Write-Back Caches in WCET Analysis*

Tobias Blaß[1], Sebastian Hahn[2], and Jan Reineke[3]

1 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`s9toblas@stud.uni-saarland.de`
2 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`sebastian.hahn@cs.uni-saarland.de`
3 **Saarland Informatics Campus, Saarland University, Saarbrücken, Germany**
`reineke@cs.uni-saarland.de`

Abstract

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

In this paper, we introduce a new approach to statically analyze the behavior of write-back caches. Prior work took an "eviction-focussed perspective", answering for each potential cache miss: May this miss evict a dirty cache line and thus cause a write back? We complement this approach by exploring a "store-focussed perspective", answering for each store: May this store dirtify a clean cache line and thus cause a write back later on?

Experimental evaluation demonstrates substantial precision improvements when both perspectives are combined. For most benchmarks, write-back caches are then preferable to write-through caches in terms of the computed WCET bounds.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases write-back caches, real-time systems, WCET analysis, cache analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.26

1 Introduction

The timely execution of programs is critical for hard real-time systems. Static worst-case execution time (WCET) analysis provides upper bounds on programs' execution times in all possible execution scenarios. These upper bounds can then be used to verify that all timing constraints of a given system are met prior to deployment in the field.

The execution time of a program heavily depends on the state of the underlying hardware platform, in particular on the state of caches, which are intended to bridge the gap between slow main memory and fast cores. WCET analysis has to precisely account for cache behavior to obtain useful time bounds.

Multiple parameters affect a cache's behavior and thus the latency of memory accesses, e.g. its capacity, associativity, block size, and replacement policy. A parameter that has so far received little attention in the literature is the write policy which determines the handling of write memory accesses. There are two common choices for the write policy: write through and write back. In a write-through cache, upon a store, the data is written to main memory directly. In a write-back cache, the data is only written to the cache and the corresponding

* This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Project PEP, and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

cache line is marked *dirty*. Once a dirty cache line is evicted from the cache its data is written back to main memory. In this way, multiple stores to the same cache line may be consolidated into a single costly main-memory access. Due to the possible performance benefits [17, 18], many embedded systems employ caches following the write-back policy, e.g. in the MPC603e [10] and the ARM946 [3] processor.

Write-back caches are commonly considered hard to analyze because write backs are decoupled in time from the corresponding stores in the program. Most literature on cache analysis is targeted at caches following the write-through policy. We are aware of only two exceptions: Alt et al. [1] and Sondag and Rajan [27] both introduce static analyses to safely approximate the set of dirty cache lines at each program point. Upon a potential cache miss, WCET analysis can use this information to determine whether or not the evicted cache line may be dirty and thus trigger a write back. As this approach tries to exclude potential write backs at cache evictions, we term this approach "eviction-focussed write-back analysis".

In this paper, we identify a complementary approach, which we term "store-focussed write-back analysis". The approach is based on two simple observations:

- each write back is preceded by a store to the cache line written back, and
- stores to dirty cache lines do not increase the number of write backs.

Thus we introduce an analysis that identifies and bounds the number of "dirtifying stores", i.e. stores to previously clean cache lines. We demonstrate at the hand of examples and later experimentally that the store-focussed approach is required to obtain good execution time bounds for caches following the write-back policy.

Our main contributions are the following:

- We introduce the "store-focussed write-back analysis" approach and the corresponding *dirtifying store* analysis, which is essential to turn write-back caches into a favorable choice for hard real-time systems.
- We present the first experimental evaluation of *both* eviction- and store-focussed write-back analyses. In particular, we demonstrate that with our new analysis write-back caches are preferable to write-through caches in terms of WCET bounds.

2 Background: Write-back Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* $\mathcal{B}$. Each block is cached as a whole in a cache line of the same size.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). For efficient look-up, each block can only be stored in a small number of cache lines known as a *cache set*. A subset of the bits of a memory block's address determines the cache set it maps to. The cache is partitioned into equally-sized cache sets. The size of a cache set in blocks is called the *associativity* k of the cache.

Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. Importantly, almost all replacement policies treat sets independently, so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies are least-recently-used (LRU), used, e.g., in various Freescale processors such as the MPC603e and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU; and first-in first-out (FIFO). In this article we focus

Listing 1 Motivation for Store-focussed Write-back Analysis

```
x = f(x);
sum = 0;
for (i=0; i<N; i++) {
    sum += arr[read_sensor()];
}
...
```

exclusively on LRU. The application of our ideas to other policies is left as future work. As the name suggests, LRU replaces the least-recently-used block upon misses.

LRU naturally gives rise to a notion of *ages* for memory blocks: The age of a block b is the number of pairwise different blocks that map to the same cache set as b that have been accessed since the last access to b. If a block has never been accessed, its age is k. Then, a block is cached if and only if its age is less than the cache's associativity k.

Given this notion of ages, the state of an LRU cache can be modeled by a mapping that assigns to each memory block its age, where ages are truncated at k, i.e., we do not distinguish ages of uncached blocks. So the set of cache states is $age : \mathcal{B} \to \{0, \ldots, k\}$. Then, the effect of an access to memory block x can be formalized as follows:

$$update(age, x) \triangleq \lambda b. \begin{cases} 0 & \text{if } x = b \\ age(b) & \text{else if } age(x) \leq age(b) \vee set(x) \neq set(b) \\ \min(k, age(b) + 1) & \text{else if } age(x) > age(b) \wedge set(x) = set(b) \end{cases} \quad (1)$$

where $set(x)$ denotes the cache set that x maps to.

There are several choices regarding the implementation of stores, determined by the answers to the following two questions:

1. When is the data written back to main memory?
   One option, known as *write through*, is to perform any store immediately in main memory. The other option, known as *write back* is to buffer the changes in the cache, marking the modified cache line as dirty. Upon an eviction of a dirty cache line its data is then written back to main memory.
2. What happens upon a write miss?
   If the memory block being modified is not contained in the cache, one can either bypass the cache and make the modification in main memory (*no write allocate*) or one can allocate a cache line and write to that line (*write allocate*).

Write-back caches usually employ *write allocate*, and write-through caches usually employ *no write allocate*. In the remainder of the paper we will only state whether a cache is write through or write back and assume the usual allocation policy.

For a write-through cache, the LRU update defined in (1) does not cover the *store miss* case; then the update is simply the identity function; assuming that the non write allocate policy is applied in write-through caches.

3 Motivating Examples

Consider the example program in Listing 1. For the sake of readability, we use C-style example programs, while the analysis is performed at the level of machine instructions. Assume that all variables, i.e., `x`, `sum`, and `i`, are kept in separate memory blocks rather than

Listing 2 Motivation for Eviction-focussed Write-back Analysis

```
if (...)
    x = 0;
if (...)
    x = 1;
b = c;
```

in registers, and that `N` is a constant held in a register. For simplicity, let us analyze the write-back behavior of this example on a fully-associative cache of size 4.

The address accessed by `arr[read_sensor()]` cannot be predicted statically as it depends on sensor readings, which only become available at runtime. In particular, it is impossible to determine whether the accessed memory block is cached or not, assuming the array is larger than the cache. Then, each of the array accesses may result in a miss and may thus potentially trigger a write back.

After the first access to the array within the loop, the variable `x` is stored in the least-recently-used cache line, as `sum`, `i`, and `arr[read_sensor()]` have been used more recently. Due to the store `x = f(x)` the cache line holding `x` is dirty. Depending on whether the following iterations access the same cache line as the first loop iteration, `x` is evicted or not. Applying the eviction-focussed approach [1, 27] it is impossible to exclude a write back in any of the loop's iterations but the first. Such an analysis would thus have to assume at least $\mathtt{N} - 1$ write backs.

