A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis

Tyler Sondag and Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA 50011
{sondag,hridesh}@iastate.edu

***Abstract*— As demand for computational power of embedded applications has increased, their architectures have become more complex. One result of this increased complexity are real-time embedded systems with set-associative multi-level caches. Multi-level caches complicate the process of program analysis techniques such as worst case execution time (WCET). To address this need we have developed a sound cache behavior analysis that handles multi-level instruction and data caches. Our technique relies on a new abstraction, *live caches*, which models relationships between cache levels to improve accuracy. Our analysis improves upon previous multi-level cache analysis in three ways. First, it handles write-back, a common feature of cache models, soundly. Second, it handles both instruction and data cache hierarchies, and third, it improves precision of cache analysis. For standard WCET benchmarks and a multi-level cache configuration analyzed by previous work, we observed that live caches improve WCET precision resulting in an average of 6.3% reduction in computed WCET.**

I. Introduction

Computing an upper bound on execution time, known as worst-case execution time (WCET), for programs running on real-time systems with hard deadlines is crucial [1]–[3]. It is essential that the computed upper bound must be sound (longer than any possible execution) [1]. However, an upper bound that is too high will result in wasted resources [1]. This is a difficult problem that requires determining upper bounds on program paths, memory access times, etc [1].

One aspect of this problem is determining an upper bound on each memory access. To drastically reduce memory access time, many processors incorporate caches, some with multiple levels. Other cache optimizations include splitting instruction and data caches. Due to the complexity of caches (multiple levels, shared vs. unified levels, associativity, replacement policy, etc), analyzing the behavior of these caches is difficult. However, if we can analyze the cache behavior precisely, we can dramatically reduce the computed worst case access time for memory accesses [2].

A large body of previous work exists to analyze cache behavior statically [1]. However, these techniques either focus on instruction caches [2], [4]–[8], or only handle single-level data caches [9]–[12]. Thus, they do not support systems that incorporate both instruction and data caches. Furthermore, since they only treat single-level data caches, they do not handle *write-back*. Write-back is a common technique used in data caches to reduce memory accesses when contents in a cache are modified. With write-back, when a modified content is evicted from cache, it is written to the next cache level or memory. So, the next access to this content may be a cache hit, whereas previous work would sometimes classify it as a potential cache miss. This affects the upper bound on memory access time. We also show that if existing analyses are applied to unified (instruction+data) cache hierarchies with write-back the result is unsound.

Our contribution is an analysis for multi-level instruction and data cache hierarchies. There are two main novelties of our technique. First, we consider the cache hierarchy as a whole (as opposed to analyzing cache levels in the hierarchy separately [4]–[12]), which improves precision. Second and more importantly we add a new abstraction, *live caches*, to the classical abstract domain for caches [2] to handle multi-level caches. Live caches describes relations between pairs of cache levels (in order to reduce the imprecision of the join analysis [2], [4]) as well as information inside a cache level (in order to handle write-back).

Improving precision is important because a misclassified memory access may drastically increase computed upper bounds of the WCET analysis, e.g. a cache miss may be as much as 30x more expensive compared to a cache hit [13]. Soundness of cache analysis is important because it directly affects soundness of WCET. An unsound application of a cache analysis may lead to incorrect WCET.

Similar to previous work [2], [9], we have used the WCET benchmarks maintained by the Mälardalen WCET research group [14] to evaluate our technique. For 20 out of 32 (62.5%) of these benchmarks our analysis technique showed precision improvements over previous analysis. On an average we saw a 6.3% reduction in computed WCET as a result of our multi-level cache analysis technique. This result is significant because it directly translates to a corresponding reduction in wasted resources for real-time systems [1].

In the rest of this paper, we discuss the technical underpinnings of this work that include:

- the definition of live caches and illustration of the key aspects of their behavior,
- an abstract interpretation for multi-level instruction and data cache analysis, and
- an empirical evaluation of our cache analysis.

```
1 A input(*sum)
2 A if(!*sum)//init arr
3 A  for(i=0; i<10; i++)
4 A   arr[i] = 0;
5 B else //compute sum
6 B  for(i=0; i<10; i++)
7 B   *sum += arr[i];
8 B output(*sum);
```

Variable	Cache block
sum	x
arr[0-7]	y
arr[8-9]	z
i	register

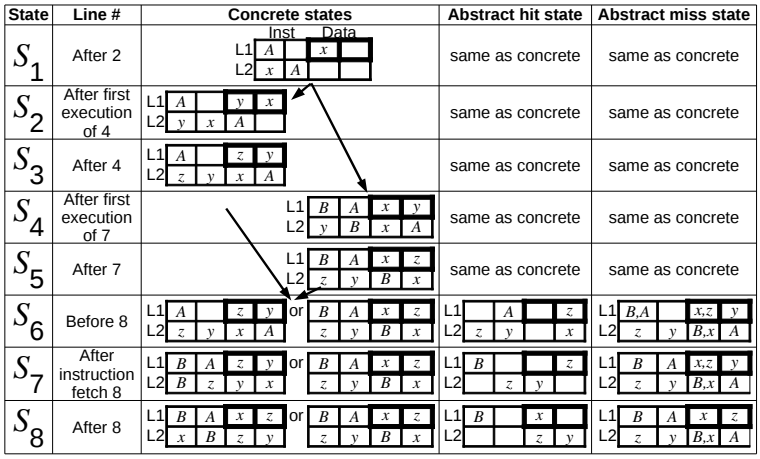

Figure 1. *Left:* code example. Letters after line numbers indicate instruction cache block containing the code for the line. Table shows variable to cache block mapping. *Right:* cache contents for each state using previous analysis [2]. Bold cells denote data cache. Loop unrolling is assumed.

II. Problems with Existing Cache Analyses

We first describe a typical cache analysis using abstract interpretation [2]. Basic cache terms are as follows.

- Lx "cache level x". Lower levels are closer to the processor and faster, but smaller.
- *capacity*$_x$ denotes the size in bytes of the Lx cache.
- *line size*$_x$ (block size) is the number of bytes loaded into the Lx cache upon a miss.
- *Block* refers to a segment of memory. As viewed from the Lx cache, there are $\mathcal{M}/$*line size*$_x$ blocks of memory, where $\mathcal{M}$ is the capacity of memory.
- *Associativity* is the number of cache lines a memory block may reside. In a 2-way associative cache, a block may be in one of two cache lines. A 1-way associative cache is direct mapped and *capacity*$_x$/*line size*$_x$-way is fully associative.
- *Hit/Miss* When a memory location is in a cache level, it is a hit; otherwise, a miss.
- *Replacement strategy*: A technique to determine which block to replace when a cache is full and an additional block is needed. Like [2], [4], we use the least recently used (LRU) policy where the block accessed least recently is removed, however, other policies could also be used.
- *Mainly-inclusive*: The relationship between cache levels. In this policy, cache levels operate independently. Updates for higher cache levels are determined based on misses and write-backs from the previous level. If Lx and Lx+1 are mainly-inclusive, then most blocks in Lx are also in Lx+1.

