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.

Early work on accounting for scheduling overheads in fixed-priority preemptive systems by Katcher et al. [31] and Burns et al. [17] 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 [7] 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 [18], which considers just the preempting task; while in 1998, Lee et al. developed the UCB-Only approach [33], which considers just the preempted task(s). Both the UCB-Union approach [46] developed by Tan and Mooney in 2007, and the ECB-Union approach [3] derived by Altmeyer et al. in 2011 consider both the preempted and preempting tasks. As does an alternative approach [45] 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 [4].

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. [5], [6] 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 [48, 38] 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. [14] integrated CRPD into analysis for fixed-priority scheduling with preemption thresholds. Further work in this area by Wang et al. [47] 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. [5].

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

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

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 [3].

Ferdinand and Wilhelm [24] 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 [43] 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 [24, 43] 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 [34] 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 [28], 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.

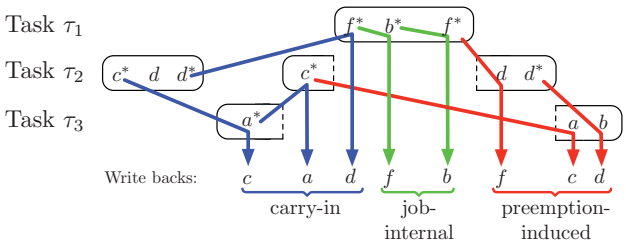


Figure 1: Example illustrating different kinds of write backs.

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 [4].

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 [9], [30]:

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

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 [15]. (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) [22]. 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.

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

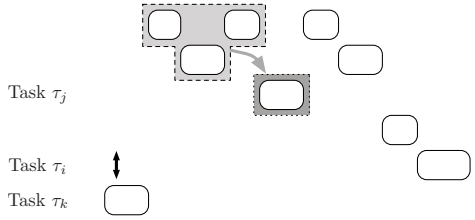
$$R_i^{NP} = W_i^{NP} + C_i \quad (7)$$

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).

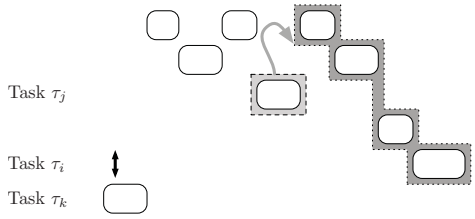
4. WRITE BACKS UNDER FPNS

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 [28], 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 [16].

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 [21].

5. WRITE BACKS UNDER FPPS

Response-time analysis for FPPS has previously been extended to account for preemption-related cache misses [3], [4] 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 [4] 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:

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

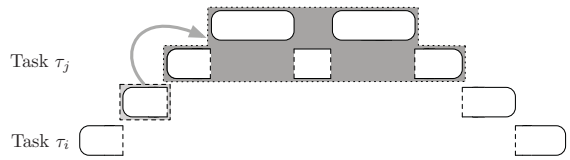
5.2 Lower-Priority Carry-in Write Backs

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.

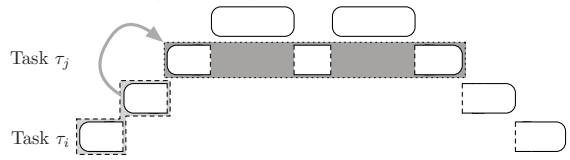
(a) the lp-carry-in write backs of dirty cache lines introduced by the job *immediately-preempted* by a job of τ_j that occur within the response time of τ_j, i.e. either executing τ_j or a higher-priority task.

(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 [21].

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 [26] 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 [21], along with a complete table of values.

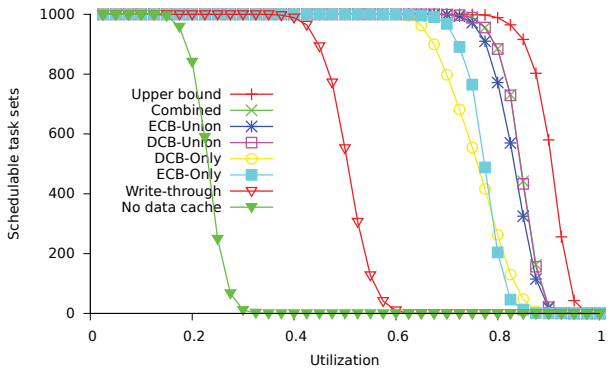


Figure 4: Number of schedulable task sets (FPPS).

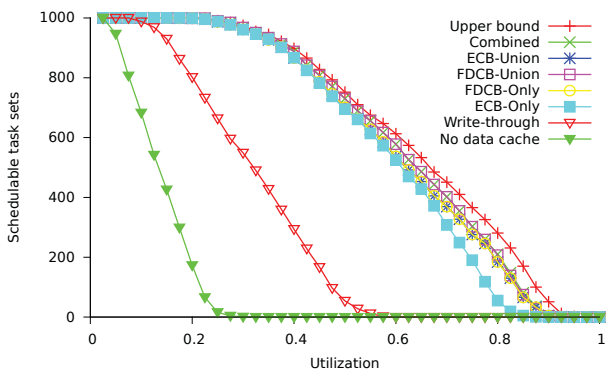


Figure 5: Number of schedulable task sets (FPNS).

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 [11]) 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 [12].
- 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 [11].

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 [21]. 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 [4].

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 [26] 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 [13]. 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* [24]. 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 [24], 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 [8] 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 [21]. 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 [36]. 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. *Analysis of Preemptively Scheduled Hard Real-time Systems*. epubli GmbH, 2013.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] S. Baruah and A. Burns. Sustainable scheduling analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, Dec 2006.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 286–297, New York, NY, USA, 2001. ACM.

[20] D-H. Chu, J. Jaffar, and R. Maghareh. Precise cache timing analysis via symbolic execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, April 2016.

[21] 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.

[22] 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.

[23] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, UK, 1998. Springer-Verlag.

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

[25] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, July 1999.

[26] 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.

[27] S. Hahn and D. Grund. Relational cache analysis for static timing analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 102–111, July 2012.

[28] 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.

[29] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011.

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

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

[32] S-K Kim, S.L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 230–240, Jun 1996.

[33] 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.

[34] 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.

[35] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 255–262, 1999.

[36] 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.

[37] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 148–157, March 2005.

[38] 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.

[39] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 195–204, 2000.

[40] M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Syst.*, 49(1):1–28, January 2013.

[41] R. Sen and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 203–212, New York, NY, USA, 2007. ACM.

[42] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 144–155, Feb 1997.

[43] 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.