Now let us adopt a store-focussed view: `x` is written to only once, and so there may be at most one corresponding write back. However, simply using the number of stores in the example program is also not beneficial: including the assignments to `i` and `sum` in the loop, there are $2\mathtt{N} + 2$ stores in the program, which is even greater than the number of write backs an eviction-focussed analysis would derive. Besides, simply counting the number of stores defeats the purpose of write-back caches, which is to consolidate multiple stores to the same cache line before writing the data back to main memory.

To account for this mechanism, we introduce the notion of *dirtifying stores*. A dirtifying store is a store to a clean cache line. The number of dirtifying stores is a bound on the number of write backs. In the example, the variable `sum` is first written to in line 2, turning its cache line dirty. All the stores to `sum` inside the loop are non-dirtifying: as `sum` is accessed in every loop iteration, it may never be evicted and thus it remains dirty throughout the entire loop execution. Similarly, the first store to `i` in the loop is dirtifying, but all subsequent stores are non-dirtifying, as `i` remains cached throughout.

We therefore conclude that as a consequence of this program's execution at most three write backs may occur, namely of `x` (committing the store from line 1), `i` (committing the stores in line 3) and `sum` (committing the stores in line 2 and 4). In Section 6, we introduce an approach to bound the number of dirtifying stores.

Although the store-focussed approach often works very well, it is not always superior to the eviction-focussed approach. Consider the example in Listing 2, which includes two potentially dirtifying stores to variable `x`. Assuming a fully-associative cache of size 2, an eviction-focussed approach may recognize that `x` can only be evicted when accessing `b` (following the access to `c`) and can therefore safely account for a single write back. For optimal results, both approaches should therefore be combined. In Section 5, we describe an eviction-focussed write-back analysis, including a dirtiness analysis, which is also required to precisely bound the number of dirtifying stores in Section 6.

4 Background: Static WCET and Cache Analysis

4.1 Static WCET Analysis

In this section, we briefly present a state-of-the-art approach to WCET analysis, which our write-back analysis is based upon. It consists of three phases: (1) value analysis, (2) microarchitectural analysis, and (3) path analysis. All analyses are performed on binary-level machine programs.

The first phase performs analyses that are independent of the underlying microarchitecture. Examples are value analyses such as constant propagation or interval analysis. The main purpose of this phase is to compute loop bounds and addresses of load and store instructions. This information is used in the following phases.

The second phase analyzes the program at the microarchitectural level. The microarchitectural analysis calculates a *state graph*. Each node in the state graph contains the state of the microarchitecture including the pipeline and the cache. Nodes are connected via edges, which represent the execution behavior of the system over time. Each edge is weighted with its transition time in clock cycles, which we refer to as $time(e)$. Additional weights may be defined if required, e.g. modeling that a loop back edge is taken, which can be constrained by a corresponding loop bound, or that a write back occurs when an edge is taken.

Unfortunately, enumerating all reachable concrete microarchitectural states is infeasible; there are simply too many. It is therefore necessary to construct the state graph using *abstract* states. Each abstract state corresponds to many concrete states at once, thereby reducing the state space. Such an abstraction comes with two functions: the *update* function and the *join* function.

- The *update* function computes the successor of an abstract state. For a correct analysis, this function has to be consistent with the concrete transition function.
- The *join* function merges two abstract states into one. The resulting state must represent all concrete states represented by its arguments, but it may contain more. The ensuing imprecision is the price for the reduced state space.

Due to the uncertainty within the abstract states, there might be multiple successor states, e.g. one for the cache hit case and one for the cache miss case. This uncertainty is represented by nondeterministic choices inside the graph, i.e. multiple edges originating from the same state.

Finally, in the path-analysis phase, the worst-case path through the state graph is determined. To do so, the state graph is encoded via linear constraints and integer linear programming (ILP) is used to determine the path through the state graph with the greatest execution time. More precisely, a *frequency variable* f_e is introduced for each edge e of the state graph, modeling how often edge e is taken in an execution. The structure of the graph is encoded via *constraints*, which restrict the valuations of the frequency variables to those corresponding to valid paths through the state graph: in particular, the sum of the frequencies of incoming edges needs to be equal to the sum of the frequencies of outgoing edges at each node of the state graph. The results of the loop bound analysis in the first phase are used to further restrict the possible paths via *loop bound constraints*. A bound on the program's execution time can then be obtained via the following objective function: $\max \sum_{\text{edge } e} time(e) \cdot f_e$.

Further constraints can be added to the ILP to exclude infeasible execution paths. We use such constraints in the store-focussed write-back analysis introduced in Section 6.

Listing 3 Motivation: Persistence analysis

```
for (int i=0; i<N; i++) {
    k = read_sensor();
    sum[k] = sum[k] + arr[k];
}
```

4.2 Cache Analysis

The goal of cache analysis is to statically prove that a memory access hits or misses the cache. Here, we limit our exposition to a brief summary of the cache analyses required in order to understand our work. A more complete and detailed overview can be found in [22].

May and Must Analysis

In order to predict cache hits and misses LRU cache analyses compute upper and lower bounds on the *ages* of memory blocks. The age of memory block b is the number of distinct memory blocks mapping to the same cache set as b accessed since the last access to b. A block is in the cache if and only if its age is less than the cache's associativity k.

Thus, upper bounds on ages may be used to predict cache hits and lower bounds may be used to predict cache misses. The two analyses computing upper and lower bounds on ages are commonly known as must and may analysis [1]. Abstract states for both analyses map blocks to their respective bounds: $\mathcal{S}_{must} = \mathcal{S}_{may} = \mathcal{B} \to \{0, \ldots, k\}$.

The two functions $update_{must}$ and $update_{may}$ define the successor of an abstract cache state when loading or storing to x. For reasons of brevity we only define $update_{must}$ here:

$$update_{must}(must, x) \triangleq \lambda b. \begin{cases} 0 & \text{if } x = b \\ must(b) & \text{else if } must(x) \leq must(b) \vee set(x) \neq set(b) \\ \min(k, must(b) + 1) & \text{else if } must(x) > must(b) \wedge set(x) = set(b) \end{cases}$$

where $set(x)$ denotes the cache set that x maps to.

As in the concrete case, for a write-through cache, the above update does not cover the *store miss* case; then the identity function is applied instead. Since the must analysis maintains upper bounds, the join function takes the maximum of both bounds: $join(S, T) = \lambda b.\ \max(S(b), T(b))$. Similarly the join for the may analysis takes the point-wise minimum of the bounds.

Persistence Analysis

May and must analysis try to classify memory accesses at a given program point as cache hits or cache misses under all circumstances. In some cases, such a classification is impossible even though the cache is highly effective. Consider the array *sum* in Listing 3. Assuming all memory blocks accessed inside the loop completely fit into the cache, none of the cache lines of *sum* can be evicted within this loop once they have been loaded. They *persist* in the cache. However, must analysis is unable to prove this assuming each call of `read_sensor` may deliver an arbitrary value within the bounds of the arrays: each access may be the first to its cache line.

One approach to persistence analysis is the *conflict-set* analysis (also called conflict counting in [6]). The conflict-set analysis simply accumulates all memory blocks that may

have been accessed. If all of these blocks simultaneously fit into the cache, then all of them can safely be classified as *persistent*. There are more sophisticated persistence analyses described in [6], however, they offer little additional precision in practice, and thus we make use of the simple conflict-set analysis in this work.

For the example above, the conflict-set analysis computes a conflict set containing all blocks of `sum` and `arr` as well as `i` and `k`. Since this set fits entirely into the cache, all accesses are classified as persistent.

If a memory block is persistent, accesses to this block may only result in a single cache miss during program execution. This property can be encoded as a linear constraint, limiting the execution paths explored to those in which the respective block causes at most one miss. If an entire array is persistent, a similar constraint can be formulated to capture that the sum of the misses upon accesses to the array is bounded by the number of cache lines the array occupies.

