Handling Write Backs in Multi-Level Cache Analysis for WCET Estimation

Zhenkai Zhang
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
zhenkai.zhang@vanderbilt.edu

Zhishan Guo
Department of Computer Science
Missouri University of Science and Technology
Rolla, MO, USA
guozh@mst.edu

Xenofon Koutsoukos
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
xenofon.koutsoukos@vanderbilt.edu

ABSTRACT

In this paper, we investigate how to soundly analyze multi-level caches that employ *write-back* policy at each level for worst-case execution time (WCET) estimation. To the best of our knowledge, there is only one existing approach for dealing with write backs in multi-level cache analysis. However, as shown in the paper, this existing approach is not sound. In order to soundly handle write backs, at a cache level, we need to consider whether a memory block is potentially dirty and when such a potentially dirty block may be evicted from the cache. To this end, we introduce a dirty attribute into *persistence* analysis for tracking dirty blocks, and over-approximate a write back window for each possible write back. Based on the overestimated write back occurring times, we propose an approach that can soundly deal with write backs in analysis of multi-level (unified) caches for WCET estimation. Possible write back costs are also integrated into path analysis. We evaluate the proposed approach on a set of benchmarks to demonstrate its effectiveness.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**; *Embedded software*;

KEYWORDS

WCET estimation, multi-level cache analysis, write back handling

1 INTRODUCTION

Hard real-time system design requires worst-case execution time (WCET) estimation for each task. Since it is impossible to derive the exact WCET of a task in general, an overestimation is necessary to ensure safety. On the other hand, the estimation should be as tight as possible to maximize the system resource utilization. However, because of the complex behavior of many micro-architectural features in modern embedded processors, it is very challenging to soundly and tightly estimate the WCET.

RTNS '17, October 4–6, 2017, Grenoble, France

ACM ISBN 978-1-4503-5286-4/17/10…$15.00
https://doi.org/10.1145/3139258.3139269

Caches are very commonly used in processors to bridge the increasing gap between the clock cycle time and main memory access time. Although the presence of caches improves the average performance, it poses great challenges on the tightness of WCET estimation. Over the past two decades, the analysis of the effects of single-level cache behavior on WCET estimation has been studied thoroughly.

Recently, analysis of cache hierarchies for WCET estimation has drawn much attention [3, 8, 9, 12, 22, 28, 29], since there is a rising need to employ high-performance processors in real-time systems, which are often equipped with multi-level caches (e.g., cache hierarchies are natural in multi-core processors[1]). However, compared to single-level caches, it is much more difficult to analyze the behavior of multi-level caches, since the interactions between cache levels need to be considered. For instance, if a memory access hits in the cache at some level, it will not proceed to affect the cache state at the next lower level.

Multi-level cache analysis for WCET estimation is still an ongoing research subject, and much work has mainly focused on instruction caches. Since every non-trivial task manipulates some data, how data references affect the behavior of a multi-level cache has to be analyzed, which should take into account the write policy as well as the write miss policy used at each level. There are two commonly employed write policies, which are *write-through* and *write-back*. When performing a write at some level, in the case of the *write-through* policy being used, the written information will propagate to the next lower level; yet, in the case of the *write-back* policy being used, the information is only written to the memory block at that level with the block being marked as dirty, and the dirty block will be written to the next lower level upon eviction. There are also two commonly used write miss policies, that are *write-allocate* and *no-write-allocate*. Upon a write miss at some level, with the *write-allocate* policy, a cache block is allocated to the memory block being written to; while, with the *no-write-allocate* policy, no cache block is allocated. Although either write policy could be used with either write miss policy, we usually have *write-back* caches use the *write-allocate* policy and *write-through* caches use the *no-write-allocate* policy.

While there has been some work on analysis of multi-level data or unified caches using the *write-through* policy [3, 12], to the best of our knowledge, only one approach has been proposed to analyze multi-level *write-back* caches so far [22]. However, as shown in this

[1] When using multi-core processors in hard real-time systems, we often fall back on techniques like cache partitioning to eliminate inter-core interference. Although the cores can be made to not interfere with each other on the shared cache(s), it is still a multi-level cache for each core.

paper, the approach proposed in [22] may not soundly handle write backs so that the WCET can be possibly underestimated. Due to a much smaller number of total writes, the *write-back* policy is more preferably used in cache hierarchies. Therefore, it is necessary to have an approach that can soundly handle write backs in multi-level *write-back* data/unified cache analysis for WCET estimation.

Compared to the *write-through* policy, the *write-back* policy poses more challenges to multi-level cache analysis, since sometimes it is hard to predict when a write back at a cache level will be triggered to happen, and when a write back happens the state at the next lower level will be updated to embrace the evicted dirty block (i.e., a write back happening at a cache level should be treated as an additional access to the next lower level during the analysis). In this paper, we propose an approach to soundly handle write backs in multi-level cache analysis, which centers on a novel dirty block tracking method.

The main contributions of this paper are: (1) We show that the approach proposed in [22] cannot soundly cope with write backs since it only considers whether the blocks are potentially dirty but not when a write back will possibly happen; (2) We propose an approach for soundly analyzing multi-level *write-back* caches, as we estimate a safe write back window for each potentially dirty block to capture possible write back occurring times; (3) We evaluate the proposed approach on a set of benchmarks, and we compare our write back window estimation method with the one proposed in [1] showing the effectiveness of our method.

The rest of this paper is organized as: Section 2 briefly sets the background on static (multi-level) cache analysis; Section 3 presents the system model under consideration; Section 4 formulates the problem; Section 5 shows the unsoundness of the only existing approach; Section 6 describes the proposed approach to write back handling in multi-level cache analysis; Section 7 evaluates the proposed approach on a set of benchmarks; Section 8 gives the related work; and Section 9 concludes this paper (with some future work).

2 BACKGROUND ON STATIC CACHE ANALYSIS

Cache analysis for WCET estimation is usually based on abstract interpretation for scalability. Such approaches aim to assign a *cache hit/miss classification* (CHMC) to each memory reference according to the abstract cache states (ACSs) derived by three different analyses [4, 24]. These analyses are usually performed on the control-flow graph (CFG) reconstructed from the low-level code of the program. At a given program point, *must* analysis derives a set of memory blocks that are *definitely* in the cache, and a memory reference to a block in the set can be classified as *always hit* (*AH*); *may* analysis determines a set of memory blocks that are *possibly* in the cache, and a memory reference to a block not in the set can be classified as *always miss* (*AM*); *persistence* analysis derives a set of memory blocks that stay in the cache once they are brought into the cache, and a memory reference to such a block is classified as *persistent* (*PS*) or *first miss* (*FM*); and, if a memory reference cannot be classified as *AH*, *AM*, or *PS*, it is categorized as *not classified* (*NC*).

Given a reference r to a memory block m, the effect of r on the ACS θ^{type}, where type is either must, may, or pers(istence), is captured by an *update* function $\mathcal{U}^{\text{type}} : \Theta^{\text{type}} \times M \to \Theta^{\text{type}}$, where Θ^{type} is the set of all the ACSs of the cache (i.e., $\theta^{\text{type}} \in \Theta^{\text{type}}$), and M is the set of all the memory blocks with respect to the cache block size (namely $m \in M$). In order to soundly merge information at a join point during analysis on the CFG, a *join* function $\mathcal{J}^{\text{type}} : \Theta^{\text{type}} \times \Theta^{\text{type}} \to \Theta^{\text{type}}$ is defined as well. The definitions of the *update* and *join* functions can be found in [4, 24]. Note that $\mathcal{J}^{\text{type}}$ is commutative and associative; in the case of combining more than two ACSs $\theta_1^{\text{type}}, \cdots, \theta_t^{\text{type}}$ ($t > 2$), we will directly use $\mathcal{J}^{\text{type}}(\theta_1^{\text{type}}, \cdots, \theta_t^{\text{type}})$ to represent the corresponding nested use of $\mathcal{J}^{\text{type}}$.