[44] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 10 pp.–236, 2006.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] S. Wegener. Computing Same Block Relations for Relational Cache Analysis. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 25–37, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[50] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 192–202, Jun 1997.

APPENDIX

A. 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. Let $S_y(\tau, p)$ be the binary result (1 or 0) of schedulability test y for a task set τ assuming parameter p .

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

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

Figures 6 and 8 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.

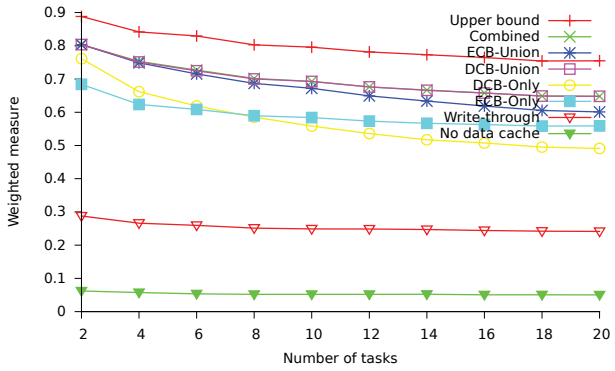


Figure 6: Weighted schedulability vs. number of tasks (FPPS).

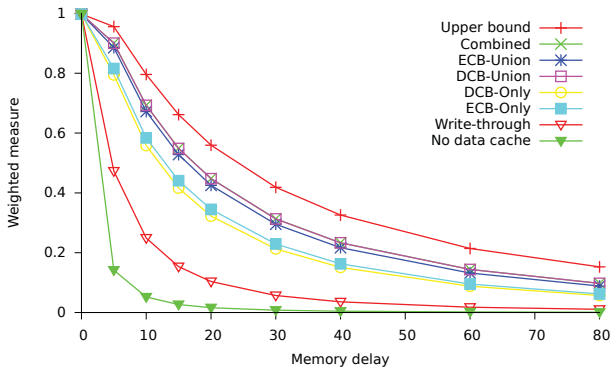


Figure 7: Weighted schedulability vs. memory latency (FPPS).

This enables an overall increase in schedulability 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 7 and 9 show how the weighted schedulability metric 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 (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.

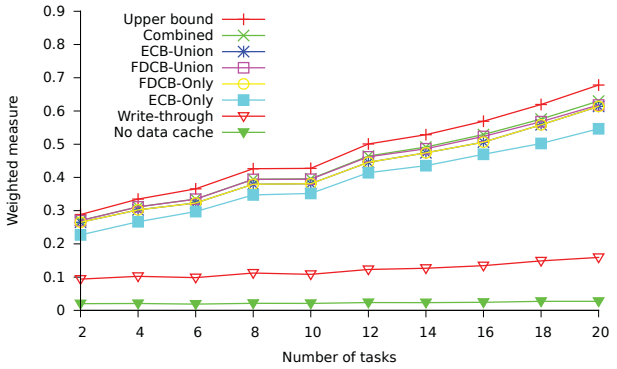


Figure 8: Weighted schedulability vs. number of tasks (FPNS).

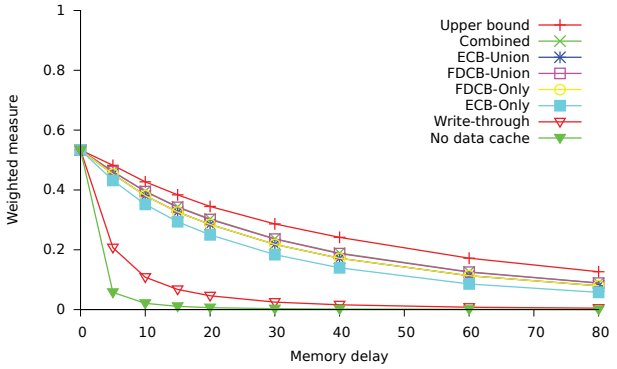


Figure 9: Weighted schedulability vs. memory latency (FPNS).

B. WORKED EXAMPLE

Below, we present a worked example illustrating the various approaches to analysing write backs and their differences in performance. Table 3 gives the task set parameters.

Table 3: Example Task Set

Task	C	T	ECB	DCB	FDCB
τ_1	100	1000	$\{1,4,5\}$	$\{1\}$	$\{1\}$
τ_2	100	1000	$\{2,3,4,5\}$	$\{2,3,4\}$	$\{2,3\}$
τ_3	100	1000	$\{2,3,5\}$	$\{2,3,5\}$	$\{2,3\}$
τ_4	100	1000	$\{1,2,3,4,5,6\}$	$\{1,2,3,4,5,6\}$	$\{1\}$

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).

B.1 Fixed Priority Non-preemptive Scheduling (FPNS)

B.1.1 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 $R_4 = 522$.

B.1.2 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 δ_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 γ accounts for write backs due to higher priority tasks:

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

The response time is then computed using (11) and (12) giving $R_1 = 204$, $R_2 = 306$, $R_3 = 408$, and $R_4 = 511$.

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

B.1.3 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 δ accounts for dirty cache lines prior to the release of the task under analysis and is given by (14) thus $\delta = 3$. The γ term is given by (13):

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

The response time is then computed using (11) and (15) giving $R_1 = 205$, $R_2 = 306$, $R_3 = 408$, and $R_4 = 509$.

B.1.4 ECB-Union

The ECB-Union approach only differs from FDCB-Only in the δ 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 τ_1 is smaller with FDCB-Union (204 vs. 205), while the response time for task τ_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.

B.2 Fixed Priority Preemptive Scheduling (FPPS)

In the case of fixed-priority preemptive scheduling, all four approaches use the same response-time equation (20), the same δ_i terms to account for initially dirty cache lines (21): $\delta_1 = 3$, $\delta_2 = 4$, $\delta_3 = 5$, $\delta_4 = 6$. and the same $\gamma_{i,j}^{\text{wb-fin}}$ terms to account for the finished carry-in write backs (28): $\gamma_{-,1}^{\text{wb-fin}} = 1$, $\gamma_{-,2}^{\text{wb-fin}} = 2$, $\gamma_{-,3}^{\text{wb-fin}} = 2$, $\gamma_{-,4}^{\text{wb-fin}} = 1$. The approaches only differ in the $\gamma_{i,j}^{\text{wb-lp}}$ terms.

B.2.1 DCB-Only

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

$R_1 = 103$, $R_2 = 209$, $R_3 = 314$, $R_4 = 429$.

B.2.2 ECB-Union

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

$R_1 = 103$, $R_2 = 207$, $R_3 = 312$, $R_4 = 424$.