A. Abstract Interpretation based Cache Analysis

Figure 1 illustrates an abstract interpretation of cache behavior for a two level cache hierarchy. The first level, L1, is a split (instruction separate from data) 2-way associative cache. The second level, L2, is a unified (containing both instruction and data blocks) 4-way associative cache. Both levels use LRU as a replacement policy with write-back and have a line size of 32 bytes. For simplicity, we consider one cache set from each cache. Since the concern is cache hits and misses, the cache state aims to capture when blocks are loaded into and evicted from each cache level.

The left side of the figure shows sample code. Letters after line numbers indicate the instruction cache block the code belongs to. The right side illustrates states for the cache analysis (assuming the initial state is empty). The column marked with concrete states shows the cache states according to the concrete semantics, whereas those marked with abstract show the states of the cache from the point of view of abstract hit/miss analysis. Thus imprecision in analysis can be analyzed by comparing them.

Similar to previous work [2], [4], for each point in a program, assume there is a list of reads and writes to analyze. Based on the reads and writes for a program point, the state of the cache is updated. Since existing cache analysis techniques [2], [4] do not handle write-back, in this example there is no difference between reads and writes.

After analyzing lines 1–2, state S_1 results. L1 shows instruction cache block A as the most recently used instruction cache block and x as the most recently used data cache block. L2 shows that x was the most recently accessed block. Notice that since L1 is a split cache instruction and data blocks are treated separately, whereas since L2 is a unified cache both type of blocks are treated uniformly.

Depending on how the **`if`** condition on line 2 evaluates control might be transferred to either line 3 or line 6. Both paths must be analyzed because actual path is unknown.

Let us first analyze the loop on lines 3–4. Note that variable `i` is assigned to a register and does not impact the data cache behavior. For simplicity we assume that the loop has been completely unrolled. S_2 shows the state after analyzing the first loop iteration. Assume that the first 8 locations of array `arr` are in block y and **`sizeof(int)`** is 4. S_2 shows that block y is the most recently accessed block in L1 data cache and most recently accessed block in L2. Tracking actual values for variables is not necessary as it doesn't influence the hit/miss behavior [4]. S_3 shows that state after all executions of the loop, where L1 data cache contains the entire array `arr` (both blocks y and z).

Next, we analyze the loop on lines 6–7. Note that the state S_4 corresponding to this code is derived from S_1

not S_3. On line 7, the value in `arr[i]` is fetched first followed by adding the result to `sum`. Thus, after the first iteration S_4 shows us that x is the most recently used block in L1 data cache, but block y has been loaded into both levels. Further, before these data locations were referenced, instruction block B was loaded into L1 instruction cache and L2. State S_5 results after analyzing all iterations of the loop. S_5 is similar to S_4 but block z has also been loaded into both cache levels.

Join function*:* When line 8 is analyzed, there are two predecessor lines 4 and 7 with state S_3 and S_5 respectively. Thus, the analysis must merge or "join" the states denoted as $S_6 = S_3 \bigwedge S_5$. Figure 1 shows the merged states for a "hit" and a "miss" analysis.The hit analysis tells us for each reference "hit" or "unknown" and the miss analysis tells us "miss" or "unknown". For example, if a block is in a state of the hit analysis, it must be a hit, otherwise it is unknown.

For the "hit" state, the worst case of block locations is used. For example, the L1 instruction cache block A was in the last space in S_5 and the first space in S_3. Thus, A in S_6 takes the last position in the L1 instruction cache.

For the "miss" state, the best case is chosen. For example, block B is in the first position of L1 in S_5 but is not in L1 cache in S_3. Thus in S_6, B takes the first position in the L1 instruction cache.

After merging potential states, line 8 is analyzed. First, instruction block B is fetched giving state S_7. Since B is not in hit analysis state S_6, B is added to L1 updating the LRU order and evicting A. We can see by looking at the possible concrete states that B may actually be an L1 hit meaning it is not sound to update L2 with B. The analysis does not have the luxury of considering all possible concrete states. In general, thus the analysis uses the miss analysis and only updates L2 with the block if the miss analysis can guarantee the reference will miss L1. Otherwise, an update to L2 with an empty block is done. Therefore, in this example, B is not added to L2 since the miss analysis can not guarantee that B will miss L1. However, the LRU order of L2 must be updated in case B does miss so an empty block is used.

Next, block x is accessed resulting in an update to L1 data cache as shown in S_8.

B. Problems that Affect Soundness and Precision

With the background on abstract interpretation based cache analysis in place, we now illustrate the three problems with existing techniques via our example in Figure 1.

Cache hierarchy*:* Consider the position of instruction block A in L1 of state S_1 of Figure 1. Even though A is the most recently used block in the L1 instruction cache, x is the most recently used block in L2. This position of A in L2 shows how data cache behavior affects the behavior of instruction blocks in a unified L2 cache. Since most L2 caches are unified, simultaneously analyzing both instruction and data behavior is necessary.

Write-back*:* Since analyzing both instruction and data cache is required, handling write-back properly is essential. To illustrate the need to handle write-back, let us revisit the transition from S_7 to S_8 of the hit analysis in Figure 1. In the "hit" state for S_7, block z in L1 may be dirty. In the transition to state S_8 this block is evicted. As a result of the potential eviction of a possibly dirty block, L2 should be updated. This additional update would force block y out of L2. *Without considering write-back, the analysis would report that y must hit L2 which is not sound.* For unified cache levels write-back also affect the soundness of instruction caches since data cache write-backs to L2 impact the instruction cache blocks in L2. Thus, it is crucial to handle write-back properly in a multi-level cache analysis.

Precision*:* Notice the existence of block x in both potential concrete states of S_7 in Figure 1. This means that x must be somewhere in the cache, however, the hit analysis can not guarantee this. Thus, even though the reference to x on line 8 will be a "hit", the analysis says "unknown" or worst case memory access. This loss of information is due to the sound approximations of the join function. Since joins are frequent (**`if`**s, loops, etc), the precision of the join function is key to the overall precision of the analysis.

III. Multi-level cache analysis with Live Caches

The soundness and precision problems with existing analyses arise because they do not take into account the global view of the hierarchy. As a result, they miss the cache blocks that must exist in the hierarchy, even though their existence in any cache level may not be soundly computed. They also do not soundly capture write-back. To solve these problems we introduce a new abstraction to the classic abstract domain of caches, that we call *live caches*.

A. Live Caches

A live cache is an abstract cache maintained for the purposes of multi-level cache analysis. For every pair of concrete cache levels the analysis maintains one live cache. These live caches contain blocks that must exist (blocks that are live references, thus the term live caches) in at least one of two different cache levels. The example in Figure 1 contains two cache levels L1 and L2. Thus the multi-level cache analysis maintains one live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$.

Figure 2 shows the "hit" analysis state for S_1 from Figure 1 as determined by our analysis. This state now includes the live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$. The state for L1, L2, and $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ is shown on top-left, bottom-left, and bottom-right respectively in this section. The live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ consists of two sets, one corresponding to L1 instruction cache and L2 and the other corresponding to L1 data cache and L2.

The associativity of each set in the live cache is equal to the larger of the two caches, e.g. the associativity of $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ is 4 since that is the larger between L1 and L2.