In the case of an A-way set associative cache using the least-recently-used (LRU) replacement policy, an ACS is composed of independent abstract set states, and each abstract set state uses ages $1, \cdots, A$ to logically order the memory blocks in it. In addition, each abstract set state in *persistence* analysis also uses a special age $\top$ to keep track of possibly evicted memory blocks mapped to the set (namely, these memory blocks are non-persistent).

For single-level caches, we do not need to concern about if they are accessed by some reference (as single-level caches are always accessed). However, in the case of multi-level caches, a cache level may not be accessed if the needed information is found at some level above it. If we treat a possible access at a level as always occurring during an analysis, the analysis may not be sound, since the set reuse distances of memory blocks can be possibly underestimated [8]. The set reuse distance between two references to the same memory block at a cache level is defined as the relative age of the block when the second reference occurs [8].

For a reference r, its *cache access classification* (CAC) in terms of a cache level is used to represent the possibility that the level will be accessed by r [8]. Let θ^{type} denote the ACS at this level immediately before r, and let m denote the memory block with respect to the cache block size at this level having the information needed by r. The CAC regulates how the effect of r on θ^{type} should be considered, as demonstrated in Tab. 1. If the CAC is *always* (*A*), the access will always occur, so r will always affect the ACS. On the other hand, if the CAC is *never* (*N*), the access will never happen, and the ACS will not be affected (so the update is equivalent to an identity function). If the CAC cannot be either *A* or *N*, it is *uncertainly* (*U*), that means the access may or may not happen. In order to ensure soundness, the update with respect to an *U* CAC needs to take into account two possible scenarios (both access occurring and access not occurring) by joining them.

Table 1: Consider the effect of reference r on the ACS of a cache level

CAC	How θ^{type} is updated
Always (*A*)	$\mathcal{U}^{\text{type}}(\theta^{\text{type}}, m)$
Never (*N*)	θ^{type}
Uncertainly (*U*)	$\mathcal{J}^{\text{type}}\left(\mathcal{U}^{\text{type}}(\theta^{\text{type}}, m), \theta^{\text{type}}\right)$

As described in [8], for a reference r that is possible to access a cache level (i.e., its CAC is not *N* at this level), if r can be safely classified as *AH* at this level, r will never need to access all the

lower levels, namely its CAC is N at any lower level; if r can be safely classified as AM at this level, r is also possible to access the next lower level, namely its CAC at the next lower level is the same as the CAC at this level (i.e., A or U). Note that if a reference always/never accesses a cache level in reality, but its CAC at that level is U in an analysis, the analysis is still sound although the result may not be precise.

3 SYSTEM MODEL

In this paper, we focus on a generalized non-inclusive[2] cache hierarchy model, which has n cache levels. At each cache level Lx (where $1 \leq x \leq n$), it is a unified cache unless otherwise specified (e.g., sometimes it is simpler and cleaner to just use multi-level data caches in examples). Note that a unified cache contains both instructions and data, so the proposed approach can be easily adapted for the analysis of cache hierarchies with levels composed of separate instruction and data caches.

At each cache level, the write policy is ***write-back***, and the write miss policy is ***write-allocate***, which is the most common combination. We assume that each cache is set associative, and the LRU replacement policy is employed. The size of a cache block may not be the same at different cache levels, but it is assumed that the block size does not increase as the level goes up. (Although most of the processors have the same block size at different levels, there are some exceptions. For example, in Alpha 21164, the L1 block size is 32B, the L2 block size is 32B/64B, and the L3 block size is 64B.)

We assume that when a write back happens, the dirty block is written to the next lower level in the memory hierarchy first, followed by the cache action that triggered the write back (i.e., no victim or write buffer is used between each pair of levels). We also assume cache levels are not searched in parallel for a piece of information, namely a cache level is searched because of a cache miss at its immediate upper level. For convenience, we say that data reads are made via load instructions and data writes are made via store instructions (i.e., RISC architecture is used), although this is not a restriction at all.

4 PROBLEM STATEMENT

When analyzing data/unified caches for WCET estimation, a recognized difficulty is due to dynamic load/store instructions, whose accessed memory addresses are not directly known but are computed at run time. Therefore, we first need to perform an address analysis to derive all the possibly accessed memory addresses for each load/store instruction. Several methods have been proposed for address analysis [21, 25], and we can just use an existing one, namely address analysis is not in the scope of this paper.

The problem that we concentrate on in this paper is how to capture and propagate the effects caused by store instructions soundly in multi-level caches, specifically in multi-level *write-back* caches, for WCET estimation. In contrast with multi-level *write-back* caches, the problem in multi-level *write-through* caches is much easier, since the information written by a store instruction will be propagated through the hierarchy at the time when the write happens; and approaches for soundly analyzing multi-level *write-through* caches have been proposed [3, 12]. The reason why the problem becomes much harder in terms of multi-level *write-back* caches is that the propagation of written information to the next lower level is always postponed; thus, at each level, we need to track the blocks that may cause write backs, estimate the time points when a write back may happen, and take into account the possible write back effects on ACSs of an analysis.

In spite of the popularity of using the *write-back* policy in cache hierarchies (especially in non-inclusive cache hierarchies), the problem of multi-level cache analysis in the presence of write backs is not well investigated.

5 UNSOUNDNESS OF EXISTING APPROACH

To the best of our knowledge, the only existing approach that takes into account write backs in multi-level cache analysis is proposed in [22]. This existing approach tracks whether a memory block is dirty at a cache level in both its *must* and *may* analyses (which are called *hit* and *miss* analyses respectively in [22]). To capture any possible write backs, the approach also introduces "phantom" memory blocks in its *must* analysis; a "phantom" memory block signifies the block may be dirty but it should not be reported as *AH* if the block is accessed. During its *must* or *may* analysis, if a tracked dirty block is evicted out of the ACS of a cache level due to a reference, the ACS of the next cache level for that analysis will be updated according to the analysis semantics and the dirty condition of that block (definitely dirty or possibly dirty)[3]. Fig. 1 shows an example of write back handling in its *must* analysis, where the memory block $\overline{m}_z$ is tracked as definitely dirty and the next reference to the memory block m_b is classified as L1 *AM* by its *may* analysis (namely, this reference to m_b will *always* result in an L2 cache access). As we can observe from the updated state, when $\overline{m}_z$ is evicted out of the ACS of L1, it is used to update the ACS of L2; and then both the ACSs of L1 and L2 are updated using m_b.