B.2.3 ECB-Only

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

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

B.2.4 DCB-Union

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

$R_1 = 103$, $R_2 = 207$, $R_3 = 313$, $R_4 = 421$.

The example shows the dominance relationships of ECB-Union over DCB-Only and DCB-Union over ECB-Only, as well as the incomparability between DCB-Only and ECB-Only and between ECB-Union and DCB-Union. The response time for task τ_3 is smaller with ECB-Union than with DCB-Union (312 vs. 313) and smaller

with DCB-Only than with ECB-Only (314 vs. 315). Vice versa, the response time for task τ_4 is smaller with DCB-Union than with ECB-Union (421 vs. 424) and smaller with ECB-Only than with DCB-Only (421 vs. 429). The combined approach, taking the minimum response times (103. 207, 312, 421) thus dominates all others.

B.3 Sustainability of the analysis

The analysis given in this paper builds upon response time analyses for FPPS and FPNS, integrating the effects of CRPD for write-back caches. The response time analyses used for FPPS and FPNS are both *sustainable* [10], 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 CRPD given in Sections 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 easily seen from the formulaes involved, since the response times computed are monotonically non-decreasing with respect to increases (additions of elements) to any of these sets.

C. 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 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; 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 between the cache and memory, which holds data 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.

Write buffers are characterized by a *depth* giving the number of entries, and a *width*, typically the same as a cache line, as well as the policies defining their operation. These policies include: the *local hazard* policy, which determines what happens when a read access occurs to an address that is currently in the write buffer; the *coalescence policy*, which determines what happens when a write access occurs to an address that is currently in the write buffer, and finally 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 [42] which discusses write buffer design from the perspective of improving average case performance).

C.1 Local hazard policy

With a write buffer between cache and memory, then a naive design could result in data inconsistency, termed a *local hazard*.

If a read occurs which misses in the cache, but the data is in the write buffer waiting to be written to memory, then if the read goes to memory, inconsistent data could be 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. We note that 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.

C.2 Coalescence policy

Entries in a write buffer consist of an address and a block of data, typically containing the same number of words as a cache line. When a write occurs and there are no entries in the write buffer, then the block is copied to the write buffer and the word that is being written is marked as valid via a flag bit, indicating that it should be later 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 an number of advantages, it enables multiple writes to the same address or same block to be coalesced, resulting in fewer writes to memory. This mechanism has similarities to a write-back cache, in that it takes advantage of the spacial locality of writes. Merging writes makes better use of the limited capacity of the write buffer.

The alternative to merging writes is to instead simply add a new entry to the write buffer on each write. This still facilitates latency hiding with the processor able to continue with other instructions while writes to memory take place; however it does not take advantage of spacial locality. We refer to this approach as *no write merge*. While the performance of no write merge can reasonably be expected to be worse than write merge in the average case, we note that it has some advantages in terms of timing composition and analysable guaranteed worst-case performance.

C.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 *eager retirement* where write buffer entries are written to memory as soon as possible, and *lazy* retirement where they are written as late as possible i.e. 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, since 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 on the other hand keep 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. 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 some value, such as 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.

C.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 behaviors 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 behaviors whereby a small change in cache contents due to preemption can result in an unbounded increase in the number of cache misses (see pages 56-57 of [2] for worked examples). Such designs are not timing compositional and present a substantial challenge in terms of analysing their worst-case performance. Such designs 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 prevent bounding the write buffer related preemption delay to a constant value, and hence effectively prevent an integrated analysis with fixed priority preemptive scheduling [39].

C.5 Domino effects

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

C.5.1 Reading from the write buffer combined with lazy retirement

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

Consider the following sequence of memory accesses, where $*$ indicates a write: $a^*, b, a, b, a, b, a, b, a, \ldots$, and a and b are mapped to the same set in a direct mapped cache. Executing this sequence of accesses results in the following behavior. Since the first access is a write, a is copied to the write buffer where, due to lazy retirement, it remains for the rest of the sequence. Next, b (a miss) evicts a from the cache. However, since a is in the write buffer it can be read from there. The result is that all accesses to b are misses while all accesses to a except the first are hits.

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.

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.

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

The write buffer contents are depicted in Figure 10 (a) for non-preempted execution, (b) for non-preempted execution, and (c) for non-preempted execution starting from a non-empty buffer.

Without preemption, 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. 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 Figure 10 (b)), 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 10 (c) illustrates what happens with non-preempted execution starting from a non-empty write buffer, containing b. This has the effect of switching the order of a and b in the buffer, the result of which is that every write in the final repeating sub-sequence c^*, b^*, a^* causes a stall. This example illustrates that an empty write buffer does not necessarily result in the worst-case behavior.

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

C.5.3 Write merge combined with eager retirement

We assume a write buffer of depth 3, with each slot able to hold an address and a word of data. We assume that writes are retired from the buffer as soon as possible in FIFO order, and that the time to retire an entry is substantially longer than that for an access that does not go to memory. Further, we assume that while an entry is being retired, it cannot be merged into by another write. The local hazard policy is irrelevant to this example, since there are no reads.

We modify the example assumed above for lazy retirement by adding a further write d^* at the beginning of the sequence. Now in the case without preemption, shown in Figure 11 (a), 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 c^* then stalls until retirement of d to memory completes. Once that happens, the buffer contains c, b, and a which starts being retired and so is unavailable for merges. It is easy to see that as the sequence progresses, the entries available for merging into are identical to those shown for a buffer of size 2 in Figure 10 (a) with a further entry that is in the process of being retired and is therefore not available for merging. As before, 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.

The case with preemption is shown in Figure 11 (b). As with eager retirement, every write access results in a stall while an entry is written to memory, with the effect persisting indefinitely.

Finally, Figure 11 (c) illustrates what happens with non-preempted execution starting from a non-empty write buffer, containing b, d, and z which is being retired. This has the effect of switching the order of a and b in the buffer, the result of which is that every write in the final repeating sub-sequence c^*, b^*, a^* causes a stall. This example shows that, similar to the case with lazy retirement, an empty write buffer does not necessarily result in the worst-case behavior.

We note that these domino effects extend to buffers of size 3 or more by using longer sub-sequences.

C.6 Analysis of Write Merge

In the previous subsection, we showed that write merge can result in domino effects with both eager and lazy retirement. This is problematic since write merge is effective in taking advantage of spacial locality. Read from the write buffer on local hazards (i.e. read access to an address in the buffer) also introduces domino effects. In the following, we therefore assume that local hazards result in a full flush of the write buffer[2].