Applying persistence analysis globally to large programs is bound to fail as memory blocks are rarely persistent throughout the entire program execution. Instead, persistence analysis is usually performed on smaller, contiguous parts of the program called *persistence scopes*. A common choice for persistence scopes, which we adopt in our experiments, are loops – they are executed more than once and they often reuse memory blocks across iterations. Persistence scopes can be nested, such that a block might be persistent in an inner loop but not persistent in the surrounding loop.

5 Eviction-focussed Write-back Analysis

In this section, we describe a simple eviction-focussed write-back analysis. The aim of this analysis is to determine for each potential cache miss in the program whether the cache miss may evict a dirty cache line and thus cause a write back.

Without further information, the analysis has to assume that every cache miss evicts a dirty cache line. To provide useful bounds, the write-back analysis has to track which memory blocks may be dirty. If the analysis can prove that all of the memory blocks that a cache miss might evict are clean, then no write back may occur.

5.1 Formalizing the Behavior of Write-back Caches

Before defining an abstraction for tracking the dirtiness of memory blocks, let us formalize the concrete behavior of a write-back cache. To this end, the dirtiness of memory block is represented by a predicate $dirty : \mathcal{B} \to \{C, D\}$, where C stands for "clean" and D for "dirty". Concrete cache states S then are pairs consisting of $S.age : \mathcal{B} \to \{0, \ldots, k\}$ and $S.dirty : \mathcal{B} \to \{C, D\}$, which together map each block to its age and its dirtiness status.

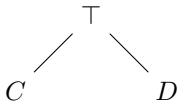
The update of ages has been defined in (1). The update of the dirtiness predicate depends on whether the access is a load or a store. Let us first consider the load case:

$$update_{load}(S.dirty, x) \triangleq \lambda b. \begin{cases} C & \text{if } evicts(S.age, x, b) \\ S.dirty(b) & \text{otherwise} \end{cases} \tag{2}$$

where $evicts(S.age, x, b)$, defined below, captures that the access to x causes the eviction of memory block b. If the access to x causes the eviction of block b (case 1), then b becomes clean. Otherwise, b's dirtiness remains the same (case 2).

The access to x evicts memory block b from the cache if they map to the same cache set, the access to x causes a miss, and b is evicted:

$$evicts(S.age, x, b) \triangleq set(x) = set(b) \wedge S.age(x) = k \wedge S.age(b) = k - 1.$$



⊔	C	D	⊤
C	C	⊤	⊤
D	⊤	D	⊤
⊤	⊤	⊤	⊤

Figure 1 Hasse diagram of the partial order on dirtiness states and the corresponding join function on dirtiness states.

In case of a store, the block that is written to becomes dirty. Other blocks may become clean if they get evicted, as modeled by the load update:

$$update_{store}(S.dirty, x) \triangleq \lambda b. \begin{cases} D & \text{if } x = b, \\ update_{load}(S.dirty, x)(b) & \text{otherwise}. \end{cases} \tag{3}$$

An access to block x causes a write back if and only if it evicts a dirty block:

$$writeback(S, x) \triangleq \exists b \in \mathcal{B}.\ S.dirty(b) = D \wedge evicts(S.age, x, b)\,.$$

5.2 Dirtiness Analysis

Dirtiness analysis [1, 27] tracks the dirtiness of memory blocks. During analysis, each block can be in one of three dirtiness states:

1. clean (C), meaning the block is definitely clean,
2. dirty (D), meaning the block is definitely dirty, and
3. unknown ($\top$), meaning the block might be either clean or dirty.

The three dirtiness states are partially ordered according to the Hasse diagram in Figure 1. The domain of the overall dirtiness analysis is thus $\mathcal{B} \to \{C, D, \top\}$ mapping each block to its dirtiness state.

To specify the abstract update function of the dirtiness analysis, we need to know when a memory block *might get evicted* from the cache and when a block *has definitely been evicted* from the cache. We define both predicates based on the information from the must and the may cache analysis. The must analysis tells us the *earliest* point in time when a block may be evicted, while the may analysis tells us the *latest* point in time when a block may be evicted. If there is any uncertainty about a block's exact age we obtain an interval of points in time at which an eviction may happen.

We use $S.may$ and $S.must$ to refer to the state of the may and must analysis *before* the current update. The predicate $may\text{-}evict(S, x, b)$ determines whether accessing x might evict b from cache state S:

$$\begin{aligned} may\text{-}evict(S, x, b) \triangleq\ & set(x) = set(b) \wedge S.must(x) = k \\ & \wedge S.may(b) < k \wedge update_{must}(S.must, x)(b) = k\,. \end{aligned}$$

Paraphrasing the formula above, block b might be evicted, if the access to x may result in a miss ($S.must(x) = k$) and b may have been cached prior to the access ($S.may(b) < k$), and b may be out of the cache after the access ($update_{must}(S.must, x)(b) = k$).

The predicate $\mathit{evicted}(S, x, b)$ determines whether block b has definitely been evicted from the cache after accessing x[1]:

$$\mathit{evicted}(S, x, b) \triangleq \mathit{update}_{\mathit{may}}(S.\mathit{may}, x)(b) = k\,.$$

Using the above predicates, we can define the abstract dirtiness update function for loads:

$$\widehat{\mathit{update}_{\mathit{load}}}(S, x) \triangleq \lambda b.\ \begin{cases} C & \text{if } \mathit{evicted}(S, x, b), \\ S(b) \sqcup C & \text{else if } \mathit{may\text{-}evict}(S, x, b), \\ S(b) & \text{otherwise}\,. \end{cases} \tag{4}$$

If a block must have been evicted (case 1, above) it is definitely clean. If it may have been evicted, it may be clean (case 2) after the access. If it was clean before $S(b) \sqcup C = C$, otherwise $S(b) \sqcup C = \top$ and the dirtiness status of b is unknown. If b may not have been evicted by the access to x, then its dirtiness status does not change (case 3).

Here, it becomes evident that the precision of the dirtiness analysis depends on the precision of may and must analysis: The more uncertainty there is about when blocks are evicted, due to uncertainty about blocks' ages, the more uncertainty there is about their dirtiness status.

Upon stores, dirtiness is updated as follows:

$$\widehat{\mathit{update}_{\mathit{store}}}(S, x) \triangleq \lambda b.\ \begin{cases} D & \text{if } x = b, \\ \widehat{\mathit{update}_{\mathit{load}}}(S, x)(b) & \text{otherwise}\,. \end{cases} \tag{5}$$

The block that is stored becomes dirty (case 1). The effect on other blocks is the same as in case of a load, and thus the update function for loads can be applied for those blocks (case 2). This closely matches the concrete update defined in (3).

Dirtiness analysis states are joined by applying the join depicted in Figure 1 to the dirtiness of each memory block:

$$\mathit{join}(S.\mathit{dirty}, T.\mathit{dirty}) \triangleq \lambda b.\ (S.\mathit{dirty}(b) \sqcup T.\mathit{dirty}(b)).$$

Based on a dirtiness analysis state we define the *may-wb* predicate, which determines whether an access might induce a write back:

$$\mathit{may\text{-}wb}(S, x) \triangleq \exists b \in \mathcal{B}.\ S.\mathit{dirty}(b) \neq C \wedge \mathit{may\text{-}evict}(S, x, b)\,. \tag{6}$$

This is the case, if a block may be evicted that is not guaranteed to be clean.

Initial State

The results produced by our WCET analysis may serve as inputs to a later schedulability analysis that checks whether a set of tasks can be scheduled on a given platform. Such an analysis for fixed-priority scheduling and platforms featuring write-back caches has recently been proposed by Davis et al. [7]. The analysis by Davis et al. accounts for the effects of initially dirty cache blocks and additional effects due to preemptions. Using this schedulability analysis, the WCET analysis can safely assume an initially clean cache, i.e. the proposed dirtiness analysis may start from the initial state $\lambda b.\, C$. If required, a potentially dirty initial cache can also be modeled by the initial state $\lambda b.\, \top$.