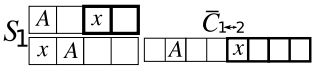

Figure 2. Hit analysis state for S_1

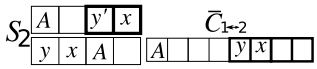

Figure 3. Hit analysis state for S_2

The position of a block in live cache is determined by taking the better case of the same block's position in the two cache levels that the live cache relates. This is because a live cache is updated whenever either of its corresponding cache levels, more specifically sets, is updated.

As mentioned previously, write-back introduces new behavior that must be modeled. Consider an example for the "hit" analysis for S_2 in Figure 3. The y' notation in L1 of S_2 means that the block y is dirty (modified).

To handle the addition of live caches and write-back, the join functions for both the "hit" and "miss" analysis must be modified. For the "hit" analysis, when a block exists in both states, the resulting block is dirty only if both blocks are dirty. However, in the case that only one block is dirty, we must keep track that a potentially dirty block may exist in that cache location in order to safely handle write-back. For live caches, we join the two corresponding cache levels giving blocks the worst-case proximity to eviction between the two. For the "miss" analysis, when a block exists in both states, if either block is dirty, the resulting block is dirty.

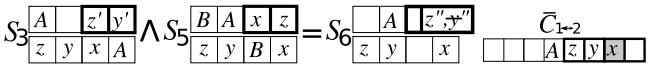

Figure 4. Hit analysis state for S_6 from Figure 1

Consider the join of states S_3 and S_5 to create S_6 for the "hit" analysis in Figure 4. Locations of blocks in live cache are determined similar to the previous examples except for block x. The location for x is chosen as a worst case location between x in L1 in S_5 and L2 in S_3. Also, note that in S_5, z is clean in L1 but in S_3 it is dirty. Thus, in S_6, z'' denotes that z may be dirty in L1. Also, notice that in S_3, y exists in L1 and is dirty, however, y does not exist in L1 in S_5. Thus, since we can not guarantee y will be in L1 cache, y should not exist in L1 in S_6. However, since y may exist and may be dirty, when it may be evicted from L1, the LRU order of L2 should be updated. Thus, y should not be reported as a hit if accessed (denoted by ~~y~~) but is kept in the state so higher levels can be modeled in a sound way. Even though y is modeled in the "miss" state, we can not safely use eviction from L1 in the "miss" state to update L2 because of the difference in updates for the "miss" analysis.

Consider the same join for the "miss" analysis. Even though z is clean in L1 of S_5, since it is dirty in L1 of S_3, it is dirty in L1 of S_6 for the "miss" analysis.

Finally, we must consider how live caches are updated. Recall that a live cache must be updated whenever either of its corresponding levels, more specifically sets, is updated. To avoid duplicate updates, we only need to update a live cache once if both corresponding sets are updated as a result of the same memory access. For example, Figures 5 shows states S_7 and S_8 from Figure 1 extended with live caches.

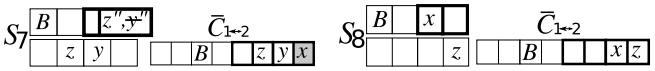

Figure 5. Hit analysis states for S_7 and S_8 from Figure 1

Recall that the transition from S_6 to S_7 in Figure 1 involved an update of both L1 instruction cache and L2 cache. Thus, the corresponding live cache sets in $\bar{C}_{1\leftrightarrow 2}$ must also be updated. Here, upon reading block B, both sets in $\bar{C}_{1\leftrightarrow 2}$ are updated. This results in eviction of block A and block x being moved to the last position in $\bar{C}_{1\leftrightarrow 2}$.

B. Benefits of Live Caches

Live caches have several benefits in that they help address the problems described in Section II-B. We now describe each of these benefits.

Improved precision*:* Recall in the example from Figure 1 that when reading x on line 8, the hit analysis could not guarantee a hit even though x is in both concrete states in S_7. In Figure 4, block x was in the second last position in the L1 data cache of S_5 and in the second last position in the L2 cache of S_3. Thus, in the resulting state, S_6, x took the second to last position (worst case) in the live cache $\bar{C}_{1\leftrightarrow 2}$. Now, consider the read to x again but using the state S_7 augmented with live caches from Figure 5. Since x is in the live cache $\bar{C}_{1\leftrightarrow 2}$, we know that in the worst case, x will hit L2. Thus, by adding live caches, an unknown, or worst case memory access, has been classified as a worst case L2 hit. For this example, we see that introducing live caches results in an access being classified as a hit of some cache level rather than a worst case memory access. Thus, live caches improve the overall precision of the analysis.

Write-back*:* To illustrate how these changes ensure soundness for the "hit" analysis, consider the transition from state S_7 to S_8 in Figure 5 as compared to the previous example in Figure 1. In the transition in Figure 5, two potentially dirty blocks are evicted from L1 (z and ~~y~~). Thus, L2 cache must be updated once for potential write-back (since loading in one new block can not result in two write-backs). Further, we update L2 again for the potential L1 miss of block x. Again we can not update L2 with x since the miss analysis can not guarantee L1 miss.

Hierarchy*:* As discussed previously, a sound analysis must model both instruction and data caches due to their impact on each other in unified cache levels. Thus, our analysis simultaneously analyzes both cache types. Also, the hierarchy as a whole contains more information than the levels do in isolation. Thus, our analysis introduces live caches which capture additional information available from the hierarchy as shown in Figures 4 and 5 with block x.

We now define the theory of our multi-level cache analysis. As is usual for an abstract interpretation based analysis [4], we first define the concrete behavior for a common type of cache hierarchy (Section IV). Then (Section V) we define the abstraction of the concrete behavior for multi-level caches and the formalism behind live caches.

IV. Concrete Cache Semantics

In this section, we define the concrete semantics for caches. That is, their *observable behavior*, not their physical behavior. Here, we focus on mainly-inclusive caches, a common multi-level cache policy (Pentium II, III, 4, etc). The discussion for other types of cache hierarchies is contained in our report [15]. Figure 6 shows our notation for concrete semantics. It is inspired from Ferdinand and Wilhelm [3], but extends it to multi-level caches.