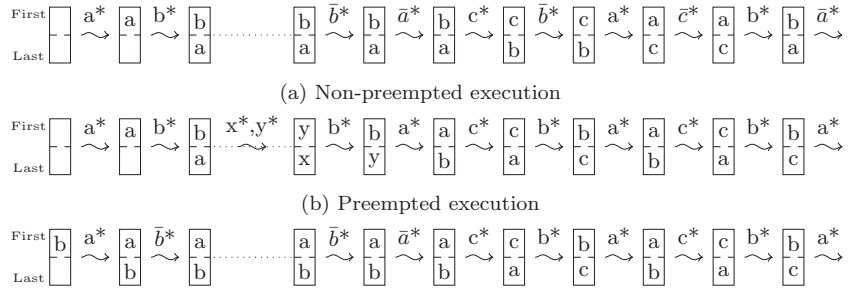
We note that sound, compositional analysis for write merge can be provided under certain specific configurations. These are (i) with a write buffer of depth 1, and (ii) for fixed priority non-preemptive scheduling with a write buffer of arbitrary but known depth. In both cases, we require full flush of the buffer on local hazards.

With a write buffer of depth 1, read accesses to the buffer flushing the contents, and lazy retirement of entries from the buffer, there can be no domino effects. In this case, sound analysis for FPNS can be achieved simply by assuming that the buffer contains junk at the start of each job[3] With FPPS, we also need to account for an additional preemption cost that equates to flushing the buffer.

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

[2]This is the policy used in the ARM 9 architecture.

[3]This junk will need to be written to memory before a further write access can make use of the buffer. Since its address is assumed to be unknown, any read is also assumed to flush the junk to memory.

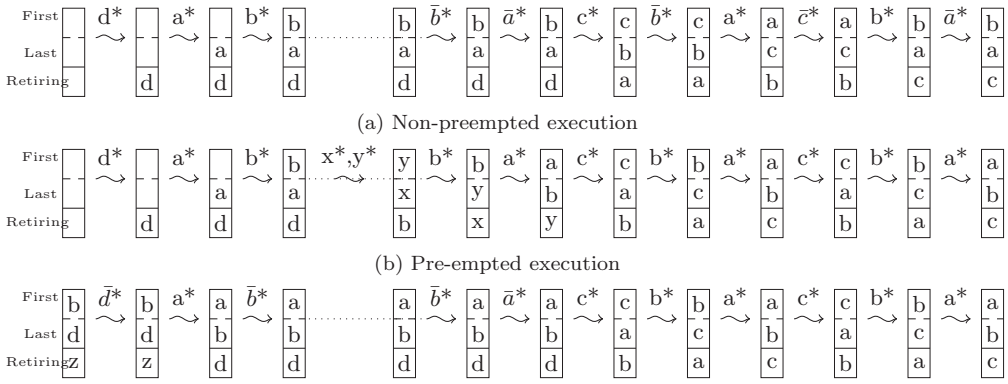


(a) Non-preempted execution

(b) Preempted execution

(c) Non-preempted execution, starting with a non-empty write buffer

Figure 10: Domino effect with write merge and lazy retirement. Note $\bar{b}$ indicates a write merge to address b in the write-buffer.



(a) Non-preempted execution

(b) Pre-empted execution

(c) Non-preempted execution, starting with a non-empty write buffer

Figure 11: Domino effect with write merge and eager retirement. Note $\bar{b}$ indicates a write merge to address b in the write-buffer.

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. We label these write accesses $w1^*$, for example in a sequence such as $r1, r2, w1^*, r3, w1^*, r4, w1^*, w2^*$, where r are 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 is guaranteed to start execution with an empty write buffer. This can be achieved if each job flushes 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 > 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.

C.7 Analysis of Write No Merge

For completeness, we now provide analysis for write no 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 only advantages that the buffer conveys is latency hiding. Hence the optimum retirement policy in this case is eager retirement. While this approach permits a simple analysis, it has the significant disadvantage that it does not make the most of spacial locality to optimise performance.

Since 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 which is then subject to an additional delay while these entries are retired (for example as a consequence of making a read to one of them). The WBRPD therefore 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 simply modeled by inflating all task execution times

Table 4: Execution times estimates for the Mälardalen and EEMBC benchmarks used for evaluation

Name	C^{wb}	$C^{\mathrm{wt}-0}$	$\frac{C^{\mathrm{wt}-0}}{C^{\mathrm{wb}}}$	$C^{\mathrm{wt}-1}$	$\frac{C^{\mathrm{wt}-1}}{C^{\mathrm{wb}}}$	$C^{\mathrm{wt}-2}$	$\frac{C^{\mathrm{wt}-2}}{C^{\mathrm{wb}}}$	$C^{\mathrm{wt}-4}$	$\frac{C^{\mathrm{wt}-4}}{C^{\mathrm{wb}}}$	C^{nc}	$\frac{C^{\mathrm{nc}}}{C^{\mathrm{wb}}}$
cnt	9325	13485	1.44	9815	1.05	9745	1.04	9695	1.03	24565	2.63
compress	10673	18713	1.75	16963	1.58	16403	1.53	14863	1.39	43443	4.07
countneg	36180	57250	1.58	56500	1.56	48450	1.33	37260	1.02	114340	3.16
crc	68889	133909	1.94	79469	1.15	79419	1.15	69759	1.01	272859	3.96
expint	9268	15208	1.64	12548	1.35	9508	1.02	9448	1.01	31098	3.35
fdct	7883	16793	2.13	11403	1.44	10203	1.29	9253	1.17	38423	4.87
fir	8328	18998	2.28	13718	1.64	8858	1.06	8548	1.02	43668	5.24
jfdctint	9711	18621	1.91	14141	1.45	12291	1.26	11601	1.19	39181	4.03
loop3	14189	28729	2.02	26909	1.89	14369	1.01	14349	1.01	57929	4.08
ludcmp	10058	15948	1.58	13178	1.31	11628	1.15	10828	1.07	39668	3.94
minver	18976	30616	1.61	23226	1.22	22276	1.17	20026	1.05	54746	2.88
ns	27464	37674	1.37	27704	1.00	27644	1.00	27624	1.00	98634	3.59
nsichneu	18988	24458	1.28	20068	1.05	20028	1.05	19988	1.05	66808	3.51
qurt	10473	16003	1.52	12293	1.17	11483	1.09	10873	1.03	23573	2.25
select	8981	17031	1.89	12181	1.35	11251	1.25	9961	1.10	30331	3.37
sqrt	27667	40537	1.46	34607	1.25	31037	1.12	28147	1.01	59117	2.13
statemate	64638	195778	3.02	120958	1.87	102918	1.59	96858	1.49	581908	9.00
a2time	12655	22975	1.81	22825	1.80	12645	.99	12635	.99	53815	4.25
aifirf	44898	86768	1.93	77528	1.72	41508	.92	41508	.92	181698	4.04
basefp	50491	92221	1.82	91421	1.81	50801	1.00	50651	1.00	213771	4.23
canrdr	32641	65211	1.99	64811	1.98	33261	1.01	33141	1.01	156611	4.79
iirflt	29995	56995	1.90	54845	1.82	34445	1.14	32865	1.09	127605	4.25
pntrch	23887	43137	1.80	42447	1.77	22627	.94	22627	.94	109257	4.57
puwmod	48782	97072	1.98	96702	1.98	48642	.99	48592	.99	239752	4.91
rspeed	10913	21393	1.96	21213	1.94	12103	1.10	11933	1.09	51713	4.73
tblook	12533	25493	2.03	22383	1.78	19923	1.58	13573	1.08	58813	4.69

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, which is $M \cdot WBT$.