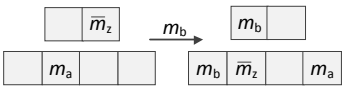

Figure 1: An example of write back handling in *must* analysis of [22]

However, the approach proposed in [22] for handling write backs may be unsound. For instance, as shown in Fig. 2, we have a 2-level data cache hierarchy (where L1 cache is 2-way set associative and L2 cache is 4-way set associative) and a CFG. The nodes in the CFG are annotated with memory blocks that are accessed by load/store instructions inside the nodes. All of the annotated memory blocks are mapped to the same L1 and L2 cache sets in the data cache hierarchy. As we can observe from the figure, m_z is written no

[2] There are three cache hierarchy types, which are inclusive, exclusive, and non-inclusive. Multi-level inclusive caches require that the contents at upper cache levels must be a subset of the contents at lower inclusive levels. Multi-level exclusive caches require that the contents at a cache level should not be duplicated at any other cache level. Multi-level non-inclusive caches allow duplicated contents existing at any cache level, but they do not strictly enforce the inclusion property.

[3] The strategy of write back handling is implicitly described in [22] using examples, which indicate their supposed way to handle write backs in multi-level cache analysis. There is also an additional "live cache" abstract domain proposed in [22] to model relationships between pairs of cache levels. However, the "live caches" do not affect how the write backs are handled, so we omit the details on "live caches" here.

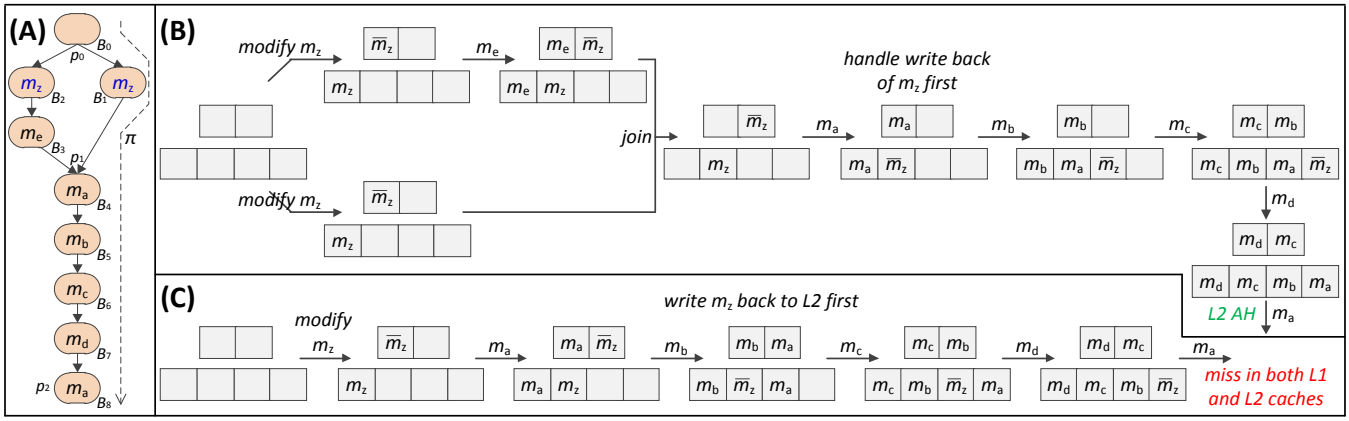

Figure 2: (A) The example CFG – m_z is written in both B_1 and B_2, and π indicates a concrete path $B_0 \rightarrow B_1 \rightarrow B_4 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7 \rightarrow B_8$; (B) Abstract cache states in *must* analysis of [22]; (C) Concrete cache states along the indicated path π

matter which branch is taken at p_0, so at the join point p_1, m_z is definitely dirty in L1 cache (which is denoted by $\overline{m}_z$ in both the abstract and concrete cache states in the figure). Due to the reference to m_a in B_4, the approach updates the ACS of L2 in its *must* analysis using $\overline{m}_z$ first before updating the ACSs of L1 and L2 using m_a. According to the derived ACSs at p_2, the reference to m_a in B_8 will be classified as L1 *AM* and L2 *AH* (to avoid cluttering the figure, we do not show the derived ACSs in its *may* analysis; yet, it is straightforward to derive them since all of the data references before p_2 have compulsory misses at both cache levels). However, if the indicated concrete path π (namely, $B_0 \rightarrow B_1 \rightarrow B_4 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7 \rightarrow B_8$) is taken, the concrete cache state of L2 at p_2 does not contain m_a, which means the reference to m_a in B_8 will have cache misses in both L1 and L2 caches. Therefore, this approach to write back handling is not sound.

The main reason why this approach cannot soundly handle write backs is that it does not take into account the uncertainty of the occurring time of a write back. Recall that *must* analysis computes upper bounds on ages of memory blocks, while *may* analysis computes lower bounds on their ages. When a tracked dirty block is forced out of the ACS at a cache level in either *must* or *may* analysis, it cannot guarantee that a write back will happen at that out-of-the-ACS moment. Instead, a write back is only incurred when a dirty block is evicted from the concrete cache state, which may be as early as when the corresponding tracked dirty block is out of the *must* ACS, or as late as when the corresponding tracked dirty block is out of the *may* ACS, or some moment in between. Failing to consider this uncertainty may result in an underestimation of the set reuse distances of memory blocks. For example, as demonstrated in Fig. 2, the approach considers once the write back of $\overline{m}_z$ to L2 cache in its *must* analysis when the reference to m_a in B_4 is processed. However, by following the indicated concrete path π, we can observe that the write back in terms of the dirty m_z actually happens when the reference to m_b in B_5 occurs, which is later than the time considered by the approach in its *must* analysis. As a consequence, the set reuse distance of m_a in L2 cache is underestimated after the reference to m_b in B_5.

6 MULTI-LEVEL WRITE-BACK CACHE ANALYSIS

In this section, we present our approach to multi-level cache analysis, which can soundly handle write backs. As a side note, we model each load/store instruction by two references – an instruction reference followed by a data reference, and we treat the two references of a load/store instruction as two different points in our analysis.

6.1 Consideration of Non-Singleton Address Set

As mentioned above, address analysis is performed to derive a set of possibly accessed addresses for each data reference. A derived address set may have more than one memory address. In order to ensure soundness, we need to take into account all the possibilities during analysis if a non-singleton address set is derived for a data reference.

Given a data reference r, if its derived address set has only a single address member, we directly use the *update* function $\mathcal{U}^{\text{type}}$ of the corresponding analysis (type $\in$ {must, may, pers}) to account for the effect of r on the input ACS θ^{type}. Otherwise, if the derived address set $\{a_1, \cdots, a_k\}$ for r is not a singleton set (i.e., $k \geq 2$), we use the following composite function [12]:

$$\mathcal{J}^{\text{type}}\Big(\mathcal{U}^{\text{type}}(\theta^{\text{type}}, m_{a_1}), \cdots, \mathcal{U}^{\text{type}}(\theta^{\text{type}}, m_{a_k})\Big),$$

where $\mathcal{J}^{\text{type}}$ is the *join* function of the corresponding analysis type, and each memory block $m_{a_i} \in \{m_{a_1}, \cdots, m_{a_k}\}$ corresponds to a memory address a_i derived for r. Basically, to handle a non-singleton address set, for each possibly accessed memory block, a copy of the input ACS is created and updated using the corresponding *update* function; and all the updated copies are joined using the corresponding *join* function. Thus, no matter which memory block is accessed in reality, its effect on the ACS is conservatively considered for soundness.

6.2 Introduction of Dirty Attribute

In order to soundly cope with write backs, we first need to capture all the memory blocks at each cache level that can be potentially

marked as dirty during a task's execution. Different from the work in [22], which separately tracks dirty blocks in *must* and *may* analyses, we propose to use *persistence* analysis as the basis, namely we only keep track of dirty blocks during *persistence* analysis at each cache level.