[1] We do not call this predicate *must-evict*, because it also captures cases where b may have been evicted earlier, and is thus not guaranteed to be evicted by the current access.

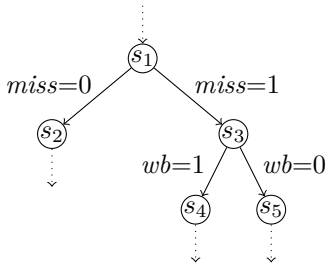


Figure 2 State graph upon cache-state uncertainty.

5.3 Integration into Microarchitectural Analysis

Microarchitectural analysis constructs the state graph that represents the cycle-level execution behavior of a program, as described in Section 4.1. The nodes in the state graph are abstract microarchitectural states. These abstract states encompass the state of the pipeline, the state of the memory controller, and the state of the caches including the abstract dirtiness state introduced above.

Assume memory block x is accessed during a cycle transition from abstract state S. If the may and must analysis cannot classify the access as cache hit or miss the analysis creates a successor state for both cases, one for the hit and one for the miss case. In general, both cases need to be considered due to *timing anomalies* [21, 25].

If a write-back cache is employed, the cache line loaded upon the cache miss will evict another cache line. The evicted cache line may be either dirty, which causes a write-back, or clean. Naively, the microarchitectural analysis would follow both cases. However, the analysis can use the *may-wb* predicate to rule out that a write back happens. If *may-wb* is false, no write back can happen and only one case needs to be considered. If *may-wb* is true, again both cases need to be considered due to the possibility of timing anomalies.

An excerpt of a state graph that arises due to cache and dirtiness uncertainty is depicted in Figure 2. After a few cycle transitions, the successors of s_2, s_4, and s_5 are likely to converge to a similar state. In that case, these successor states are joined into a single analysis state to keep the complexity of the microarchitectural analysis at an acceptable level.

6 Store-focussed Write-back Analysis

As we have shown by example in Section 3, it can be beneficial to bound the number of write backs by analyzing the stores that may occur during program execution rather than the evictions, which eventually trigger write backs. While stores are trivial to locate, the exact position of the write back of a particular dirty memory block depends on the cache state and thereby on the history of the program. Uncertainty about the program flow therefore affects eviction-focussed analyses more than store-focussed ones.

The simplest store-focussed analysis only exploits the fact that a write-back cache defers stores but never generate additional ones. The number of write backs is therefore bounded by the number of stores.

This property, which we call the *store bound*, can be expressed in the path analysis ILP formulation. As can be seen in Figure 2, microarchitectural analysis annotates transitions in the state graph with multiple edge weights, including $wb(e)$ and $st(e)$. $wb(e)$ denotes the number of write backs on edge e in the state graph, and $st(e)$ denotes the number of stores on edge e. Usually, these edge weights are either 0 or 1 for a given edge, but in principle

Listing 4 Listing 1 revisited. The dirtiness analysis state is shown on the right. Dirtifying stores are underlined.

```
x = f(x);                                          x ↦ D, sum ↦ C, i ↦ C
sum = 0;                                           x ↦ D, sum ↦ D, i ↦ C
for (i=0; i<N; i++) {
    sum += arr[read_sensor()];                     x ↦ ⊤, sum ↦ D, i ↦ D
}
...
```

they can be larger if several consecutive edges are merged for efficiency reasons. Given these edge weights, the following linear constraint corresponds exactly to the fact that the number of stores bounds the number of write backs:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} st(e) \cdot f_e, \tag{7}$$

where f_e is the frequency variable that captures how often edge e is taken.

With this constraint, path analysis implicitly only considers executions in which the number of write backs is bounded by the number of stores. We note, however, that the constraint does not exclude those infeasible executions in which write backs occur *before* stores. This limitation appears difficult to eliminate without generating a much larger ILP.

6.1 Dirtifying Stores

Simply bounding the number of write backs by the number of stores may yield unsatisfactory results. In particular, the store bound never predicts fewer memory accesses than would occur in a system with a write-through cache. The crucial advantage of write-back caches is the ability to consolidate multiple stores into a single write back. An effective analysis has to capture this behavior. Approaching the problem from a store-focussed perspective, this means that a write-back analysis has to recognize whether a store targets a clean cache line. We call such stores *dirtifying*, since they cause a cache line to become dirty.

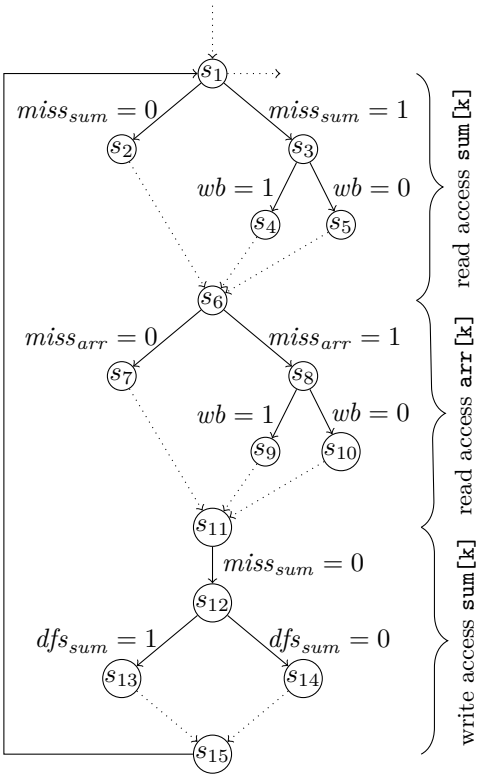
The dirtifying store analysis is based on the observation that only dirtifying stores may cause write backs. After the first store to a cache line, this line has to be written back no matter what, so additional stores to the cache line do not increase the future number of write backs. We use the dirtiness analysis described in Section 5.2 to identify definitely dirty cache lines, and refine the $st(e)$ weight to the dirtifying store weight $dfs(e)$ defined as follows:

$$dfs(e) \triangleq |\{b \in stores(e) \mid S.dirty(b) \neq D\}|$$

where $stores(e)$ is the set of memory blocks targeted by stores on e and S is the abstract cache state at the source of e. By construction $dfs(e) \leq st(e)$. Given $dfs(e)$, the constraint in Equation 7 can then be improved as follows:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} dfs(e) \cdot f_e\,. \tag{8}$$

To understand the effect of this analysis consider Listing 4. The dirtiness analysis classifies `x` and `sum` as definitely dirty in the two lines leading up to the loop. Variable `i` is classified as dirty following the loop initialization. Both `i` and `sum` are guaranteed to remain in the cache, and so remain definitely dirty during the entire loop execution. Thus the stores to



Persistence constraints:

$$\sum_{\text{edge } e} miss_{sum}(e) \cdot f_e \leq \#cachelines_{sum}$$

$$\sum_{\text{edge } e} miss_{arr}(e) \cdot f_e \leq \#cachelines_{arr}$$

Write-back constraint:

$$\sum_{\text{edge } e} wb(e) \cdot f_e \leq \sum_{\text{edge } e} \left(\sum_{\text{variable } v} dfs_v(e) \right) \cdot f_e$$

Persistent dirtifying store constraint:

$$\sum_{\text{edge } e} dfs_{sum}(e) \cdot f_e \leq \#cachelines_{sum}$$

Figure 3 Simplified state graph for Listing 3 and generated constraints.