n_x:	The **number of blocks** that fit into the Lx cache at one time. $n_x = \mathit{capacity}_x/\mathit{line\ size}_x$.
A_x:	**Associativity** of the Lx cache (A_x-way set associative). For direct mapped caches, $A_x = 1$ and for fully associative, $A_x = n_x$.
H:	$H = \langle C_1, \cdots, C_{\mathcal{N}} \rangle$, where $\mathcal{N}$ is the number of cache levels. State of the **cache hierarchy**.
C_x:	$C_x = (S_{1,x}, \cdots, S_{n_x/A_x,x})$ The x^{th} **level of cache** (L1, L2, etc) and consists of n_x/A_x cache line sets (or block sets).
$S_{i,x}$:	$S_{i,x} = \langle l^i_{1,x}, \cdots, l^i_{A_x,x} \rangle$ Represents an associative **cache set**. Sequence of cache lines, ordering defines the LRU order where the last line is the least recently used. $S_{i,x}(l^i_{j,x})$ is a look-up of the block in the j^{th} line in the i^{th} cache set of the Lx cache.
$l^i_{j,x}$:	The j^{th} **cache line** in the i^{th} associative set in the Lx cache (contains one memory block).
M_x:	Set of blocks $\{m_{1,x}, \cdots, m_{\mathcal{M}/\mathit{line\ size}_x,x}\}$, where $\mathcal{M}$ is the capacity of memory. M_x **models the memory** as viewed from the Lx cache. The line size changes this view of memory.
$m_{i,x}$:	The i^{th} **memory block** as viewed from the Lx cache (as large as the line size of Lx cache).
I:	***empty block*** indicates that no value exists in cache yet.
M'_x:	$M'_x = M_x \cup \{I\}$ **memory with the empty block**.
adr_x:	$adr_x : M_x \to \mathbb{N}$ function **mapping memory blocks** (as viewed from Lx cache) **to their start address** and is defined as $adr_x(m_{i,x}) = n$ where $n = \lfloor i(\mathit{line\ size}_x) \rfloor$.
set_x:	$set_x : M_x \to C_x$ **maps blocks to cache sets**. $set_x(m_{i,x}) = S_{j,x}$, where $j = adr_x(m_{i,x}) \mod (n_x/A_x) + 1$.
δ:	$\delta : C_x \times S_{i,x} \to \{true, false\}$ True if the input block is **dirty**.
$\preceq$:	Used for cache levels with different line sizes. e.g.: $m_{i,x} \preceq m_{i',x+1}$ means that all of $m_{i,x}$ is in $m_{i',x+1}$ Formally, $m_{i,x} \preceq m_{j,y} \Leftrightarrow adr_y(m_{j,y}) \leq adr_x(m_{i,x}) \leq adr_y(m_{j+1,y}), x < y$
$C_{x \vee y}$:	$C_{x \vee y} = (S_{1 \vee 1, x \vee y}, S_{1 \vee 2, x \vee y}, \cdots, S_{n_x/A_x \vee n_y/A_y, x \vee y})$ Cache level that contains blocks in either C_x, C_y, or both.
$A_{x \vee y}$:	**Associativity** of $C_{x \vee y}$. $A_{x \vee y} = \max(A_x, A_y)$.
$S_{i \vee k, x \vee y}$:	$S_{i \vee k, x \vee y} = \langle l^{i \vee k}_{1, x \vee y}, \cdots, l^{i \vee k}_{A_{x \vee y}, x \vee y} \rangle$ Represents an associative **cache set** in $C_{x \vee y}$. Sequence of cache lines, ordering defines the LRU order where the last line is the least recently used.
$l^{i \vee k}_{j, x \vee y}$:	The j^{th} **cache line** in the set corresponding to the i^{th} associative set in the Lx and k^{th} associative set in the Ly (a set of memory blocks).

Figure 6. Notation for Concrete Semantics

A. Concrete Semantics of Reads for a Cache Level

Single-level update*:* Since we are modeling the LRU replacement policy, we define how memory reads and writes effect the LRU order. Our update function ($\mathcal{U} : H \times M_x \to H$) for handling individual cache sets is borrowed from previous work [3]. It takes a cache level and a block and returns the updated cache as follows.

$$\mathcal{U}(C_x, m_{j,x}) =$$

$$\begin{cases} l^i_{1,x} \mapsto m_{j,x}, \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k-1,x}) | k \in \{2, \cdots, h\}, \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k,x}) | k \in \{h+1, \cdots, A_x\} & \text{if } \exists h, i : S_{i,x}(l^i_{h,x}) = m_{j,x} \\ \\ l^i_{1,x} \mapsto m_{j,x}, \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k-1,x}) | k \in \{2, \cdots, A_x\} & \text{otherwise} \\ \quad set_x(m_{j,x}) \to S_{i,x} \end{cases}$$

The first case updates the LRU order when the block is already in the cache level. The second case adds the block to the cache level and updates the LRU order .

Multi-level update*:* A technique based on mainly-inclusive caches can rely heavily on previous work since the replacement policies for each level are independent [13]. Such a technique was defined in previous work for instruction caches [2]. However, by considering levels in isolation the analysis ignores much information available in the hierarchy. Thus, we define multi-level semantics to gain knowledge from the hierarchy.

We now define the update function, $\mathcal{R}_x : H \times M'_x \to H$ which takes a cache level and a memory block and returns the updated cache level. For simplicity, reads and writes both use the same update function. The only difference is that a write will mark the block as dirty in the first level cache. This function is defined as

$$\mathcal{R}_x(m_{j,x}) =$$

$$\begin{cases} \mathcal{U}(C_x, m_{j,x}) & \text{if } \exists i, h : S_{i,x}(l^i_{h,x}) = m_{j,x} \\ \\ C_y \mapsto \mathcal{R}_y(m_{j',y}) \\ \mathcal{U}(C_x, m_{j,x}) & y = x+1,\ \text{if } set_x(m_{j,x}) \to S_{i,x},\ \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x} \wedge \neg\delta(S_{i,x}, l^i_{A_x,x}), m_{j,x} \preceq m_{j',y} \\ \\ C_y \mapsto \mathcal{W}_{S,y}(S_{k,y}) \\ C_y \mapsto \mathcal{R}_y(m_{j',y}) \\ \mathcal{U}(C_x, m_{j,x}) & y = x+1,\ \text{if } set_x(m_{j,x}) \to S_{i,x},\ \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x} \wedge \delta(S_{i,x}, l^i_{A_x,x}) \\ & m_{j,x} \preceq m_{j',y}, S_{i,x}(l^i_{A_x,x}) \to m_{e,x} \\ & set_y(m_{e,x}) \to S_{k,y} \end{cases}$$

In the first case, the block is in the cache set, so LRU order of the set is updated. In the second case, the block is not in the cache set and the evicted block is clean. A read is issued to the next cache level (Lx+1). Then, LRU order of the current set in Lx is updated. In the final case, the block is not in the cache set and the evicted block is dirty. LRU order of the next cache level (Lx+1) is updated for the dirty block which is marked as dirty (via $\mathcal{W}_{S,y}$ defined in Section IV-B). Then, a read of the new block is issued to the next level of cache (Lx+1). Finally, the LRU order for the current level (Lx) is updated.

B. Concrete Semantics of Writes for a Cache Set

Since actual values are not relevant for cache analysis, the only difference between reads and writes is that writes mark the block as dirty. We use this fact to simplify definition of write. The semantics of write uses the read semantics to update the LRU order to move the written block to the first location and marks this block as dirty as shown below.

$$\mathcal{W}_{S,x}(S_{i,x}) = S_{i,x} \oplus \delta(l^i_{1,x}) \mapsto \mathit{true}$$

Here $\oplus$ is the overriding function for finite sets. It replaces the previous state of the cache line with the new state.

C. Read/Write Concrete Semantics for Cache Hierarchies

We now discuss reads and writes for an entire cache hierarchy. A formal description is in our technical report [15].

For a read, first the correct set is found for the L1 cache. Next, a read to L1 cache is issued for the block containing the reference. Since reads are defined recursively, the L1 read function will handle calling reads for the higher cache levels. Write is similar to read, except that after the read to L1 is issued the first line in the L1 set is marked as dirty.