In order to track whether a memory block m is dirty in our *persistence* analysis, we introduce a dirty attribute $m.d$, which has one of the following three values:

- *CL*: The memory block m is *clean*.
- *DD*: The memory block m is *definitely dirty*.
- *PD*: The memory block m is *possibly dirty*.

We extend the *update* and *join* functions of *persistence* analysis to take into account this dirty attribute. (Sound *persistence* analysis could be either *may* analysis based [4] or younger set based [10], and the comparison between these two methods can be found in [27].)

Given an input ACS θ^{pers} and an accessed memory block m to the *update* function of *persistence* analysis, we set the dirty attribute $m.d$ as follows:

$$m.d = \begin{cases} DD & m \text{ is written} \\ CL & m \text{ is read } \wedge\, m \notin \theta^{\text{pers}} \\ m.d & \text{otherwise} \end{cases},$$

namely if m is modified, $m.d$ will be set as *DD*; but if m is not modified and m has never been referenced yet (as θ^{pers} keeps all the memory blocks referenced so far, if we have $m \notin \theta^{\text{pers}}$, m has not been referenced), $m.d$ will be set as *CL*; otherwise, we do not change the dirty attribute of m (no matter whether m is in an age within $1, \cdots, A$, or the special age $\top$, where the cache is A-way set associative). From now on, we will use a tag $\overline{m}$ to denote that m is modified when applying the *update* function of *persistence* analysis. Moreover, we may treat write back of m to some level as that m is loaded and then modified at that level.

Given two input ACSs θ_1^{pers} and θ_2^{pers} to the *join* function of *persistence* analysis, for each memory block m in the resultant ACS, we set the dirty attribute $m.d$ as follows:

$$m.d = \begin{cases} DD & m \in \theta_1^{\text{pers}} \text{ with } DD \;\wedge\, m \in \theta_2^{\text{pers}} \text{ with } DD \\ CL & \begin{array}{l}(m \in \theta_1^{\text{pers}} \text{ with } CL \;\wedge\, m \notin \theta_2^{\text{pers}}) \vee \\ (m \in \theta_2^{\text{pers}} \text{ with } CL \;\wedge\, m \notin \theta_1^{\text{pers}}) \vee \\ (m \in \theta_1^{\text{pers}} \text{ with } CL \;\wedge\, m \in \theta_2^{\text{pers}} \text{ with } CL)\end{array} \\ PD & \text{otherwise} \end{cases},$$

namely $m.d$ is set as *DD* in the joined ACS only when m is definitely dirty in both input ACSs, while, $m.d$ is set as *CL* if either m is a clean block in both input ACSs or m exists in only one of the input ACSs with $m.d$ equal to *CL*; otherwise, if m is a possibly dirty block in at least one of the input ACSs, or there are not two identical dirty attributes (e.g., $m.d$ is *CL* and *DD* respectively), or $m.d$ is *DD* in one ACS but m does not exist in the other ACS, $m.d$ is set as *PD*.

Note that in the case of a data reference due to a dynamic store instruction which may modify more than one memory block $m_{a_1}, \cdots, m_{a_k}$ (where $k \geq 2$), we will first have $m_{a_i}.d$ (where $1 \leq i \leq k$) set as *DD* in the updated ACS copy by $\mathcal{U}^{\text{pers}}(\theta^{\text{pers}}, \overline{m}_{a_i})$; however, $m_{a_i}.d$ will become *PD* after joining all the updated copies if $m_{a_i}.d$ is not *DD* in one or more of the other updated ACS copies (in other words, if $m_{a_i}.d$ is *DD* after considering the reference with a non-singleton address set, it has to be *DD* at least prior to the operation).

6.3 Estimation of Write Back Occurring Time

In order to clearly present our approach to multi-level *write-back* cache analysis, we investigate how to soundly bound the possible occurring time of a potential write back. For the sake of clarity, we develop the notions against a single-level cache here, which will be used later in our approach.

Recall that *persistence* analysis determines whether a memory block stays in the cache at a program point once the block was brought into the cache. If a memory block m just becomes non-persistent at a program point p (i.e. m becomes the special age $\top$ in the corresponding abstract cache set state), and $m.d$ is not *CL*, it is possible that m was modified before p and is evicted out of the cache at p, namely a write back may occur at p. Since *persistence* analysis over-approximates m's maximal age, it is sound to begin accounting for the possible effect of write back of m at p. In addition, since we may not know the exact occurring time of write back of m (if it is written back), as long as m stays in age $\top$ and also $m.d$ is not *CL* after p, we need to consider that m may be written back at one of the following points instead of p, namely, all these points form a write back window for m, and if m is written back, the write back happens at some point within the window.

As we know when a memory block is certainly accessed at a point, the block will become age 1 in *persistence* analysis, which will discontinue write back consideration of the block if it was in age $\top$ with *DD* or *PD* dirty attribute before this point. On the other hand, we also seek to conservatively "sanitize" blocks in age $\top$ having *DD* or *PD* dirty attribute, which will discontinue write back consideration as well.

In order to safely "sanitize" a memory block m whose dirty attribute is *DD* or *PD* at some point, we need to make sure that m is always clean at the point in any possible execution. Recall that *may* analysis is used to determine whether a memory block is certainly not in the cache at a given point. If *may* analysis determines that m is not in the cache at any point immediately before a reference r, m should always be clean with respect to the cache when r is going to occur (if r will modify m, m becomes dirty afterwards).

To facilitate "sanitization", during *persistence* analysis, we first apply an auxiliary function $\mathcal{S} : \Theta^{\text{pers}} \times \Theta^{\text{may}} \rightarrow \Theta^{\text{pers}}$ to change certain blocks' dirty attribute before using the needed function of *persistence* analysis on the ACS. Given a reference r, let θ^{may} denote the ACS updated due to r in *may* analysis, and let θ^{pers} denote the ACS updated due to r in *persistence* analysis. If there is only one reference immediately following r, we will use $\mathcal{S}(\theta^{\text{pers}}, \theta^{\text{may}})$ as the input ACS for the reference to the *update* function $\mathcal{U}^{\text{pers}}$, which is defined as follows:

$$\mathcal{S}(\theta^{\text{pers}}, \theta^{\text{may}}) = \theta^{\text{pers}} \text{ with } \forall i \in \{1, \cdots, d\}, \forall m \in \theta^{\text{pers}}(i)(\top) :$$

$$m.d = \begin{cases} CL & m \notin \theta^{\text{may}} \wedge m.d \text{ is not } CL \\ PD & m \in \theta^{\text{may}} \wedge m.d \text{ is } DD \\ m.d & \text{otherwise} \end{cases},$$

where d denotes the number of cache sets in the cache, and $\theta^{\text{pers}}(i)(\top)$ denotes the set of blocks whose age is $\top$ in the ith abstract set state

of θ^{pers}. As we can see from the definition, the function $\mathcal{S}$ will not modify the relative ages among blocks, but it may change the dirty attribute of some blocks which are already non-persistent (i.e., such blocks may have been written back): if m is definitely not in the cache according to θ^{may} (i.e., $m \notin \theta^{\text{may}}$)[4], $m.d$ is set as *CL* because even if m were dirty, it would have been written back before the reference; on the contrary, if $m.d$ is *DD* but m may be in the cache (given by $m \in \theta^{\text{may}}$) or may have been evicted and written back (since m has age $\top$ in θ^{pers}), we cannot guarantee m is still definitely dirty before the reference, so $m.d$ is set as *PD*; otherwise, $m.d$ is not changed. We also apply this function $\mathcal{S}$ before merging the ACSs at a join point in *persistence* analysis, namely, given two sets of ACSs $\langle\theta_1^{\text{pers}}, \theta_1^{\text{may}}\rangle$ and $\langle\theta_2^{\text{pers}}, \theta_2^{\text{may}}\rangle$, the joined ACS is computed by:

$$\mathcal{J}^{\text{pers}}\Big(\mathcal{S}(\theta_1^{\text{pers}}, \theta_1^{\text{may}}), \mathcal{S}(\theta_2^{\text{pers}}, \theta_2^{\text{may}})\Big).$$