C.8 Write buffers and write-back caches

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

In theory, the domino effects noted previously with a write buffer and write-through cache also apply in the case of a write-back cache. (The precise sequences needed to show this behavior differ, since writes first have to be evicted from the cache before they are written to the buffer; however, 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; however, since a write-back cache already captures spacial locality of writes, there is little advantage to be gained from buffering writes, except for latency hiding. This can be sufficiently well with a buffer of depth 1. (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.

C.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, since the aim is to hide write latency. 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 write no 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, assuming these policies, are given in Table 4

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 Figures 12 to 17, 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 2 and 4 with potentially optimistic analysis; the latter indicated by 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 a repeat of those presented in Figures 4 to 9. For write-back caches, the analysis used is the combined approach which is the most effective of all the methods presented in this paper.

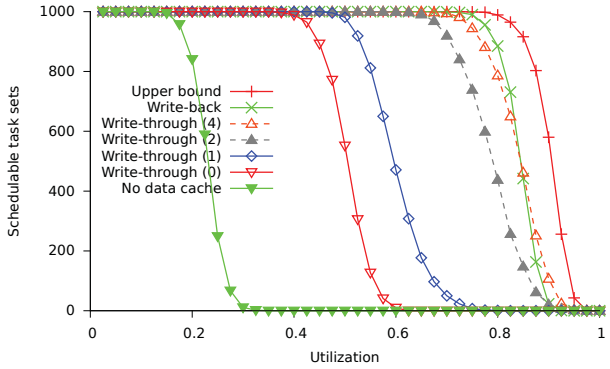


Figure 12: Number of schedulable task sets (FPPS).

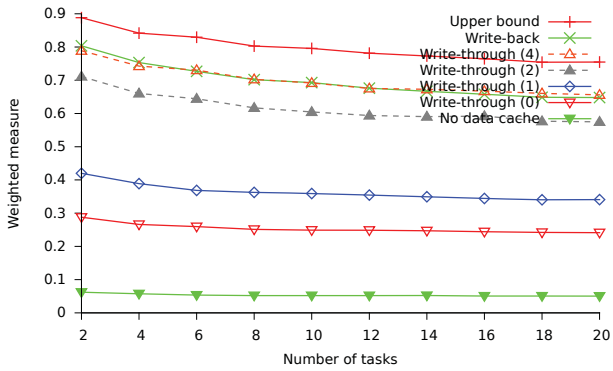


Figure 13: Weighted schedulability vs. number of tasks (FPPS).

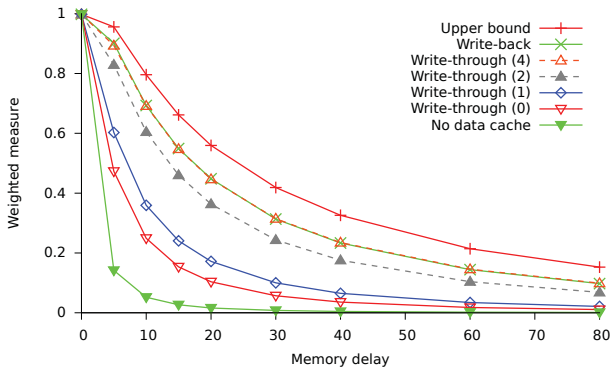


Figure 14: Weighted schedulability vs. memory latency (FPPS).

We observe in Table 4 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.

Note: The results for a write-back cache with a write buffer of depth 1 and *eager retirement*, are TBD.

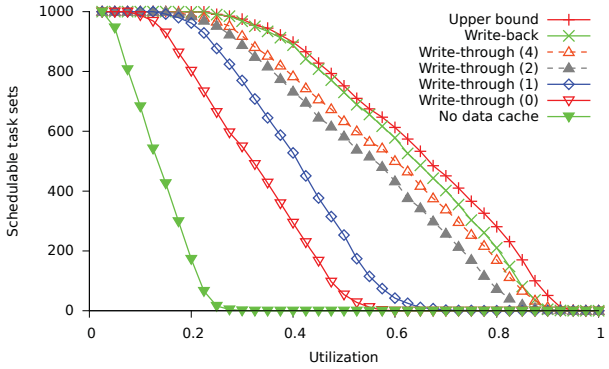


Figure 15: Number of schedulable task sets (FPNS).

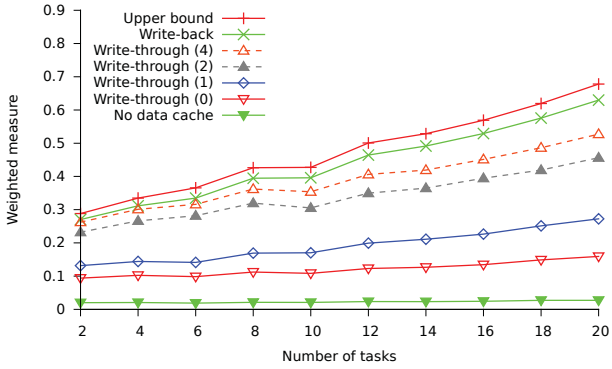


Figure 16: Weighted schedulability vs. number of tasks (FPNS).

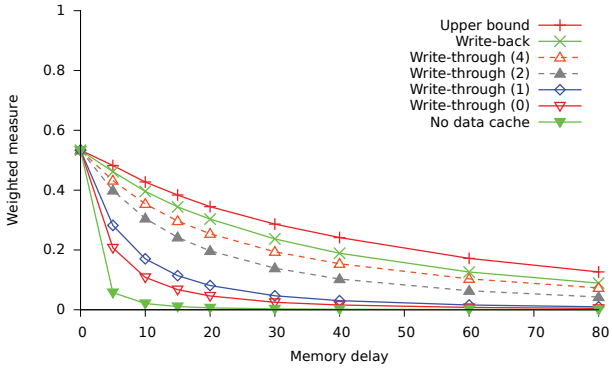


Figure 17: Weighted schedulability vs. memory latency (FPNS).