Finally one more addition is needed to enable proper concretization of live caches. This addition, denoted $C_{x\vee y}$, is similar to a live cache in that it captures blocks in two different cache levels, Lx and Ly. Like live caches, there is an associative set in $C_{x\vee y}$ for all combinations of sets in C_x and C_y that may contain common addresses. Also, the associativity of the sets is equal to the larger of the two levels ($A_{x\vee y} = \max(A_x, A_y)$).

For simplicity, instead of re-defining the update functions to include this new model, we re-create the state for each of these caches, $C_{x\vee y}$, after each update. Creating the state for each $C_{x\vee y}$ is as simple as copying C_x and C_y into $C_{x\vee y}$ where blocks are put at the same proximity to eviction as in the standard cache level. This creation is defined as follows.

$$m_{j,w} \preceq l^{i\vee k}_{A_{x\vee y}-h,x\vee y} \in S_{i\vee k,x\vee y} \in C_{x\vee y} \in H \iff \begin{array}{l} m_{j,w} \preceq l^{i}_{A_x-h,x} \in S_{i,x} \in C_x \in H \\ \vee \\ m_{j,w} \preceq l^{k}_{A_y-h,y} \in S_{k,y} \in C_y \in H \end{array}$$

V. Abstract Cache Semantics

This section discusses our abstract semantics including notations, join and update functions, and hit/miss analysis. Most notations are similar to Figure 6. Below we outline major differences.

Single block to Set*:* Since the run-time path is unknown, lines may hold more than one value (illustrated in Figure 1). Therefore, the cache line look-up becomes $\mathcal{S}(l^i_{j,x}) : \mathcal{S} \rightarrow \mathcal{P}(M')$ which simply means that a look-up returns a set of blocks rather than a single block.

Live cache notation*:* The notation $\bar{\mathcal{C}}_{x\leftrightarrow y}$ denotes a live cache corresponding to Lx and Ly caches. Recall that with each live cache set, we have two corresponding sets one in Lx and one in Ly cache. Suppose we have a set in live cache corresponding to sets $\mathcal{S}_{i,x} \in C_x$ and $\mathcal{S}_{k,y} \in C_y$. Then, to refer to this set we use $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y}$. As mentioned previously, this set ($\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$) is updated whenever either corresponding set is updated ($\mathcal{S}_{i,x}$ or $\mathcal{S}_{k,y}$).

Write-back*:* The function δ needs to be updated to account for two things. First, cache lines hold sets in the abstract semantics. Second, for soundness, we need to know if a block *may* be dirty ($''$ in Figure 1), for precision, we need to know if a block *must* be dirty ($'$ in Figure 1). First, $\delta : M \times \mathcal{S} \rightarrow \{true, false\}$ takes a block and a cache line and returns true if and only if that block must be dirty. Second, $\bar{\delta} : M \times \mathcal{S} \rightarrow \{true, false\}$ is the same as δ except it returns false if an only if the block can not be dirty.

Hierarchy*:* Abstract hierarchy state $\mathcal{H} = \{\mathcal{C}_1, \ldots, \mathcal{C}_{\mathcal{N}}\} \cup \mathcal{V}$, where $\mathcal{V}$ is the set of live caches.

A. Hit/Miss Analysis

As shown in Section II, the analysis consists of two parts, a hit (must) and miss (may) analysis [2], [4]. For each memory access, the hit analysis reports either hit or unsure (for each level). Similarly, the miss analysis reports either miss or unsure. Each analysis is sound (if the hit analysis reports a hit for an access, it will definitely be a hit). Combining the two we determine for each access if the access will hit in Lx, the access will miss in Lx, or unknown if the access will hit or miss in Lx. An advantage of incorporating live caches is that we can now determine if the access in the worst case will hit Lx (when $m_{j,w}$ is in $\bar{\mathcal{C}}_{x\leftrightarrow x}$) or hit L$y$ (when $m_{j,w}$ is in $\bar{\mathcal{C}}_{x\leftrightarrow y}$). This new information results in a tighter upper bound since we classify additional accesses as hits. Formal treatment of miss analysis is omitted here, but included in our technical report [15].

Like previous work [2], [3] we differentiate between first-hit/miss and always-hit/miss. Since many blocks will not be in cache for their first use but will be for accesses thereafter, this reduces the loss of precision introduced by compulsory (cold) misses. For example, in Figure 1 (if running example has a loop), the first iteration of the loop in lines 3-4 results in a miss to load the data (y). However, y is used throughout all other iterations resulting in hits. Differentiating between first/always-hit/miss results in 1 miss and 7 hits for the first 8 iterations instead of 8 unknowns (worst case miss).

B. Join function

The "join" function is used to combine states from two program paths. Since we are designing a sound analysis, this must be a worst case combination. This is illustrated in Figure 1. Compared to join functions of previous work [2], our definition handles live caches and write-back soundly.

The join function for our hit analysis is denoted as $\bigwedge_{hit}(\mathcal{H}', \mathcal{H}'') = \mathcal{H}$. We define the behavior of this function case by case, first informally, then formally. Here, $\mathcal{S}_{i,x} \in \mathcal{C}_x \in \mathcal{H}$ and $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y} \in \mathcal{H}$. Also, assume that cache sets and levels marked with $'$ belong to $\mathcal{H}'$ and those marked with $''$ belong to $\mathcal{H}''$. Since $\bigwedge(\mathcal{H}', \mathcal{H}'') = \bigwedge(\mathcal{H}'', \mathcal{H}')$ we remove extra cases without loss of generality. If a case does not appear in the definition, it is ignored and thus does not exist in the joined cache hierarchy.

- The first case is for standard abstract caches, $\mathcal{C}_x$, when two references exist on the same level. In this case, like previous work [4], we take the later position ($\max(a, b)$) and set the dirty (δ) and may dirty ($\bar{\delta}$) states of the block.
  If $\exists a, b : m_{j,w} \preceq \mathcal{S}'_{i,x}(l^i_{a,x}), m_{j,w} \preceq \mathcal{S}''_{i,x}(l^i_{b,x})$
  then $m_{j,w} \preceq \mathcal{S}_{i,x}(l^i_{t,x})$ where $t = \max(a, b)$,
  $\delta(m_{j,w}, l^i_{t,x}) = \delta'(m_{j,w}, l^i_{a,x}) \wedge \delta''(m_{j,w}, l^i_{b,x})$,
  $\bar{\delta}(m_{j,w}, l^i_{t,x}) = \bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x})$.
- The second case deals with write-back when a dirty reference exists in one case ($\mathcal{H}'$) but not the other ($\mathcal{H}''$). We must capture that a block in this location may be dirty but since the block doesn't exist in both cases, we cannot

keep it in the joined state ($\mathcal{H}$). Thus, we use an empty block (I) that is marked maybe dirty $\bar{\delta}(I, l^x_{a,i})$.

If $m_{j,w} \preceq \mathcal{S}'_{i,x}(l^i_{a,x}) \wedge \bar{\delta}'(m_{j,w}, l^i_{a,x}) \wedge \nexists b, y, k : m_{j,w} \preceq \mathcal{S}''_{k,y}(l^k_{b,y})$
then $I \preceq \mathcal{S}_{i,x}(l^i_{a,x})$where $\bar{\delta}(I, l^i_{a,x})$.