It should be clear that applying $\mathcal{S}$ before or after merging does not affect soundness but possibly precision. For example, for a block m, if $m.d$ is *PD/DD* in θ_1^{pers} but m can be "sanitized" according to θ_1^{may} (i.e., $m \notin \theta_1^{\text{may}}$), while $m.d$ is *CL* in θ_2^{pers} and m is in θ_2^{may}, $m.d$ will be *CL* when applying $\mathcal{S}$ before merging the ACSs. In contrast, if we join the corresponding ACSs first and then apply $\mathcal{S}$ on the joined ACSs, $m.d$ will be *PD* since the joined *may* ACS will have m, which prevents m from being "sanitized" in the joined *persistence* ACS.

Note that if the write back window for a memory block m contains only one point, it does not guarantee there is definitely a write back of m occurring at that point. In order to ensure a write back of m definitely occurring at a reference r, it has to satisfy the following condition:

$$m \in \delta^{\text{must}}(r) \wedge m \in \delta^{\text{may}}(r) \wedge m.d \text{ is } DD,$$

where $\delta^{\text{must}}(r)$ (resp. $\delta^{\text{may}}(r)$) is the set of memory blocks that are forced out of the ACS during applying the *update* function of *must* (resp. *may*) analysis in terms of r. The rationale is that if this condition is satisfied, immediately before r, m must be in the cache (inferred from $m \in \delta^{\text{must}}(r)$) with the oldest age (further inferred from $m \in \delta^{\text{may}}(r)$); since the block accessed by r is definitely not in the cache (inferred from $m \in \delta^{\text{may}}(r)$), when r occurs, m must be evicted out and written back as m is definitely dirty at that point.

Estimating write back occurring time in the context of single-level caches has also been simply investigated in [1]. However, the method proposed in [1] is more pessimistic than ours, and it is also problematic if integrated into multi-level *write-back* cache analysis. Detailed comparisons and discussions are given in Section 7.

6.4 Analysis of Multi-Level Unified Caches with Write Backs

Similar to many other methods for multi-level non-inclusive cache analysis (e.g., [3, 8, 12]), our approach also analyzes a cache hierarchy in a level-by-level manner, which means we start from the first level and move downwards to the last level. At each level, *must*, *may*, and *persistence* analyses are carried out in the listed order[5] (it is the extended *persistence* analysis as described above).

[4] Note that if both *may* and *persistence* analyses are sound, a memory block will be in age $\top$ in *persistence* analysis no later than it is evicted out of the ACS in *may* analysis.
[5] If *may* analysis based *persistence* analysis is used [4], these two analyses can be performed concurrently. Otherwise, *may* analysis needs to be finished before *persistence* analysis due to the use of *may* ACS in $\mathcal{S}$ function.

Since L1 is always accessed, and the contents at L1 will not be affected by other cache level(s), we analyze L1 directly like a single-level. Otherwise, we need to consider memory access filtering behavior and write back behavior when analyzing any other level. To this end, we have a new strategy when updating ACS with respect to a reference, although we make no changes on how to join ACSs at a join point.

To facilitate presentation, let us use the following notations. Given a reference r and a level Lx ($1 \le x \le n$), $\theta_{x,r}^{\text{type,in}}$ (resp. $\theta_{x,r}^{\text{type,out}}$) represents the L$x$ ACS of the type indicated analysis into (resp. out of) the update process accounting for r; $\delta_x^{\text{must}}(r)$ (resp. $\delta_x^{\text{may}}(r)$) gives the set of memory blocks that are evicted out of the Lx ACS in the update process of *must* (resp. *may*) analysis considering r; $\{m_{a_1}, \cdots, m_{a_k}\}_{x,r}$ denotes the set of memory blocks possibly accessed by r with respect to the Lx cache block size CBS_x, where each $a_i \in \{a_1, \cdots, a_k\}$ ($k \ge 1$) is an address that may be referenced by r; $\text{CAC}_x^{a_i}$ denotes the Lx cache access classification in terms of a_i being referenced. Recall that we "sanitize" blocks in *persistence* analysis before updating the ACS, and $\theta_{x,r}^{\text{type,in}}$ will be acquired according to Tab. 2.

Algorithm 1: ACS update strategy at level Lx ($2 \le x \le n$)

```
   /* let θ_1^temp be a temporary ACS                                  */
   /* account for write back effect                                    */
1  θ_1^temp ← θ_{x,r}^{type,in};
2  if ∃m' ∈ (δ_{x-1}^must(r) ∩ δ_{x-1}^may(r)) then
3      m ← memory block corresp. to m' w.r.t. CBS_x;
4      if m'.d is DD in θ_{x-1,r}^{pers,out} then
5          θ_1^temp ← U^type(θ_{x,r}^{type,in}, m̄);
6      else if m'.d is PD in θ_{x-1,r}^{pers,out} then
7          θ_1^temp ← J^type(U^type(θ_{x,r}^{type,in}, m̄), θ_{x,r}^{type,in});
8      end
9  else
10     foreach cache set i possibly affected by r at L(x − 1) do
11         foreach m' ∈ θ_{x-1,r}^{pers,out}(i)(⊤) ∧ m'.d is not CL do
12             m ← memory block corresp. to m' w.r.t. CBS_x;
13             θ_1^temp ← J^type(U^type(θ_{x,r}^{type,in}, m̄), θ_1^temp);
14         end
15     end
16 end
   /* let θ_2^temp be another temporary ACS                            */
   /* account for cache access effect                                  */
17 θ_{x,r}^{type,out} ← ⊥;
18 foreach m_{a_i} ∈ {m_{a_1}, ··· , m_{a_k}}_{x,r} do
19     if CAC_x^{a_i} is A then
20         θ_2^temp ← U^type(θ_1^temp, m_{a_i});
21     else if CAC_x^{a_i} is N then
22         θ_2^temp ← θ_1^temp;
23     else
24         θ_2^temp ← J^type(U^type(θ_1^temp, m_{a_i}), θ_1^temp);
25     end
26     θ_{x,r}^{type,out} ← J^type(θ_2^temp, θ_{x,r}^{type,out});
27 end
```