`i` and `sum` inside the loop are *non*-dirtifying. The loop itself might evict `x`, so during the loop `x`'s dirtiness is classified as $\top$. Since all accesses to `arr` may evict `x`, each access to `arr` generates a write-back edge. Enforcing the store bound yields a vacuous constraint, since there are $2\mathtt{N} + 2$ stores in the program but only N potential write backs. The dirtifying store analysis, however, recognizes three dirtifying stores (one each to `x`, `sum`, and `i`) and therefore only allows for three write backs as a consequence of the code visible in the listing.

6.2 Stores to Persistent Blocks

The analysis of dirtifying store described above relies on the must-dirty information obtained from the dirtiness analysis. Consequently, the approach suffers from the same shortcomings as the must analysis does in predicting cache hits. Reconsider Listing 3, which motivated persistence analysis. Since the precise blocks accessed by the store to `sum` are unknown at analysis time, no block belonging to `sum` is guaranteed to be cached in any loop iteration or to have been written to. Consequently, the analysis classifies the dirtiness of the array's memory blocks as $\top$ and therefore cannot exclude any of the stores to be dirtifying. As a consequence, the analysis must account for N write backs due to the stores to `sum`.

Persistence analysis exists to remedy this shortcoming of the must analysis; assuming that all blocks accessed in the loop together fit into the cache, persistence analysis proves that `sum` and `arr` are never evicted and therefore cause at most one cache miss for each cache line in `sum` and `arr`. A similar argument holds for the number of dirtifying stores: If `sum` is never evicted, there can be at most one dirtifying store to each cache line of `sum`. For every persistence constraint, we therefore also generate a corresponding constraint bounding

the number of dirtifying stores. Instead of N potential write backs, the path analysis now accounts for at most as many write backs as there are cache lines spanned by `sum`.

Implementing such a persistence-like bound requires dedicated edges for the dirtifying and the non-dirtifying case within the state graph. Figure 3 shows a simplified version of the state graph produced by the microarchitectural analysis for the program in Listing 3. For the sake of readability, we only show the relevant abstract states and relevant, non-zero edge weights. The edge weight $miss_v(e)$ denotes the number of misses to variable v on edge e, $dfs_v(e)$ the number of dirtifying stores to variable v, and $wb(e)$ the number of dirty cache lines written back on edge e. The constant $\#cachelines_v$ refers to the number of cache lines that variable v maps to, which is 1 for basic types such as integers, and may be larger for arrays, depending on the number of array elements and the size of the base type. Next to the state graph, we give the linear constraints obtained from persistence analysis and our store-focused write-back analysis.

Initial State

In the presented analysis, we again assumed an initially clean cache. The analysis can be adjusted to correctly account for initial unknown dirtiness. In that case, one needs to add the number of lines in the cache to the right-hand side of the constraint in Equations 7 and 8.

7 Experimental Evaluation

7.1 Executive Summary

In this section, we evaluate our write-back analysis. First, we determine its overall effectiveness by comparing WCET bounds for a system with a write-back cache (WB) with those obtained for an otherwise equivalent system featuring a write-through cache (WT). In many cases, WCET bounds for the write-back cache are more than 15% lower, making write-back caches not only a feasible but also a desirable choice in hard real-time systems.

Second, we evaluate the individual components of our write-back analysis. We demonstrate that all components presented in this work improve the analysis. We also evaluate a pure eviction-focussed analysis, following Alt et al.'s dirtiness analysis. We observe that it is ineffective in most benchmarks. In the few cases in which it is effective, however, it yields significant improvements over a pure store-focussed analysis.

Third, we evaluate the analysis cost of our analysis for write-back caches. We show that the analysis comes at reasonable cost in terms of both analysis time and memory consumption. Often it even turns out to be cheaper than an analysis for a write-through cache.

Finally, we evaluate how the memory latency affects write-back performance. Since a write backs transfers a whole cache line, while a direct store to memory only transfers a single word, write backs usually take slightly longer. We identify how much longer a write back may take before write-back caches cease to be profitable.

7.2 Experimental Setup

We base our evaluation on the Mälardalen benchmark suite [11]. Due to restrictions of our analyzer, we exclude those benchmarks that do not compile, have external function calls, or use recursion. We additionally generated seven benchmarks with *SCADE* [26], an industry-strength commercial model-based design tool.

We apply our analysis to a processor with a classic 5-stage in-order pipeline that supports a subset of the ARM instruction set architecture. The modeled processor features separate

data and instruction caches. Each of them is a 2-way LRU cache with 16-byte line size and 32 cache sets. As the benchmarks have a small memory footprint, we consequently choose such a small cache size. Finally, the processor contains a single-entry write buffer that buffers stores on their way to main memory. The processor therefore does not wait for stores to complete unless another memory operation is already pending.

The parameters of main memory, such as the latency of an access, are an important factor in our evaluation. While a write-through cache transfers one word (4 bytes) during a store, a write-back cache writes back an entire cache line (in our case 16 bytes). Memory chips support accesses in *burst* mode for this purpose: multiple words are transferred in a single access, one word per cycle. Thus, a line-wide access is slightly more expensive than a word-wide access, but a lot cheaper than accessing all words of a line individually. We choose 10 cycles as the latency of the first word accessed and 1 cycle for each following word in burst mode. These are realistic timings for DRAM chips, e.g. the Micron MT46V16M16 Automotive DDR SDRAM [24]. At the start of an access the previously open row is closed ($t_{RP} = 15ns$), the new row is opened ($t_{RCD} = 15ns$), and the respective column is accessed ($CL = 3$ cycles). For a clock rate of 200 MHz, this amounts to 9 cycles needed to setup the access. Each consecutive word within the burst access is then transferred in one cycle, resulting in the above timing.

All evaluations were performed using our own timing analyzer *llvmta* first used and described in [16]. The analyzer operates on the binary-level program representation generated by the LLVM compiler infrastructure. We employ trace partitioning [23] to perform context-sensitive analyses. We choose contexts to distinguish different call sites of a function and to peel the first iteration of each loop. This is necessary to obtain useful must- and may cache information. Our address analysis determines intervals of potentially accessed addresses for each load and store instruction. We make use of the LLVM-internal scalar evolution analysis to obtain loop bounds.

Since our analyzer cannot handle some of the more involved features used by the LLVM optimizer, all benchmarks are compiled without optimization. This is so far a common choice for safety-critical systems [9]. However, the lack of efficient register allocation frequently causes needless spills and reloads. While a write-back cache can handle both accesses inside the cache, a write-through cache accesses memory on each register spill. Thus, it is not obvious whether our evaluation results also apply to highly optimized machine code.

7.3 Write-through versus Write-back Caches

Table 1 shows the WCET bounds obtained for all benchmarks for a system with unblocked stores (i.e. with a write buffer, which is the standard configuration) and for a system with blocked stores. These results are summarized in Figure 4 via two histograms: both histograms depict the number of benchmarks that fall into a particular bin of ratios between the WCET under write through (WT) and write back (WB). The histogram on the left corresponds to the case with unblocked stores, while the histogram on the right corresponds to the case with blocked stores. Benchmarks on the left of the line at 1.0 have smaller WCET bounds under WB than under WT, and vice versa benchmarks on the right of the line have larger WCET bounds under WB than under WT. A logarithmic scale is applied on the x-axis as the values are ratios, and so a ratio of 2.0 is at the same distance to 1.0 as a ratio of 0.5 is in the opposite direction.

The first observation is that WB is preferable to WT on most benchmarks (32/36 and 34/36, respectively) in both cases. The second observation is that the ratio WB/WT is usually smaller in case of blocked stores than it is in case of unblocked stores. A larger

Table 1 WCET bounds (in 1000 cycles) for write-through and write-back caches with blocked and unblocked stores and ratios between the WCET bounds in the two cases. The memory share is an estimate on the time the write-back system spends accessing memory. WB free is the write-back WCET assuming write backs take no time.