This observation is born out by the schedulability analysis results. Figures 12 to 17 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 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.

Table 5: Execution times estimates for all of the Mälardalen and EEMBC benchmarks

Name	C^{wb}	$C^{\text{wt}-0}$	$\frac{C^{\text{wt}-0}}{C^{\text{wb}}}$	$C^{\text{wt}-1}$	$\frac{C^{\text{wt}-1}}{C^{\text{wb}}}$	$C^{\text{wt}-2}$	$\frac{C^{\text{wt}-2}}{C^{\text{wb}}}$	$C^{\text{wt}-4}$	$\frac{C^{\text{wt}-4}}{C^{\text{wb}}}$	C^{nc}	$\frac{C^{\text{nc}}}{C^{\text{wb}}}$
adpcm_dec	630673	1013293	1.60	750193	1.18	634063	1.00	632473	1.00	1866353	2.95
adpcm_enc	632065	1016075	1.60	751225	1.18	634665	1.00	633835	1.00	1872975	2.96
binarysearch	1938	2958	1.52	2278	1.17	2178	1.12	2138	1.10	4388	2.26
bs	1888	2908	1.54	2218	1.17	2128	1.12	2088	1.10	4338	2.29
bsort100	273945	527345	1.92	380985	1.39	284415	1.03	282195	1.03	1579665	5.76
cnt	9325	13485	1.44	9815	1.05	9745	1.04	9695	1.03	24565	2.63
compress	10673	18713	1.75	16963	1.58	16403	1.53	14863	1.39	43443	4.07
countneg	36180	57250	1.58	56500	1.56	48450	1.33	37260	1.02	114340	3.16
cover	6041	10611	1.75	9921	1.64	6241	1.03	6211	1.02	20341	3.36
crc	68889	133909	1.94	79469	1.15	79419	1.15	69759	1.01	272859	3.96
duff	4361	8641	1.98	7501	1.72	4701	1.07	4651	1.06	18801	4.31
edn	167426	318356	1.90	179776	1.07	170796	1.02	167486	1.00	905206	5.40
expint	9268	15208	1.64	12548	1.35	9508	1.02	9448	1.01	31098	3.35
fac	2206	3826	1.73	2656	1.20	2576	1.16	2416	1.09	5926	2.68
fdct	7883	16793	2.13	11403	1.44	10203	1.29	9253	1.17	38423	4.87
fft1	94559	139939	1.47	112149	1.18	103769	1.09	97439	1.03	206419	2.18
fibcall	2304	4374	1.89	3124	1.35	2504	1.08	2474	1.07	7594	3.29
fir	8328	18998	2.28	13718	1.64	8858	1.06	8548	1.02	43668	5.24
insertsort	3428	6358	1.85	4628	1.35	3748	1.09	3638	1.06	16158	4.71
janne	2238	3568	1.59	2498	1.11	2418	1.08	2398	1.07	5718	2.55
jfdctint	9711	18621	1.91	14141	1.45	12291	1.26	11601	1.19	39181	4.03
lcdnum	1934	3184	1.64	2274	1.17	2154	1.11	2124	1.09	4764	2.46
lms	3026313	4232013	1.39	3634173	1.20	3245623	1.07	3072853	1.01	6764523	2.23
loop3	14189	28729	2.02	26909	1.89	14369	1.01	14349	1.01	57929	4.08
ludcmp	10058	15948	1.58	13178	1.31	11628	1.15	10828	1.07	39668	3.94
matmult	385500	503360	1.30	418240	1.08	418190	1.08	388670	1.00	1165620	3.02
minver	18976	30616	1.61	23226	1.22	22276	1.17	20026	1.05	54746	2.88
ndes	110457	239807	2.17	163627	1.48	127167	1.15	113367	1.02	615987	5.57
ns	27464	37674	1.37	27704	1.00	27644	1.00	27624	1.00	98634	3.59
nsichneu	18988	24458	1.28	20068	1.05	20028	1.05	19988	1.05	66808	3.51
petrinet	4732	6642	1.40	5192	1.09	5152	1.08	5112	1.08	16372	3.45
qsort	1625	2545	1.56	1895	1.16	1855	1.14	1825	1.12	3425	2.10
qurt	10473	16003	1.52	12293	1.17	11483	1.09	10873	1.03	23573	2.25
recursion	6694	14624	2.18	9444	1.41	8074	1.20	7474	1.11	25774	3.85
select	8981	17031	1.89	12181	1.35	11251	1.25	9961	1.10	30331	3.37
sqrt	27667	40537	1.46	34607	1.25	31037	1.12	28147	1.01	59117	2.13
st	1502532	1818162	1.21	1680412	1.11	1531952	1.01	1503842	1.00	2759462	1.83
statemate	64638	195778	3.02	120958	1.87	102918	1.59	96858	1.49	581908	9.00
a2time	12655	22975	1.81	22825	1.80	12645	.99	12635	.99	53815	4.25
aifftr	2394700	4542600	1.89	2475390	1.03	2379620	.99	2370360	.98	9995200	4.17
aifirf	44898	86768	1.93	77528	1.72	41508	.92	41508	.92	181698	4.04
aiifft	2338700	4473260	1.91	2435610	1.04	2317730	.99	2317330	.99	9755590	4.17
basefp	50491	92221	1.82	91421	1.81	50801	1.00	50651	1.00	213771	4.23
bitmnp	195198	257238	1.31	216878	1.11	206788	1.05	198958	1.01	686778	3.51
cacheb	332278	556798	1.67	554728	1.66	551398	1.65	211398	.63	937468	2.82
canrdr	32641	65211	1.99	64811	1.98	33261	1.01	33141	1.01	156611	4.79
idctrn	400997	678097	1.69	651887	1.62	401887	1.00	401437	1.00	1675437	4.17
iirflt	29995	56995	1.90	54845	1.82	34445	1.14	32865	1.09	127605	4.25
matrix	11795117	15853627	1.34	11613177	.98	11351287	.96	11350567	.96	41363137	3.50
pntrch	23887	43137	1.80	42447	1.77	22627	.94	22627	.94	109257	4.57
puwmod	48782	97072	1.98	96702	1.98	48642	.99	48592	.99	239752	4.91
rspeed	10913	21393	1.96	21213	1.94	12103	1.10	11933	1.09	51713	4.73
tblook	12533	25493	2.03	22383	1.78	19923	1.58	13573	1.08	58813	4.69
ttsprk	4529	9159	2.02	6549	1.44	6309	1.39	5919	1.30	14699	3.24

Table 5 gives information for the *complete* set of Mälardalen and EEMBC benchmarks. We observe that the stand alone WCETs are 1.21 to 3.02 times lower with a write-back cache and no write buffer, than with a write-through cache with no write-buffer, and respectively 0.98 to 1.98, 0.92 to 1.59, and 0.63 to 1.49 times lower than a with write-through cache with write buffers of depth 1, 2 and 4. (Note, the cacheb benchmark appears as an outlier which has substantially improved performance with a write-through cache and a write buffer of depth 4. This benchmark is designed to stress the cache, and has a behavior that a write buffer of depth 4 is able to largely circumvent).