Table 2: Input ACS to the update process considering reference r

ACS	r is preceded by r' only	r is preceded by $r'_1, \cdots, r'_t$ $(t \geq 2)$
$\theta^{\text{must,in}}_{x,r}$	$\theta^{\text{must,out}}_{x,r'}$	$\mathcal{J}^{\text{must}}(\theta^{\text{must,out}}_{x,r'_1}, \cdots, \theta^{\text{must,out}}_{x,r'_t})$
$\theta^{\text{may,in}}_{x,r}$	$\theta^{\text{may,out}}_{x,r'}$	$\mathcal{J}^{\text{may}}(\theta^{\text{may,out}}_{x,r'_1}, \cdots, \theta^{\text{may,out}}_{x,r'_t})$
$\theta^{\text{pers,in}}_{x,r}$	$\mathcal{S}(\theta^{\text{pers,out}}_{x,r'}, \theta^{\text{may,out}}_{x,r'})$	$\mathcal{J}^{\text{pers}}\left(\mathcal{S}(\theta^{\text{pers,out}}_{x,r'_1}, \theta^{\text{may,out}}_{x,r'_1}), \cdots, \mathcal{S}(\theta^{\text{pers,out}}_{x,r'_t}, \theta^{\text{may,out}}_{x,r'_t})\right)$

Given a reference r and a level Lx ($2 \leq x \leq n$), we employ the strategy described in Algorithm 1 to update the ACS. The first part (lines 1–16) of the update strategy considers if there is a definite/possible write back and the corresponding effect. We use θ^{temp}_1 as the ACS to capture any potential write back effect. If $\left(\delta^{\text{must}}_{x-1}(r) \cap \delta^{\text{may}}_{x-1}(r)\right)$ is not empty, it must be a singleton set, since a memory block m' in the intersection is *guaranteed to be evicted exactly when r happens*, which cannot be more than one. Since CBS_{x-1} may be smaller than CBS_x, line 3 obtains the memory block m corresponding to m' in terms of the Lx cache block size. The aforementioned condition for a definite write back will be met, if m' is definitely dirty at level L$(x-1)$ (line 4); line 5 captures the effect of this definite write back on ACS. On the contrary, if m' is just possibly dirty (line 6), we need to combine two scenarios at line 7 that are m is modified and no write back happens. Note that the tag $\overline{m}$ indicates m is modified and the tag is considered only in *persistence* analysis for setting dirty attribute. In other words, $\overline{m}$ just means m for both *must* and *may* analyses.

On the other hand, if we do not find such a block at line 2, we still need to consider each possible write back (lines 9–16). Since we may not know which L$(x-1)$ cache set is accessed when r occurs, we have to consider every possibility (line 10). Given a possibly accessed cache set, for each memory block m' in the set, whose write back window at L$(x-1)$ contains r (i.e., m' is non-persistent and $m'.d$ is not *CL*, as checked in line 11), line 12 finds the corresponding memory block m with respect to CBS_x, and line 13 considers the possible write back effect on ACS. Because it is uncertain whether there is a write back and (if there is) which memory block is written back, we join all the possible scenarios (including that no write back is incurred) – if $m_1, \cdots, m_y$ ($y \geq 1$) are all the memory blocks that are considered at line 13, after the first part θ^{temp}_1 will be equivalent to the result of the following expansion:

$$\mathcal{J}^{\text{type}}\left(\mathcal{U}^{\text{type}}(\theta^{\text{type,in}}_{x,r}, \overline{m}_1), \cdots, \mathcal{U}^{\text{type}}(\theta^{\text{type,in}}_{x,r}, \overline{m}_y), \theta^{\text{type,in}}_{x,r}\right).$$

If there is no such block found at line 11 (i.e., there is no possible write back), line 13 will not be reached and θ^{temp}_1 will still be $\theta^{\text{type,in}}_{x,r}$ as expected.

The second part (lines 17–27) considers the effect of information access needed by r on the resultant ACS from the first part. It follows the well-established approach relying on cache access classifications [3, 8, 12]. If one of multiple memory blocks may be accessed by r, the effect of each one of them on the ACS needs to be taken into account and combined, as previously described. It is worth noting that, at a cache level lower than L1, a memory block can become dirty only due to some write back issued from the immediate upper level. Thus, the second part will not tag any memory block as modified no matter whether or not r is a data reference due to a store instruction. (We only consider whether a data reference is due to a store instruction to tag its accessed memory block when analyzing L1 cache.)

Although the approach is targeted at analysis of multi-level *write-back* caches, at each level of which it is a unified cache, it is straightforward to make the approach applicable to dealing with separate instruction and data caches at some level(s). For example, if separate caches are used at L1 and a unified cache is used at L2, all the possible write backs issued from L1 can only be from L1 data cache when data references occur. Thus, given a reference, we identify which cache can be affected by the reference at the immediate upper level and use its ACS in the first part of the approach.

6.5 Integration of Write Back Costs into Path Analysis

As a *de facto* method, Implicit Path Enumeration Technique (IPET) is used to calculate the WCET bound [15]. It uses a set of integer linear constraints that combines the flow information and the timing effects of multi-level caches [9, 12].

In the light of our system model (write buffers are not used), write back costs need to be considered explicitly in IPET for path analysis. To derive the WCET, we maximize the following objective function:

$$\sum_{i=1}^{v} c_i \cdot x_i + \sum_{j=1}^{n} d_j \cdot y_j \ ,$$

where x_i is the number of times the basic block B_i is executed (v basic blocks in total), c_i is the worst-case cache-aware cost without considering the portion due to write backs of the basic block B_i, y_j is an upper bound on the number of write backs issued from the jth cache level (n cache levels in total), and d_j is the cost of a write back occurring at the jth cache level. We have x_i's subject to both structural and functional constraints as described in [15], and we calculate each c_i according to the derived CHMC of each reference in the basic block as stated in [9, 12].

Because the number of write backs issued from some level cannot be more than the number of modifications made to this level, we will impose the following constraints on y_j's:

$$0 \leq y_1 \leq \sum_{i=1}^{v} s_i \cdot x_i \ ; \quad 0 \leq y_2 \leq y_1 \ ; \quad \cdots \quad 0 \leq y_n \leq y_{n-1} \ ,$$

where s_i is the number of store instructions in the basic block B_i. In addition, since we can over-approximate the number of possible

write backs in B_i issued from the jth level, which is denoted by w_i^j, we will also impose the following constraints:

$$y_1 \leq \sum_{i=1}^{v} w_i^1 \cdot x_i \; ; \quad \cdots \quad y_n \leq \sum_{i=1}^{v} w_i^n \cdot x_i \; .$$

In B_i, although a memory block at the jth level may be written back to the next level at one of several points, as long as these points are consecutive without any possible modification of the block, it is treated as the same possible write back at all these points; and we have w_i^j equal to the number of distinct points, each of which can have a different possible write back issued from the jth level.

7 EVALUATION

In this section, we evaluate the proposed approach to multi-level *write-back* cache analysis for WCET estimation. We have developed a research prototype tool with the approach. It takes MIPS R3000 compliant binaries and reconstructs CFGs from them. It also computes context-sensitive call graphs to improve analysis precision. The CPLEX solver is employed to solve the generated ILP (Integer Linear Programming) problems.

Due to the limitations of our current tool, we only consider the timing effects of multi-level caches and we do not account for the effects caused by other micro-architectural features like pipelines and branch predictors. Accordingly, we assume there are no timing anomalies [16], and a reference that is classified as *NC* can be treated as a *AM* when used for WCET estimation.

The evaluation is performed on a set of benchmarks maintained by the Mälardalen WCET research group [6]. The used benchmarks are shown in Tab. 5, and they are compiled for MIPS R3000 using gcc-3.4.4. The size of each benchmark covers both its code and data. Some of the benchmarks operate on big arrays (e.g., *matmult* and *crc*), and some of them do not have load/store instructions accessing more than one address (e.g., *expint* and *prime*).