		Blocked stores			Unblocked stores				Unblocked stores	
	Benchmark	WT	WB	$\frac{WB}{WT}$	WT	WB	$\frac{WB}{WT}$	D-cache misses	Mem. share	$\frac{WB\ \text{free}}{WB}$
Mälardalen	adpcm	1453	858	59%	1018	857	84%	1186	4%	99%
	bs	2	2	67%	2	2	76%	13	31%	100%
	bsort100	1911	1102	58%	1264	1102	87%	58	0%	100%
	cnt	52	31	58%	40	30	76%	64	6%	98%
	compress	360	341	95%	316	330	104%	6296	50%	83%
	crc	767	464	61%	532	449	84%	1875	11%	95%
	expint	270	170	63%	174	170	98%	9	0%	100%
	fdct	10	11	114%	9	11	127%	325	82%	81%
	fft1	78	37	46%	59	37	62%	39	3%	100%
	fibcall	5	3	54%	4	3	76%	3	3%	100%
	fir	3940	2738	69%	3437	2652	77%	79807	78%	89%
	insertsort	8	5	57%	6	5	86%	4	2%	100%
	janne-complex	19	12	63%	13	12	91%	4	1%	100%
	jfdctint	17	15	90%	13	14	115%	287	54%	85%
	lcdnum	4	3	79%	3	3	88%	23	23%	96%
	lms	4818	3403	71%	3716	3376	91%	23990	18%	97%
	ludcmp	110	96	87%	99	95	96%	3521	97%	88%
	matmult	1543	1615	105%	1348	1547	115%	57213	96%	85%
	minver	39	28	73%	32	28	87%	359	34%	89%
	ndes	529	325	61%	414	319	77%	2796	23%	95%
	ns	105	84	80%	86	84	98%	791	25%	100%
	nsichneu	104	100	96%	99	97	98%	739	20%	92%
	prime	82	56	68%	56	56	100%	5	0%	100%
	qsort-exam	100	67	67%	67	67	100%	371	14%	96%
	qurt	29	16	54%	22	16	70%	41	7%	100%
	select	53	44	83%	47	43	93%	613	37%	91%
	sqrt	9	7	71%	8	7	83%	6	3%	100%
	statemate	21	17	82%	20	17	85%	26	4%	100%
	ud	63	53	85%	53	53	100%	1468	73%	92%
	geo. mean			71%			89%			95%
SCADE	cruise-control	162	122	75%	158	122	77%	94	2%	100%
	digital-stopwatch	3031	1793	59%	2321	1772	76%	5354	8%	97%
	es-lift	160	133	83%	155	132	85%	1773	35%	98%
	flight-control	3608	2271	63%	2951	2257	76%	9216	11%	96%
	pilot	226	192	85%	205	189	92%	3488	48%	89%
	roboDog	440	407	93%	424	402	95%	6587	43%	90%
	trolleybus	1150	1022	89%	1127	1005	89%	18736	48%	89%
	geo. mean			77%			84%			94%
	geo. mean (Mäl. + SCADE)			72%			88%			95%

number of store accesses to main memory makes write buffers more profitable. Thus, systems with write-back caches profit less from the additional write buffer because they write less often to main memory than write-through caches. As write buffers are common in modern processors, this demonstrates that it is essential to account for their presence for a fair comparison between the two write policies.

7.4 Analysis Precision: Decomposition into Contributions

We now evaluate the effects of the individual analysis components. Starting at the naive analysis that assumes a write back on every cache miss, Figure 5 shows the incremental improvement up to the final analysis.

As predicted in Section 6, the store bound has next to no effect on the WCET bound. Recognizing dirtifying stores is required to achieve appreciable bound reductions, as can be

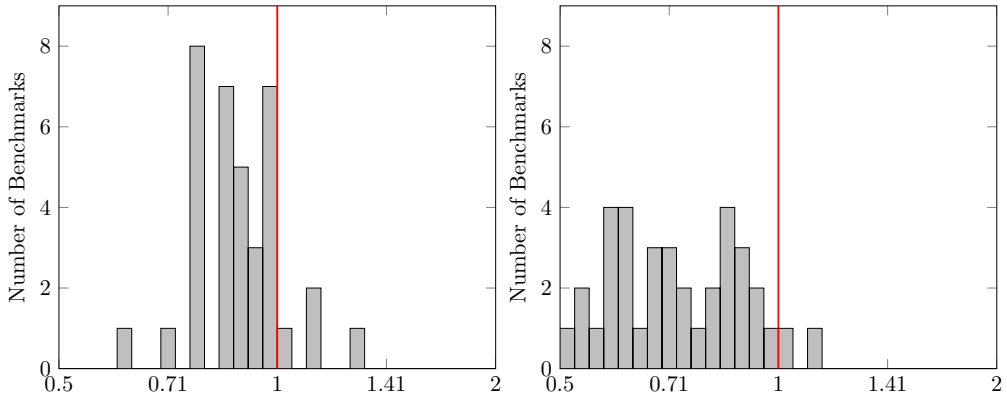
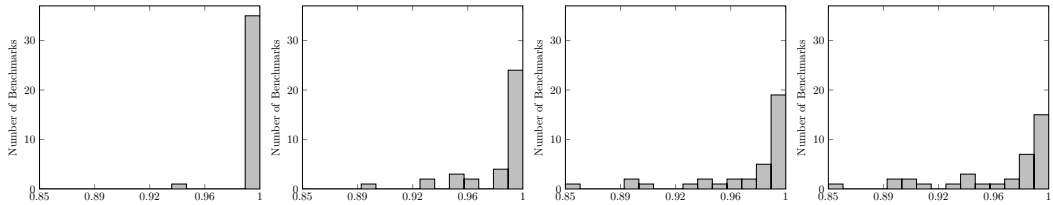


Figure 4 Histogram of ratios of WCET bounds compared to the write-through system. Left: Unblocked stores via a single-entry write buffer. Right: Blocked stores and thus no write buffer.



(a) Store Bound **(b)** + Dirtifying stores **(c)** + Store-persistence **(d)** + Eviction analysis

Figure 5 Histogram of ratios of WCET bounds compared to the naive analysis that assumes a write back on every cache miss.

observed in part (b) of Figure 5. Most benchmarks are still unaffected by the analysis, though. We conjecture that non-dirtifying stores often occur in loops, where multiple iterations target the same cache line. The regular dirtifying store analysis fails to recognize this unless the accessed block is known precisely. We therefore developed store-persistence constraints, which recognize stores to persistent blocks (which may be part of an array) and bound the number of dirtifying stores appropriately. Figure 5(c) indicates that the conjecture about non-dirtifying stores is correct; adding the persistence constraints improves the results significantly and achieves the greatest speedup, e.g. reducing *ludcmp*'s bound by 15%.

With Figure 5(c) we have reached the the final result for a pure store-focussed analysis. We propose to combine the store-focussed analysis with an eviction-focussed analysis. Figure 6 shows the reason: the two approaches are orthogonal, i.e. programs are usually either suited to one or the other. This effect can also be observed in Figure 5(d): eviction-focussed analysis yields crucial bound improvements on a few benchmarks but has no effect on the majority of benchmarks. Since one needs a microarchitectural dirtiness analysis to implement the dirtifying store bound anyway, performing an eviction-based analysis has only a negligible effect on the analysis runtime.

Surprisingly, about half of the benchmarks in Figure 5(d) are not affected by either analysis. One reason is that many benchmarks spend little time accessing memory; write-back analysis therefore cannot possibly have a large effect on the overall WCET. We provide an estimate of the amount of time a benchmarks spends accessing memory in the *Memory share* column of Table 1. This amount of time is estimated by multiplying the number of data cache misses with the latency of *two* memory accesses (i.e. a cache miss and a write back). Dividing this time by the WCET yields the memory share.