- The third case is where the same block is in different levels in the two cases. We find the shortest distance to eviction between the two blocks and put the block into the live cache the same distance from eviction.

If $\exists a, b : m_{j,w} \preceq \mathcal{S}'_{i,x}(l^i_{a,x}) \wedge \neg\delta'(m_{j,w}, l^i_{a,x}) \wedge m_{j,w} \preceq \mathcal{S}''_{k,y}(l^k_{b,y})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$where $t = A_{x,y}$–min$(A_x$–a, A_y–$b), x < y$,
$\neg\delta(m_{j,w}, \bar{l}^{i,k}_{t,x,y}), \bar{\delta}(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^k_{b,y})$.

- The fourth case handles states where the block is in different levels (Lw and Ly) in two states and the lower block is dirty. In this case, we take the age of the block in the higher cache. This is safe since the block in the lower cache will be written to the higher cache on eviction. Thus, it will be moved to the first spot in the higher cache.

If $\exists a,b,w,p : m_{j,w} \preceq \mathcal{S}'_{p,w}(l^p_{a,w}) \wedge \delta'(m_{j,w}, l^p_{a,w}) \wedge m_{j,w} \preceq \mathcal{S}''_{k,y}(l^k_{b,y})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$ where $t = A_{y,y} - (A_y - b)$
$\delta(m_{j,w}, \bar{l}^{k,k}_{t,y,y}) = \delta''(m_{j,w}, l^k_{b,y}), \bar{\delta}(m_{j,w}, \bar{l}^{k,k}_{t,y,y}), x = y$.

- The fifth case is when the block exists in the same level of live cache in both cases. This is similar to the first case except we are dealing with live caches instead of abstract caches. Like the first case, we take the later position.

If $\exists a, b : m_{j,w} \preceq \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{a,x,y}), \wedge m_{j,w} \preceq \bar{\mathcal{S}}''_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{b,x,y})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$ where $t = \max(a, b)$,
$\delta(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \delta'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \wedge \delta''(m_{j,w}, \bar{l}^{i,k}_{b,x,y})$
$\bar{\delta}(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, \bar{l}^{i,k}_{b,x,y})$.

- The sixth case is when a block exists in live cache in one state and an abstract cache in the other. The block in the abstract cache is also clean. This case is similar to the third, in that we find the block closest to eviction. We give the block in the joined state the same proximity to being evicted in the live cache.

If $\exists a, b : m_{j,w} \preceq \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{a,x,y}) \wedge m_{j,w} \preceq \mathcal{S}''_{i,x}(l^i_{b,x}) \wedge \neg\delta''(m_{j,w}, l^i_{b,x})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$ where $t = A_{x,y}$–min$(A_{x,y}$–a, A_x–$b), x < y$,
$\neg\delta(m_{j,w}, \bar{l}^{i,k}_{t,x,y}), \bar{\delta}(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}),$.

- The seventh case is the same as the previous except that the block in live cache is dirty. In this case (since the dirty block is in a lower level), like the fourth case, we take the position of the block in the higher level. The higher level is the live cache since the block is in the lower of the two levels that the live cache corresponds to.

If $\exists a, b : m_{j,w} \preceq \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{a,x,y}) \wedge m_{j,w} \preceq \mathcal{S}''_{i,x}(l^i_{b,x}) \wedge \delta''(m_{j,w}, l^i_{b,x})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$ where $t = a, \neg\delta(m_{j,w}, \bar{l}^{i,k}_{t,x,y})$,
$\bar{\delta}(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}), x < y$.

- The eighth case is like the previous except the dirty block is in the higher of the two levels corresponding to the live cache. Like the sixth case, take the worst case position between these two blocks and place it in the live cache.

If $\exists a, b : m_{j,w} \preceq \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{a,x,y}) \wedge m_{j,w} \preceq \mathcal{S}''_{i,y}(l^k_{b,y})$
then $m_{j,w} \preceq \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})$ where $t = A_{x,y} - \min(A_{x,y} - a, A_y - b)$
$\delta(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \delta'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \wedge \delta''(m_{j,w}, l^k_{b,y})$,
$\bar{\delta}(m_{j,w}, \bar{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \bar{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^k_{b,y})$.

Soundness*:* The proof of soundness of the join function uses standard analysis of its cases [15].

C. Abstract Update

A major difference between the concrete and abstract semantics is that in the abstract semantics we have that cache lines may contain sets of possible blocks. This change requires minor changes to the notation to look for membership in the sets of blocks rather than comparing to a single block. We also need to consider empty sets.

Update using live caches for unknown address references*:* Since we are interested in an analysis for all types of caches, we must consider the case where we do not know which cache set a reference will belong to. For example, consider the following code:

```
1 void foo(int x){
2   int * a = (int *) malloc(sizeof(int));
3   *a = x * 2;
4   if(*a > 10){ ...
```

In this simple function, memory is allocated and the pointer, `a`, is set to the address of this new memory. Next, the location pointed to by `a` is modified based on the function input `x`. In the last line, we read the location pointed to by `a`. If we are unable to determine the actual address of this new memory statically, we can not determine which cache set it will belong to. Since previous analysis techniques are designed for instruction caches if we use a previous analysis techniques we lose information about `a`. As a result, we do not have any information about `a` on the second line. However, we can see that the write to `a` will be a hit.

Live caches help solve this problem. We now describe a special live cache set ($\bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}$) for each live cache corresponding to a single level ($\bar{\mathcal{C}}_{x\leftrightarrow x}$). Note that we only require one additional set per cache level.Since there are typically at most a few cache levels, we need few extra sets to handle this important case. This new set corresponds to all live cache sets in this cache. This set is updated whenever any set in $\mathcal{C}_x$ is updated. Since cache sets are typically only 8 or 16 lines wide, we typically only need in the tens of new cache lines for this extension to live caches. References in this new set are likely to have a short lifetime since sets are typically small. However, these sets catch cases like those illustrated previously which occur frequently in practice.

Update for live cache*:* We now define how live caches are updated. Recall that a live cache corresponds to two separate caches. Also, live cache sets correspond to two sets, on in each of these two caches. Thus, whenever we update a set in the Lx cache, we must update all live cache sets that correspond to the set in Lx. This is defined as

$$\mathcal{U}(\mathcal{S}_{i,x}, B) =$$

$$\begin{cases} \forall y, k \text{ s.t. } \bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y}, \mathcal{R}_{x,y}(\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}, B) \\ \forall y, k \text{ s.t. } y \neq x \wedge \bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y}, \mathcal{R}_{x,y}(\bar{\mathcal{S}}_{y,k\leftrightarrow x,i}, B) \\ \mathcal{R}_{x,x}(\bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}, B) \\ \dots \end{cases} \dots$$

Where $B \subseteq M$ is a set of blocks to be read. The updates in this equation are done regardless of the case for updating and the other updates done (represented by $\dots$). The restriction

of $y \neq x$ in the second set of updates ensures that live caches corresponding to single cache levels are not updated twice.