7.1 Comparison of Methods for Estimating Write Back Occurring Time

As discussed in Section 6, it is crucial to have a sound method for bounding the possible occurring time of a potential write back. In terms of single-level *write-back* caches, there is a method proposed in [1], which relies on *may* analysis to delimit possible write back ranges, and uses *must* analysis to help identify definite write backs. Analytically, our method for write back occurring time estimation dominates, especially when cache capacity is relatively large – our method will report no write backs if potentially dirty blocks stay as persistent, but the method in [1] will continuously report possible write backs as long as potentially dirty blocks are still in *may* ACS.

Since the original method in [1] is only suitable for single-level *write-back* cache analysis (it requires a small modification for multi-level *write-back* cache analysis, as described later), we compare our method for bounding write back windows with theirs in terms of single-level caches. In the experiments, we fix the cache block size as 32B and associativity as 4-way. For each benchmark, we perform two comparisons by changing the capacity of the unified cache – one is relatively large and the other one is relatively small, compared to the benchmark size. Due to the space limitation, we will not list the capacity configurations here, but they are the same as the L2 capacities shown in Tab. 4 for each benchmark (e.g., for the benchmark *bs*, the large one is 2KB and the small one is 256B). We use the number of program points where write backs are estimated to occur as the metrics to evaluate the precision.

Table 3: Number of program points where write backs are estimated to occur

Benchmark	#Pts	Large Config.		Small Config.	
		$\#WP_{Our}$	$\#WP_{[1]'s}$	$\#WP_{Our}$	$\#WP_{[1]'s}$
bs	111	0	33	12	100
insertsort	145	0	56	86	137
prime	338	0	149	55	303
expint	318	0	115	31	264
bsort100	198	0	137	170	180
cnt	325	0	178	244	291
qurt	1243	0	435	375	1053
select	537	0	180	225	421
crc	676	0	531	115	634
ns	189	0	38	0	56
matmult	418	0	102	192	375
statemate	3410	0	809	51	1281

The results are shown in Tab. 3, where the second column (#Pts) also gives the total points in each benchmark (recall that a load/store instruction is modeled with two points instead of one). From the results, we can see our method always dominates, i.e., it can more precisely identify where write backs may occur. The results match the aforementioned analytical expectation. The precision of this estimation may have a considerable impact on the overall analysis.

7.2 Effects of Occurring Time Estimation on Multi-Level Write Back Cache Analysis

The original method proposed in [1] is actually problematic, since it only distinguishes two states: "dirty" and "clean". For example, if a block m is "dirty" along one path and "clean" along another path, m is set as "dirty" after the paths are joined. It is possible that a definite write back of m will be given by the analysis under certain scenarios later, which is not correct if the path with "clean" m is concretely taken. Treating a possible write back as a definite write back may result in unsound multi-level cache analysis. A straightforward fix is to introduce a third state "possibly dirty", as what we have proposed in Section 6.

We integrate the fixed method of [1] into our multi-level *write-back* cache analysis, and perform a set of experiments to show the effects of the precision of write back occurring time estimation on multi-level *write-back* cache analysis. All the experiments are carried out on a two-level cache hierarchy, which uses *write-back* and *write-allocate* policies at each level. Certain parameters of the hierarchy are fixed, which are shown in Tab. 4. We assume that the write back stall at a level is the same as the access latency of its next lower level, and we also assume that any needed information can be found in the main memory with a 100-cycle latency.

We carry out the experiments on each benchmark by changing L1 and L2 cache capacities. For a benchmark, we use two capacity configurations. In the first configuration, L1 cache size is greater

Table 4: Fixed parameters of two-level cache hierarchy

Level	Block Size	Associativity	Latency	Write Back Stall
L1	16B	2-way	1-cycle	10-cycle
L2	32B	4-way	10-cycle	100-cycle

than the size of the benchmark but not larger than twice the benchmark size. Additionally, L2 cache size is four times greater than L1 cache size. In the second configuration, L2 cache size is smaller than the size of the benchmark but not less than half the benchmark size. Moreover, L1 cache size is half the size of L2 cache. The two configurations for each benchmark are shown in Tab. 5.

Table 5: Cache capacity configurations for each benchmark

Benchmark	Size	Configuration 1		Configuration 2	
		L1	L2	L1	L2
bs	480B	512B	2KB	128B	256B
insertsort	500B	512B	2KB	128B	256B
prime	628B	1KB	4KB	256B	512B
expint	976B	1KB	4KB	256B	512B
bsort100	1008B	1KB	4KB	256B	512B
cnt	1444B	2KB	8KB	512B	1KB
qurt	1580B	2KB	8KB	512B	1KB
select	1716B	2KB	8KB	512B	1KB
crc	2183B	4KB	16KB	1KB	2KB
ns	5624B	8KB	32KB	2KB	4KB
matmult	5804B	8KB	32KB	2KB	4KB
statemate	10591B	16KB	64KB	4KB	8KB

Tab. 6 shows the experimental results, where the estimates are in clock cycles. For each benchmark, we also calculate how much the relative pessimism is by $\frac{\text{WCET}_{\text{[1]'s}}}{\text{WCET}_{\text{our}}} - 1$. From the results, we can observe that the dominance of our approach is overwhelming when the cache capacities are relatively large (the relative pessimism ranges from 59.4% to even 968.5%). The reason for this phenomenon is not hard to explain – when the precision of write back occurring time estimation is low, too many false-positive write backs will significantly inflate the WCET estimates. The comparison under the small configuration is not as striking as that under the large one, due to the write back bounds imposed by our constraints (Section 6.5); but our approach still dominates in most cases. The experiments are carried out on a Linux machine with a 3.4GHz quad-core processor and 16GB memory. All of the analysis times range from hundreds of milliseconds to tens of seconds.

8 RELATED WORK

Static cache analysis for WCET estimation has been studied extensively over the past two decades, which is mainly based on either abstract interpretation or static cache simulation [19, 24]. Much work focuses on single-level caches, especially that involving data cache analysis. In [25], static cache simulation is extended to carry out analysis of single-level data caches. In [20], an approach based on cache miss equations is proposed to derive exact data cache hit/miss patterns even in the presence of non-rectangular loops. In [21], *must* analysis is used for data cache analysis, which may need to partially unroll loops for more analysis precision. As argued in [10], *persistence* analysis is more suitable for data cache analysis; however, the original *persistence* analysis proposed in [5] is unsound, and the sound ones are introduced in [4, 10]. (A comparison of these two sound *persistence* analyses is investigated in [27].) Based on *persistence* analysis, in [10], a scope-aware data cache analysis method is proposed, which captures the temporal usage of memory blocks possibly accessed by a data reference over different loop iterations. In [23], input dependent and independent data cache behavior is studied, where *persistence* analysis and *pigeon-hole* principle are combined to deal with input dependent (unpredictable) data references.

The first multi-level (non-inclusive) cache analysis is proposed in [18], which is an extension to static cache simulation. Later, in [8], it has been pointed out that this method is not suitable for analyzing multi-level set associative caches, and it is proposed to use *cache access classification* (CAC) to filter the references at each cache level and to define an update strategy to take into account the uncertain accesses. Methods for analyzing multi-level instruction caches of types other than non-inlucisve are also presented in [9, 28, 29]. Based on the work in [8], an approach for analyzing multi-level non-inclusive data caches with *write-through* and *no-write-allocate* policies is proposed in [12], and a method for analyzing non-inclusive unified cache hierarchies with *write-through* and *write-allocate* policies is proposed in [3]. In [22], an abstract domain called live caches is proposed to model relationships between cache levels, and this domain is used to improve the precision of multi-level unified cache analysis with the *write-back* policy. As discussed in this paper, the approach to write back handling in [22] is not sound.