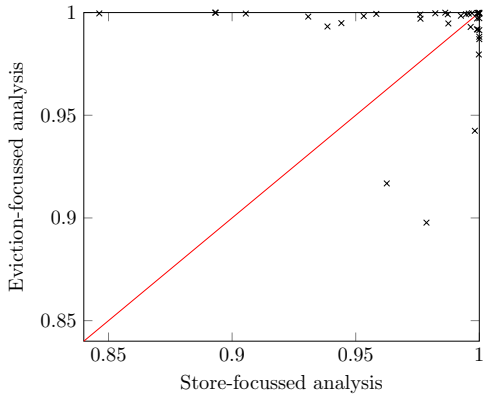


Figure 6 Comparison between the store-focussed analysis and the eviction-focussed analysis. The WCETs are normalized to the naive write-back analysis.

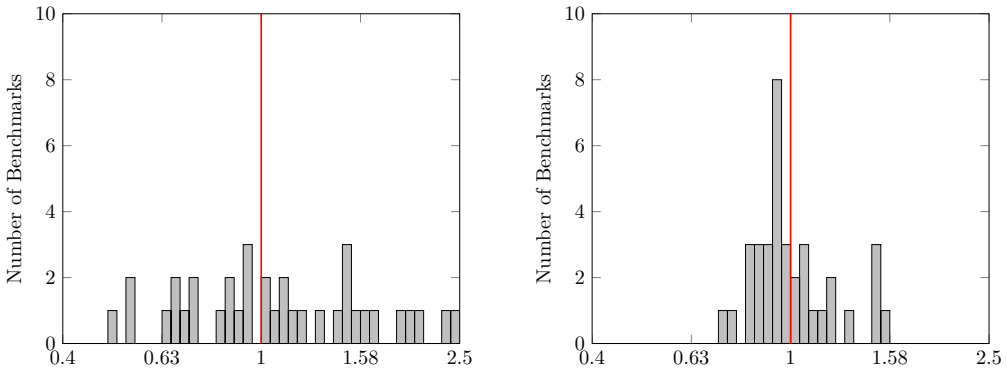


Figure 7 Histogram of ratio of runtimes (left) and memory consumptions (right) of the write-back analysis relative to the write-through analysis.

Consider for example *bsort100*. As one can see in Table 1, it spends less than 0.5% of its runtime accessing memory, *even if all cache misses are write backs*. This means that even a perfect write-back analysis would have no noticeable impact on *bsort100*'s WCET bound. In total 10 benchmarks spend $\leq 3\%$ of their runtime accessing memory.

This property of benchmarks can also be observed by considering WCET bounds obtained under the assumption that write backs are free (called "WB free" in the table). This value represents a fundamental lower bound on what an analysis can achieve by proving write backs impossible, if the number of cache misses accounted for remains constant. Table 1 shows that 13 benchmarks are already within 3% of this boundary. In total, our analysis achieves 95% of this boundary on average. Significant further improvements can therefore only stem from improving the underlying cache analysis, thereby reducing the number of cache misses accounted for.

7.5 Analysis Performance

Although analysis performance has not been a major goal in this work, it is important that the analysis is not prohibitively expensive. Figure 7 shows a histogram of the ratio of analysis runtime and memory consumption for the analysis of the write-back cache compared to the analysis of the write-through cache for each of the 36 benchmarks. The results vary widely: some programs take half the analysis time, some double analysis time. This may be

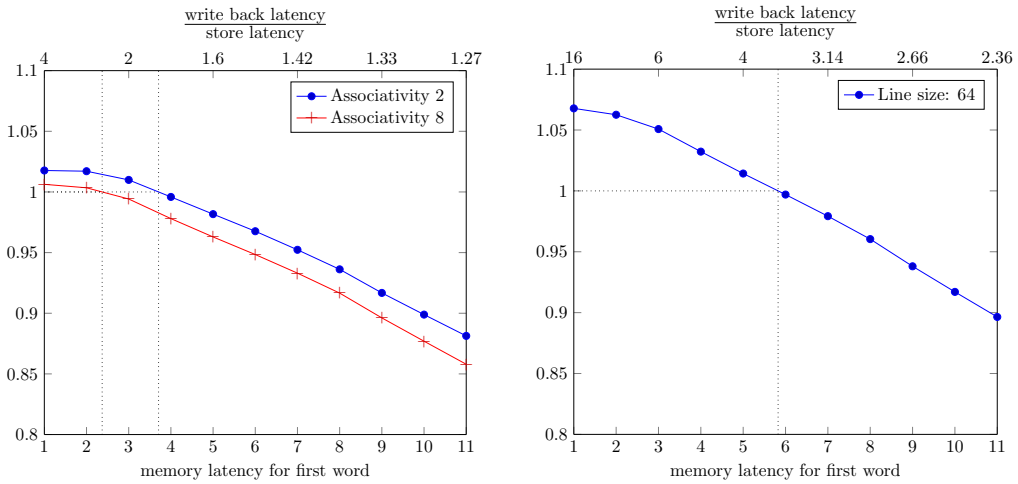


Figure 8 Geometric mean of WB/WT ratios for different latencies to access the first word of a cacheline. Each consecutive word always requires one additional cycle.

explained by two opposing effects: (i) On the one hand, it seems natural that the analysis of write-back caches is more expensive: Besides the increased size of abstract cache states due to dirtiness information, the microarchitectural analysis performs additional splits due to uncertainty about whether a write back happens or not. (ii) On the other hand, write-back caches reduce execution times which manifest in smaller state graphs with fewer states to explore for microarchitectural analysis whenever the eviction-focussed analysis is successful in excluding writebacks. All in all, we conclude that the write-back analysis comes at a reasonable cost.

7.6 Influence of Memory Latency on Write-back Execution Times

Recall our cache and memory parameters: cache line size of 16 bytes, i.e., 4 words of 4 bytes, a 10-cycle latency for the first word accessed, and a 1-cycle burst latency for consecutive words within a burst access. A write back then costs $\frac{13}{10} = 1.3$ times as much as a write-through store. On average, programs should therefore perform 0.3 stores on each dirty cache line before its eviction to compensate for the increased cost.

Clearly the profitability of the write-back cache crucially depends on these memory timing parameters. We therefore consider different latency scenarios to see how the picture might change for a main memory with different latency characteristics. To this end, we vary the initial latency, i.e. the latency to access the first word within a burst access from 1 to 11. The burst latency, i.e. the latency to access additional consecutive words, is kept constant at one additional cycle. An initial latency of 1 models the case of a pure random-access memory, where the latency of each word is the same. The higher the initial latency, the smaller the gap between the latency of a write back and the latency of a write-through store. We used an initial latency of 10 in all other experiments in this paper because it is realistic for modern embedded memory as discussed in Section 7.2.

Figure 8 shows the geometric mean of the WB/WT ratios for different initial latencies. We consider three cache configurations: (1) the standard two-way set-associative 1 KiB cache configuration described in Section 7.2 (“Associativity 2” in the figure), (2) an eight-way set-associative 4 KiB cache (“Associativity 8”), and (3) a two-way set-associative 4 KiB cache with a line size of 64 bytes (“Line size: 64”).

It is informative to check where the curves reach a ratio of 1, the break-even point between write back and write through. In the standard configuration write back is profitable for initial latencies above around 3.7. A slightly lower value is obtained for the larger cache with higher associativity (beneficial above 2.4). In case of a line size of 64 bytes the break-even point is around 5.8. Modern memories should lie well to the right of each of these points, rendering write-back caches profitable.

8 Related Work

In this section, we discuss related work on WCET and response-time analysis for systems with instruction and data caches; in particular work targeting write-back caches. All approaches described below assume caches with LRU replacement.

Analysis for write-back caches inherits all the challenges imposed by data-cache analysis compared with instruction-cache analysis. The main additional difficulty of data-cache analysis is obtaining information about the accessed addresses. While addresses of instructions inside a binary are easy to obtain, the addresses of data accesses can often not be pinned down to a single value. Examples include array accesses within a loop or input-dependent accesses resulting in a range of possibly accessed addresses.

