 2010.

[29] J. Staschulat, S. Schliecker, and R. Ernst. 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)*, pages 41–48, July 2005.

[30] Y. Tan and V. J. Mooney. Timing analysis for preemptive multi-tasking real-time systems with caches. *ACM Trans. on Embedded Comput. Syst.*, 6(1), 2007.

[31] C. Wang, Z. Gu, and H. Zeng. 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)*, pages 69–79, 2015.

[32] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 328–335, 1999.