Cache hierarchies are natural in multi-core processors, and much work focuses on analysis of inter-core interferences on shared instruction caches [7, 14, 17, 26]. Some work tries to take into account data references as well. In [13], it analyzes conflicts on shared data caches and proposes bypass heuristics to reduce these conflicts. In [2], an analysis framework that covers different micro-architectural components including data caches in a multi-core processor is proposed. It is assumed that the *write-through* policy is employed in [2, 13].

In order to avoid using too pessimistic estimation, probabilistic timing analysis (PTA) techniques have been proposed to produce multiple estimations with the probabilities that they can be exceeded. In [11], a measurement-based PTA approach is proposed to estimate probabilistic WCET in the presence of multi-level unified caches. In this paper, we want to guarantee safety, and consider PTA as a complementary methodology to our approach.

9 CONCLUSION AND FUTURE WORK

How to soundly analyze multi-level caches in the presence of write backs is challenging, and there are pitfalls – the only existing approach to the problem is shown as unsound. As the first step towards sound multi-level *write-back* cache analysis, we propose an approach based on a novel dirty block tracking method that estimates possible write back occurring times. We evaluate the proposed approach on a set of benchmarks with a two-level cache

Table 6: WCET estimates and relative pessimism comparison

Benchmark	Large Config.			Small Config.		
	$WCET_{our}$	$WCET_{[1]'s}$	$\frac{WCET_{[1]'s}}{WCET_{our}} - 1$	$WCET_{our}$	$WCET_{[1]'s}$	$\frac{WCET_{[1]'s}}{WCET_{our}} - 1$
bs	3112	4960	59.4%	10134	10134	0%
insertsort	16043	44343	176.4%	176913	183213	3.6%
prime	41433	359835	768.5%	1146835	1148265	0.1%
expint	16758	117423	600.7%	406163	426163	4.9%
bsort100	1118657	3693657	230.2%	19016917	20026717	5.3%
cnt	20440	104560	411.5%	440562	440662	<0.1%
qurt	27698	68630	147.8%	135374	139474	3.0%
select	30202	63852	111.4%	198732	198732	0%
crc	86328	922443	968.5%	571148	1136163	98.9%
ns	35873	139693	289.4%	54763	151803	177.2%
matmult	712632	2972252	317.1%	23066142	23066142	0%
statemate	57719	102799	78.1%	98490	137980	40.1%

hierarchy, and compare our write back window estimation method with the one proposed in [1]. The experimental results show that the precision of write back occurring time estimation matters significantly when the cache capacities are relatively large.

In the future, we plan to study how to reduce pessimism in our analysis. One method is to use loop unrolling as described in [21], but it is very expensive. Scope-aware cache analysis is promising and can greatly improve the precision of *persistence* analysis [10]. However, we also need to improve the precision of *may* and *must* analyses to reduce the possible write back window and obtain more definite write backs. In addition, we will refine the integration with path analysis to have more fine-grained representation of write backs in IPET. As another direction, we will also investigate multi-level cache-related preemption delay in the presence of write backs, which complements our previous work in [30].

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (CNS-1739328), as well as a startup grant and a seed grant from Intelligent System Center at Missouri University of Science and Technology.

REFERENCES

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache Behavior Prediction by Abstract Interpretation. In *SAS '96*. 52–66.
[2] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. 2012. A Unified WCET Analysis Framework for Multi-core Platforms. In *RTAS '12*. 99–108.
[3] Sudipta Chattopadhyay and Abhik Roychoudhury. 2009. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *RTSS '09*. 47–56.
[4] Christoph Cullmann. 2013. Cache Persistence Analysis: Theory and Practice. *ACM Transactions on Embedded Computing Systems* 12, 1s, Article 40 (March 2013), 25 pages.
[5] Christian Ferdinand and Reinhard Wilhelm. 1998. On Predicting Data Cache Behavior for Real-Time Systems. In *LCTES '98*. 16–30.
[6] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *WCET '10*. 136–146.
[7] Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *RTSS '09*. 68–77.
[8] Damien Hardy and Isabelle Puaut. 2008. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *RTSS '08*. 456–466.
[9] Damien Hardy and Isabelle Puaut. 2011. WCET Analysis of Instruction Cache Hierarchies. *Journal of Systems Architecture* 57, 7 (Aug. 2011), 677–694.
[10] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-Aware Data Cache Analysis for WCET Estimation. In *RTAS '11*. 203–212.
[11] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. 2013. Multi-level Unified Caches for Probabilistically Time Analysable Real-Time Systems. In *RTSS '13*. 360–371.
[12] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2009. WCET Analysis of Multi-Level Set-Associative Data Caches. In *WCET '09*. 1–12.
[13] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2010. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *RTNS '10*. 2283.
[14] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2009. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *RTSS '09*. 57–67.
[15] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *DAC '95*. 456–461.
[16] Thomas Lundqvist and Per Stenström. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *RTSS '99*. 12–.
[17] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. 2010. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *RTSS '10*. 339–349.
[18] Frank Mueller. 1997. Timing Predictions for Multi-Level Caches. In *LCTRTS '97*. 29–36.
[19] Frank Mueller. 2000. Timing Analysis for Instruction Caches. *Real-Time Systems* 18, 2/3 (May 2000), 217–247.
[20] Harini Ramaprasad and Frank Mueller. 2005. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *RTAS '05*. 148–157.
[21] Rathijit Sen and Y. N. Srikant. 2007. WCET Estimation for Executables in the Presence of Data Caches. In *EMSOFT '07*. 203–212.
[22] Tyler Sondag and Hridesh Rajan. 2010. A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis. In *RTSS '10*. 395–404.
[23] Jan Staschulat and Rolf Ernst. 2006. Worst Case Timing Analysis of Input Dependent Data Cache Behavior. In *ECRTS '06*. 227–236.
[24] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems* 18, 2/3 (May 2000), 157–179.
[25] Randall T. White, Frank Mueller, Chris Healy, David Whalley, and Marion Harmon. 1999. Timing Analysis for Data and Wrap-Around Fill Caches. *Real-Time Systems* 17, 2-3 (Dec. 1999), 209–233.
[26] Jun Yan and Wei Zhang. 2008. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *RTAS '08*. 80–89.
[27] Zhenkai Zhang and Xenofon Koutsoukos. 2015. Improving the Precision of Abstract Interpretation Based Cache Persistence Analysis. In *LCTES'15*. Article 10, 10 pages.
[28] Zhenkai Zhang and Xenofon Koutsoukos. 2015. Precise Multi-level Inclusive Cache Analysis for WCET Estimation. In *RTSS '15*. 350–360.
[29] Zhenkai Zhang and Xenofon Koutsoukos. 2015. Top-Down and Bottom-Up Multi-Level Cache Analysis for WCET Estimation. In *RTAS '15*. 24–35.
[30] Zhenkai Zhang and Xenofon Koutsoukos. 2016. Cache-related Preemption Delay Analysis for Multi-level Inclusive Caches. In *EMSOFT '16*. Article 16, 10 pages.