Alt et al. [1] first introduced cache analysis based on abstract interpretation. They proposed must and may analysis to classify accesses as always hit/miss. They considered data cache analysis only for scalar accesses whose addresses could be precisely determined. For write-back caches, they extend the may analysis to track whether a block may be dirty. Based on this may-dirtiness information they locally exclude a write back if only clean blocks may get evicted. No experimental results concerning the proposed write-back analysis are given in Alt et al. [1] nor in the subsequent journal paper by Ferdinand and Wilhelm [8].

In [28], White et al. extend their prior work on static cache simulation to data caches. In cache simulation, they use abstract cache states, comparable to may-cache states, and (post-)dominator information to derive always hit/miss and first hit/miss classifications.

Ferdinand and Wilhelm [8] present a persistence analysis to improve the analysis of data caches. A memory block is deemed persistent if it cannot be evicted once it is loaded into the cache. The persistence analysis can handle ranges of possibly accessed addresses.

Huynh et al. [15] introduce the notion of temporal scope to improve data cache persistence analysis. The temporal scope of a memory block denotes the loop iterations in which the block might be accessed. Memory blocks with non-overlapping temporal scopes can thus not conflict within the persistence analysis. Their persistence analysis could be used to further increase the precision of our approach. It is, however, not obvious how widely applicable it is. In addition, Huynh et al. fix a problem in the original persistence analysis by Ferdinand and Wilhelm. In the same year, Cullmann [5] also provided a corrected persistence analysis. Later, Cullmann [6] describes the conflict-set analysis (termed "conflict counting" analysis there), which we use as a persistence analysis in this paper.

Sondag and Rajan [27] propose an analysis of multi-level caches and contribute the notion of *live caches* to reason about cache blocks that must be in one of the cache levels but are not guaranteed to be in any particular one. Analyzing write backs from the L1 cache to the L2 cache is necessary to correctly model the behavior of the L2 cache. Their eviction-focussed analysis uses a must/may-dirty analysis, similar to our dirtiness analysis. However, it remains unclear how their analysis works exactly; for instance, their update function does not distinguish between loads and stores. They do not use the must-dirty information to derive a set of dirtifying stores which we consider essential to precise write-back analysis.

They evaluate the impact of the live caches on the provable multi-level cache performance, but they give no results concerning their write-back analysis.

Lesage et al. [20] also consider the analysis of multi-level data caches. However, they limit themselves to the analysis of a write-through and write-no-allocate policy to avoid the complications induced by write-back caches.

Hahn and Grund [12] present relational cache analysis to overcome the necessity of exact absolute address information. Instead of using absolute addresses, they use relations between referenced addresses, such as *same block* or *different set*, to analyze a task's cache behavior. With such a relational analysis consecutive accesses to the same but unknown address can be classified as hits. Such an analysis could be used to derive sharper bounds on the number of dirtifying stores.

A detailed survey on cache analysis is given by Lv et al. [22].

The work discussed above targets the timing and cache analysis of individual tasks. Based on per-task characteristics, schedulability analyses determines whether a set of tasks can be scheduled together on a hardware platform. Some work on schedulability analysis takes platform-induced overheads into account, such as the effects of preemptive scheduling on the cache behavior [4, 19, 2].

Davis et al. [7] consider the effect of write-back caches on a preemptively scheduled fixed-priority system. Their response-time analysis uses a per-task characterization of dirty cache blocks (DCBs) and final dirty cache blocks (FDCBs). The evaluation is based on simulated execution traces of the tasks, because no WCET analysis accounting for write-back caches was available at the time of publication. Basing the analysis on simulation results, however, avoids any uncertainty that arises within static analysis. The analysis techniques presented in this paper could be used to provide the needed per-task WCET characterization by static analysis. With some further effort, the dirtiness analysis could also be used to extract DCBs and FDCBs. Davis et al. and other approaches to response-time analysis rely on timing compositionality [14]. As an example, Davis et al. assume that the cost of each additional write back is bounded by the memory latency. Due to amplifying timing anomalies, this assumption does not hold for most hardware platforms rendering naive compositional analysis unsound. However, the approach introduced in [13] to enable compositional timing analysis even in the presence of anomalies, is applicable also to the WCET analysis presented in this paper.

9 Conclusions and Perspectives

We have discussed and fleshed out the existing eviction-focussed approach to write-back analysis. We have also introduced a new store-focussed approach. As both approaches are beneficial on almost disjoint sets of tasks it is beneficial to combine both in a single analysis, as we have done.

To the best of our knowledge, we have conducted the first experimental evaluation of any WCET analysis for write-back caches. It shows that write-back caches are preferable to write-through caches from a WCET perspective for most benchmarks. The evaluation also demonstrates that write buffers are much more valuable in conjunction with write-through caches than with write-back caches. As write buffers are common in modern processors, it is essential to account for their presence for a fair comparison between the two write policies. It is future work to evaluate whether larger write buffers further shift results in favor of write-through caches.

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996. doi:10.1007/3-540-61739-6_33.

2 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 – December 2, 2011*, pages 261–271, 2011. doi:10.1109/RTSS.2011.31.

3 ARM Limited. *ARM946E-S Technical Reference Manual. Available at* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0201d/DDI0201D_arm946es_r1p1_trm.pdf.

4 José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. 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.

5 Christoph Cullmann. Cache persistence analysis: a novel approach – theory and practice. In *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 121–130, 2011. doi:10.1145/1967677.1967695.

6 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013. doi:10.1145/2435227.2435236.

7 Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 309–318, 2016. doi:10.1145/2997465.2997476.

8 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999. doi:10.1023/A:1008186323068.

9 Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011: Predictability and Performance in Embedded Systems*, volume 18, pages 59–68. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2011.

10 Freescale Semiconductor, Inc. *MPC603e RISC Microprocessor User's Manual. Available at* http://www.nxp.com/files/32bit/doc/ref_manual/MPC603EUM.pdf.

11 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.

12 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 102–111, 2012. doi:10.1109/ECRTS.2012.14.

13 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 299–308, 2016. doi:10.1145/2997465.2997471.

14 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

15 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications*

Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011, pages 203–212, 2011. doi:10.1109/RTAS.2011.27.

16 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 193–202, 2015. doi:10.1145/2834848.2834872.

17 Norman P. Jouppi. Cache write policies and performance. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 191–201. ACM, 1993. doi:10.1145/165123.165154.

18 Daniel Kröning and Silvia M. Müller. The impact of write-back on the cache performance. In *Proceedings of the IASTED International Conference on Applied Informatics, Innsbruck*, pages 213–217. ACTA Press, 2000.

19 Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

20 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OASICS*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2009. URL: http://www.dagstuhl.de/dagpub/978-3-939897-14-9, doi:10.4230/OASIcs.WCET.2009.2283.

21 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 12–21, 1999. doi:10.1109/REAL.1999.818824.

22 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *LITES*, 3(1):05:1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

23 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In Shmuel Sagiv, editor, *14th European Symposium on Programming (ESOP) 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005. doi:10.1007/978-3-540-31987-0_2.

24 Micron Technology, Inc. *Automotive DDR SDRAM MT46V32M8, MT46V16M16. Available at* https://www.micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf.

25 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, 2006. doi:10.4230/OASIcs.WCET.2006.671.

26 SCADE suite. URL: http://www.esterel-technologies.com/products/scade-suite/.

27 Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 – December 3, 2010*, pages 395–404, 2010. doi:10.1109/RTSS.2010.8.

28 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium, RTAS'97, Montreal, Canada, June 9-11, 1997*, pages 192–202, 1997. doi:10.1109/RTTAS.1997.601358.