The live cache update is defined as:

$$\mathcal{R}_{x,y}(\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}, B) =$$

$$\begin{cases} \mathcal{U}(\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}, B) & x < y \\ \forall p : B_e^p \neq \phi \vee \exists m_{e,w} \in B_e : & x = y, i = k \\ \bar{\delta}(m_{e,w}, \bar{l}^{i,k}_{A_{x,y},x,y}) \wedge & \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{A_{x,y},x,y}) \rightarrow B_e \\ set_z(m_{e,w}) \rightarrow S_{p,z}, & z = y+1, B_e^p = \{m_{e,w} \in B_e | \\ \quad \mathcal{R}_{z,z}(\bar{\mathcal{S}}_{z,p\leftrightarrow z,p}, B_e^p) & \delta(m_{e,w}, \bar{l}^{i,k}_{A_{x,y},x,y}), \\ \mathcal{U}(\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}, B) & set_z(m_{e,w}) \rightarrow S_{p,z}\} \end{cases}$$

$$\mathcal{R}_{x,x}(\bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}, B) =$$

$$\begin{cases} \mathcal{R}_{z,z}(\bar{\mathcal{S}}_{z,\forall\leftrightarrow z,\forall}, B_e^d) & \bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}(\bar{l}^{\forall}_{A_x,x}) \rightarrow B_e \\ \mathcal{U}(\bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}, B) & z = x+1, B_e^d = \{m_{e,w} \in B_e | \delta(m_{e,w}, \bar{l}^{\forall}_{A_x,x})\} \end{cases}$$

This definition relies on update function, $\mathcal{U}$, similar to that for concrete cache sets given in Section IV-A. In this case we instead update live cache sets. Like abstract cache states, each line in a live cache set may contain a set of blocks.

In the first case, since $x < y$ the LRU order is updated with the new set of blocks B (we never have blocks that are definitely dirty in a live cache connected to two separate levels). In the second case, all blocks being evicted from the live cache are written back to the next level of live cache.

In practice, the number of updates can be reduced slightly while maintaining soundness. For example, suppose a read misses both L1 and L2. According to our theoretical rules, the live cache set corresponding to both the set in L1 and in L2 would be updated. Typically, this is the same set in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$. Thus, there is no reason to update this set twice since both levels are updated only once (unless a block is evicted from L1 to make room for the new block). Thus, we can simply update the corresponding set in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ once.

D. Concretization

As is standard, for determining the meaning of an abstract state, a concretization function, γ, is used. The concretization of standard abstract cache levels, $\mathcal{C}_x$, follows previous work [3] and is relatively straightforward. Live caches, however, are somewhat more challenging.

Suppose we have a live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$. A block in this cache represents a worst case L2 hit. However, we can not simply put this block in the concrete L2 cache since it might not actually be in L2.

To address this problem, recall the extension $C_{x\vee y}$ discussed previously in Section IV-C. Since this portion of the concrete state contains blocks that are in either Lx, Ly, or both Lx and Ly, live cache blocks are concretized safely into $C_{x\vee y}$. To concretize a block in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$, we place it in $C_{1\vee 2}$ at the same proximity to eviction as in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$. This is defined as follows. Suppose $H = \gamma(\mathcal{H})$. The portion of concretization for live caches is defined as follows.

$$m_{j,w} \preceq \bar{l}^{i,k}_{A_{x,y}-h,x,y} \in \bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y} \in \mathcal{H}$$
$$\Rightarrow m_{j,w} \preceq l^{i\vee k}_{A_{x\vee y}-h,x\vee y} \in S_{i\vee k,x\vee y} \in C_{x\vee y} \in H$$

Termination of the Analysis*:* Like previous work [2], our analysis is guaranteed to terminate. For a full discussion we refer to our technical report [15].

VI. Evaluation

The goals of our analysis were to handle cache hierarchies, handle write-back and improve precision. So far, via the definition of our abstract domain for multi-level caches, we have shown that our approach can handle cache hierarchies and write-back. In this section we evaluate our third claim that our approach improves the precision of multi-level cache analysis over previous work [2].

A. Improved Precision

To verify that live caches improve precision in practice, we compared two static analysis approaches, one with live caches and one without. The cache analysis was implemented as part of our prototype tool which analyzes program binaries. At this time, the entire analysis process is not completely automated as it requires determining bounds on loop iterations by hand for many loops. For our experimentation, loop bounds were determined either from comments in the benchmark source code or by manual inspection of the source code. Like previous work [2], [9], we used the WCET benchmarks maintained by the Mälardalen WCET research group [14]. We ran our prototype on all 32 of these benchmarks. For these benchmarks, we observed the average space overhead to be 95% (max 120%) and time overhead to be 57.8%. This is primarily because we have not yet attempted to produce an optimized implementation. A more optimized representation of live caches will drastically reduce both space and time overhead, however, producing such implementation was not the focus of this paper.

To determine the performance of our analysis we used a cache configuration also used by previous work [2] (L1: 8k, 4-way, L2: 64k, 8-way). We also use the same cost in cycles for accessing levels in the memory hierarchy (L1 hit: 1 cycle, L2 hit: 10 cycles, Memory access: 100 cycles). The improvement in precision for overall cache/memory access time for each benchmark is shown in Figure 7.

This data shows that introducing live caches into the analysis resulted in up to a 31% reduction in worst case memory access time. The benchmarks bsort100, cnt, cover, edn, fac, fibcall, fir, fdct, loop3, nsichneu, qsort-exam, and select did not show any improvement. We observed that these benchmarks typically have few joins. For benchmarks reported in previous work [2] we observed an average improvement of 6.4% and the average improvement over all benchmarks with non-zero improvement was 6.3%. Overall, 62.5% of the benchmarks showed some improvement in precision. As the box-plot shows, there is a high variability between benchmarks (standard deviation of 6.1%).

Improvement>5%		0%<Improvement<5%	
Benchmark	**Improv.**	**Benchmark**	**Improv.**
qurt	31.0%	compress	4.7%
sqrt	23.5%	minver	3.5%
insertsort	19.5%	janne_complex	3.1%
lcdnum	14.0%	jfdctint	3.0%
matmult	12.1%	adpcm	2.9%
ns	11.0%	lms	2.6%
statemate	9.2%	minmax	2.4%
duff	7.3%	fft1	1.1%
crc	7.2%	ludcmp	0.3%
bs	5.8%	expint	0.3%

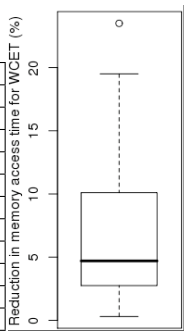

Figure 7. Reduction in worst case cache/memory access time due to live caches. L1: 8kb 4-way, L2: 64kb 8-way. Left: improvement per benchmark, Right: boxplot for those benchmarks with non-zero improvement.

B. Impact of program characteristics

We performed a multiple linear regression analysis using the Lindeman, Meranda and Gold (LMG) method to determine relationships between program characteristics and precision improvement. The strongest model includes the following predictors: floating point computations (y/n), if-then-else structures (count), size of the binary, joins in the binary (count), and accuracy of analysis without live caches.