D. EVALUATION WITH ALL MÄLARDALEN AND EEMBC BENCHMARKS

For reference, here we provide figures illustrating the performance of the various analyses when all of the Mälardalen and EEMBC benchmarks are used. Across all of the benchmarks there is a large variation in WCET values.

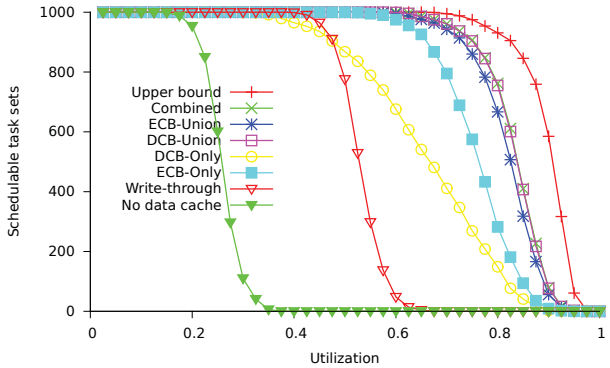


Figure 18: Number of schedulable task sets (FPPS).

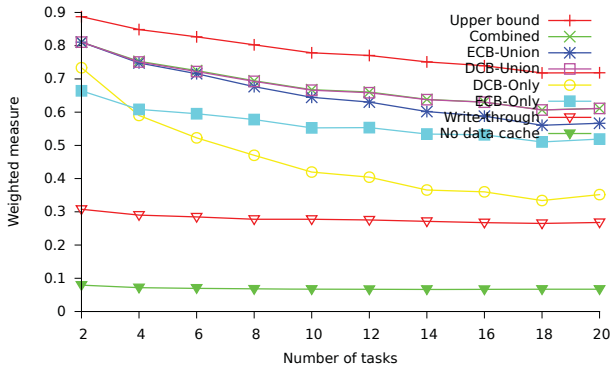


Figure 19: Weighted schedulability vs. number of tasks (FPPS).

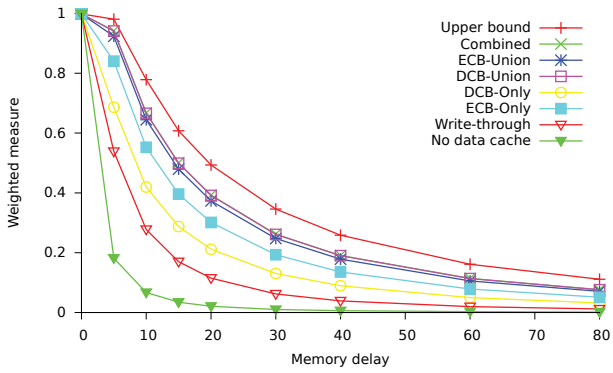


Figure 20: Weighted schedulability vs. memory latency (FPPS).

This leads to a large range of task periods, since they are computed from WCETs using an unbiased set of utilization values generated according to the Uunifast method [12]. Due to the large range of periods and WCETs, many of the task sets are unschedulable under FPNS even when overheads are not considered. This is an issue with non-preemptive scheduling per se, rather than with the analyses presented.

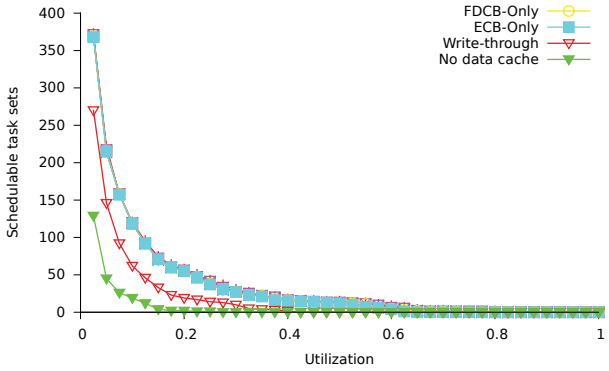


Figure 21: Number of schedulable task sets (FPNS).

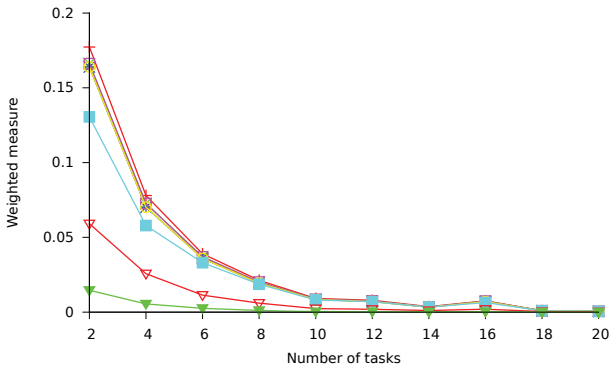


Figure 22: Weighted schedulability vs. number of tasks (FPNS).

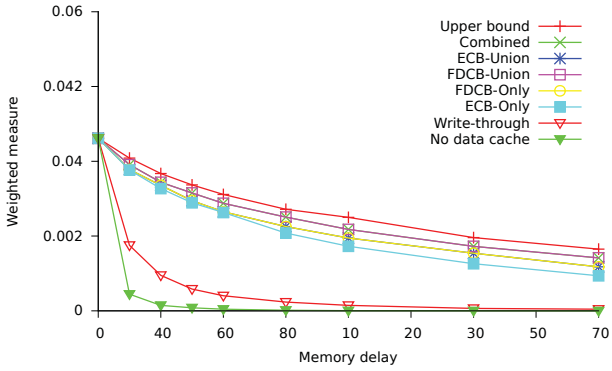


Figure 23: Weighted schedulability vs. memory latency (FPNS).