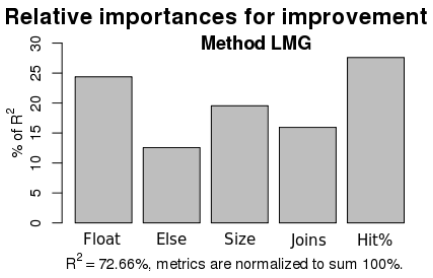

Figure 8. Relative importance, Improv > 0

These results are shown in Figure 8. Besides the characteristics shown in Figure 8, we also considered others such as number of loops, nesting of control structures, etc, but none of them were found to be strong factors. The figure shows five predictors that were the strongest with an adjusted correlation coefficient, R^2, of 0.6 (std $R^2 = 0.73$).

Number of joins and complex branching*:* As expected, we found these to be a large factor. In other words, the more joins and branching, the higher the expected improvement, which is encouraging because these are the cases where precision matters.

Size*:* Another encouraging result was that as the program size increased precision improvement also increased, which suggests that for programs larger than the WCET benchmarks, we could see similar, if not better, precision.

Floating point operations and accuracy of previous analysis*:* It was somewhat surprising to see these factors as strong predictors. Float programs have an average lower miss rate for instruction and unified caches than non-float programs (although slightly higher data cache miss rates) [16]. Thus, the previous technique should have higher accuracy for these benchmarks. Increased hit percentage of previous analysis implies some increase in knowledge which means greater knowledge for our analysis as well. More information will increase the precision of our cache model.

Scalability*:* We found a strong logarithmic correlation between space overhead and number of joins. This suggests that our approach should scale as well as the previous technique [2]. We also found a strong linear correlation ($R^2 = 0.94$) between run-time and number of joins. This means that for large complex programs, we should not see an explosion of run-time, rather a more gentle increase.

C. Impact of hardware configuration

Besides the simple cache configuration presented before, we tested a variety of cache configurations to determine if target platform has an impact on precision improvement of our analysis. We include cache configurations found in processors ranging from early models with multi-level caches to modern processors. Our results show several noticeable trends which we now summarize.

First, we found a positive correlation between the size of L2 cache and precision improvement. However, this improvement is small. For example, when quadrupling L2 size, experiments showed on average that accuracy only improved by about 0.13%. This result is however pleasant in that it suggests that our analysis is useful for large caches as well. L1 cache sizes had no significant impact on precision.

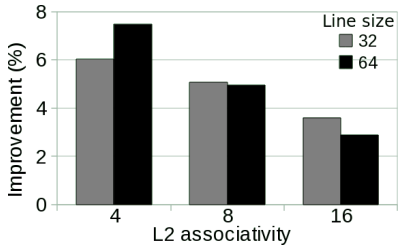

Figure 9. Average improvement for varying L2 associativity and linesize (L1: 32k, 4-way. L2: 256k).

We found that associativity plays the largest role in impacting performance. Figure 9 shows average improvement for a variety of L2 associativity levels and linesizes. We see a clear negative correlation between associativity and improvement, however, even for highly associative caches, we see an improvement of nearly 4% on average. The figure also depicts the results for varying linesize. We did not find a statistically significant difference in average improvement for 32byte vs 64byte linesizes.

Summary*:* In summary, we observed the following.

- Most programs see precision improvement.

- As program complexity and size increase, precision generally increases.
- Caches from small to large see precision improvement.
- Time and space overheads scale well for program size and complexity.

VII. Related Work

Existing work either focuses on instruction caches [2], [4]–[8] or on data caches [9]–[12]. In contrast, our technique works for both instruction and data caches.

Most similar among instruction cache techniques is that of Hardy and Puaut [2] which makes use of abstract interpretation to address multi-level instruction caches. However, as mentioned previously, this technique is only for mainly-inclusive instruction caches. We address all types of multi-level caches as well as data caches. This includes handling complications caused by write-back and other multi-level cache policies. We have also developed live caches which improve upon the accuracy of their technique by capturing memory references that exist in different cache levels.

Another technique was proposed by Sen and Srikant [9] which uses abstract interpretation to predict data cache behavior. They make use of previous work to track addresses [17]. This work does not consider multi-level data caches and thus does not handle write-back. Our work considers multi-level caches and write-back for data caches.

The original technique for predicting cache behavior using abstract interpretation was proposed by Alt *et al.* [4]. Like the work by Hardy and Puaut [2], we make use of the ideas presented in this paper to form the basis of our analysis. Our analysis extends this work by handling multi-level caches and write-back for these multi-level caches.

There are several ideas to handle tracking address values in programs [10], [17]. This is an orthogonal issue to cache behavior analysis. Similar to previous work on cache analysis our technique also makes use of these ideas.

Yan and Zhang's analysis technique [6] considers interference between threads [18], but it doesn't handle both instruction and data caches. We don't consider interference but handle both type of caches.

VIII. Conclusion

Precise cache abstractions are needed to get correct worst case execution time by static analysis. Existing cache analyses [2], [4]–[12] do not provide precise enough abstractions for realistic cache hierarchies which include split and unified multi-level caches as well as write-back. We have discussed a multi-level cache analysis enabled by a new abstraction, live caches, that solves these problems. Evaluation of our technique using standard WCET benchmarks shows an average of 6.3% reduction in 66.5% benchmarks. Since memory access time is often a significant portion of the execution time of a system, we expect to see similar improvements in larger real-time systems. Since estimating WCET is important for real-time systems we expect our technique to lead to resource savings in their implementation.

Acknowledgments*:* This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CCF-08-46059.

References

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. Emb. Comp. Syst.*, vol. 7, no. 3, 2008.

[2] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *RTSS*, 2008.

[3] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-timesystems," *Real-Time Syst.*, vol. 17, no. 2-3, 1999.

[4] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *SAS*. Springer-Verlag, 1996.

[5] J. Yan and W. Zhang, "WCET analysis of instruction caches with prefetching," in *LCTES*, 2007.

[6] ——, "WCET analysis for multi-core processors with shared L2 instruction caches," in *RTAS*, 2008.

[7] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 3, 1999.

[8] C.A. Healy *et al.*, "Bounding pipeline and instruction cache performance," *IEEE Trans. Comput.*, vol. 48, no. 1, 1999.

[9] R. Sen and Y. N. Srikant, "WCET estimation for executables in the presence of data caches," in *EMSOFT*, 2007.

[10] R.T. White *et al.*, "Timing analysis for data caches and set-associative caches," in *RTAS*, 1997.

[11] C. Cascaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *ICS '03*, 2003.

[12] V.-M. Panait, A. Sasturkar, and W.-F. Wong, "Static identification of delinquent loads," in *CGO*, 2004, p. 303.

[13] M. Zahran, "Cache replacement policy revisited," in *WDDD held in conjunction with ISCA*, June 2007.

[14] "WCET project/benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[15] T. Sondag and H. Rajan, "An abstract domain for multi-level caches," *Technical Report Iowa State University, Department of Computer Science, 09-20b*, April 2010.

[16] J. F. Cantin and M. D. Hill, "Cache performance for selected SPEC CPU2000 benchmarks," *SIGARCH Comput. Archit. News*, vol. 29, no. 4, pp. 13–18, 2001.

[17] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *In CC*. Springer-Verlag, 2004.

[18] S. Lim *et al.*, "An accurate worst case timing analysis for risc processors," *IEEE Trans. Softw. Eng.*, vol. 21, no. 7, 1995.