Table 6: Data from all the Mälardalen and EEMBC benchmarks

Name	C^{wb}	C^{wt}	C^{wt}/C^{wb}	C^{nc}	C^{nc}/C^{wb}	$\|UCB^I\|$	$\|ECB^I\|$	$\|UCB^D\|$	$\|ECB^D\|$	$\|DCB\|$	$\|FDCB\|$
adpcm_dec	630673	1013293	1.60	1866353	2.95	101	230	43	101	39	39
adpcm_enc	632065	1016075	1.60	1872975	2.96	116	242	39	103	38	38
binarysearch	1938	2958	1.52	4388	2.26	10	65	10	52	14	14
bs	1888	2908	1.54	4338	2.29	10	65	9	52	14	14
bsort100	273945	527345	1.92	1579665	5.76	12	76	19	59	26	26
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
cover	6041	10611	1.75	20341	3.36	10	163	9	68	12	12
crc	68889	133909	1.94	272859	3.96	19	89	25	73	40	39
duff	4361	8641	1.98	18801	4.31	14	78	10	51	17	17
edn	167426	318356	1.90	905206	5.40	63	202	41	104	53	51
expint	9268	15208	1.64	31098	3.35	16	76	11	42	13	13
fac	2206	3826	1.73	5926	2.68	7	63	10	45	15	15
fdct	7883	16793	2.13	38423	4.87	52	144	15	48	19	19
fft1	94559	139939	1.47	206419	2.18	114	176	19	54	22	22
fibcall	2304	4374	1.89	7594	3.29	9	61	10	45	12	12
fir	8328	18998	2.28	43668	5.24	22	83	17	57	17	16
insertsort	3428	6358	1.85	16158	4.71	8	68	10	53	16	16
janne	2238	3568	1.59	5718	2.55	9	65	9	47	12	12
jfdctint	9711	18621	1.91	39181	4.03	46	145	17	53	23	23
lcdnum	1934	3184	1.64	4764	2.46	11	66	11	50	14	14
lms	3026313	4232013	1.39	6764523	2.23	105	189	45	87	51	51
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
matmult	385500	503360	1.30	1165620	3.02	16	81	107	191	164	162
minver	18976	30616	1.61	54746	2.88	103	213	18	71	33	33
ndes	110457	239807	2.17	615987	5.57	50	161	46	90	31	31
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
petrinet	4732	6642	1.40	16372	3.45	141	198	19	52	21	21
qsort	1625	2545	1.56	3425	2.10	7	64	11	45	14	14
qurt	10473	16003	1.52	23573	2.25	61	132	14	49	17	17
recursion	6694	14624	2.18	25774	3.85	8	64	11	49	18	18
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
st	1502532	1818162	1.21	2759462	1.83	79	146	262	283	258	255
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
aifftr	2394700	4542600	1.89	9995200	4.17	66	356	299	512	512	341
aifirf	44898	86768	1.93	181698	4.04	25	141	33	188	161	54
aiifft	2338700	4473260	1.91	9755590	4.17	66	320	245	512	512	332
basefp	50491	92221	1.82	213771	4.23	11	88	15	512	507	467
bitmnp	195198	257238	1.31	686778	3.51	118	300	28	91	65	54
cacheb	332278	556798	1.67	937468	2.82	31	87	22	512	512	496
canrdr	32641	65211	1.99	156611	4.79	8	40	9	371	195	186
idctrn	400997	678097	1.69	1675437	4.17	46	264	48	342	268	267
iirflt	29995	56995	1.90	127605	4.25	35	288	28	259	147	138
matrix	11795117	15853627	1.34	41363137	3.50	360	368	435	512	512	380
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
ttsprk	4529	9159	2.02	14699	3.24	24	93	26	41	28	28

In the case of FPPS, the wider range of benchmarks, WCETs and task periods makes little fundamental difference to the results. Comparing in detail Figures 4 and 18, we observe that with a larger range of task periods, schedulability is marginally improved for the upper bound, a well-known effect with FPPS. This translates to similar small improvements in schedulability with no cache and with a write-through cache.

With more preemptions, the ECB-only and particularly the DCB-only approaches for a write-back cache show a decrease in performance due to increased pessimism. The ECB-Union, UCB-Union, and Combined approaches have very similar performance in both cases. With more preemptions a small increase in pessimism can therefore be noted, since the gap to the upper bound opens slightly. These increases in pessimism with more preemptions are inline with the effects of increasing the number of tasks, which also increases the number of preemptions, see Figures 6 and 19.

The WCETs, ECBs, UCBs, DCBs, and FCBs for the *complete* set of benchmarks are given in Table 6.

E. 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 their 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. [32] 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. [32] used a method based on the pigeonhole principle. For each loop, they compute the (i) 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.

In 1996, Li et al. [34] 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.

In 1997, White et al. [50] 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.

In 1998, Ferdinand and Wilhelm [23] 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 full resolved, and thus may take a range of values. Ferdinand and Wilhelm [23] note that *Must* and *May* analysis can be used to determine the data cache behavior 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 behavior 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 behavior to be determined. Solving these equations exactly can however be computationally very expensive.

In 1999, Ghosh et al. [25] introduced a *Cache Miss Equation* framework. This method generates a set of Diophantine equations[4] that describe the behavior 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. Further, no input dependent conditionals are permitted. In 2005, Ramaprasad and Mueller [37] extended the CME approach to handle more general loops via *forced loop fusion*, data-dependent conditionals, and accurately handle scalar variables. Their method produces exact data cache reference patterns, giving the position of misses in a sequence of references.

In 1999, Lundqvist and Stenstrom [35], 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 were the data structure placed in cached memory.

Lundqvist and Stenstrom [35] 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).

In 2001, Chatterjee et al. [19] developed an exact analysis of the cache behavior 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

[4]Equation in two or more unknown values where only integer solutions are sought.

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.

In 2006, Staschulat and Ernst [44] 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 behavior 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.

In 2007, Sen and Srikant [41] presented an approach that combines automatic executable analysis to determine the addresses accessed and a Must analysis for determining cache behavior, 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 solved using an ILP formulation. Sen and Srikant 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, showing improvements in precision with more loop expansion. The virtual loop unrolling is however expensive in terms of analysis time.

In 2011, Huynh et al. [29] 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. 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. [29] also fixed a problem in the original persistence analysis.

In 2012, Wegener [49] 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.

Also in 2012, Hahn and Grund [27] 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 such as same block, same cache set different block, different cache set, between symbolic names, as well as approximations that are still 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 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, since it is at least as precise. Hahn and Grund [27] 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.

In 2013, Schoeberl et al. [40] 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. present a scope-based persistence analysis.