4-1-2010

An Abstract Domain for Multi-level Caches

Tyler Sondag
Iowa State University, tylersondag@gmail.com

Hridesh Rajan
Iowa State University

Recommended Citation

Sondag, Tyler and Rajan, Hridesh, "An Abstract Domain for Multi-level Caches" (2010). *Computer Science Technical Reports*. 247.
http://lib.dr.iastate.edu/cs_techreports/247

An Abstract Domain for Multi-level Caches

Abstract

Many program analyses and optimizations rely on knowledge of cache behavior. The precision of the underlying cache model is increasingly important with the recent uptake of multi-core and many-core architectures for two reasons. First, per-core cache sizes generally decrease as the number of cores becomes large resulting in more cache misses. Second, large scale sharing of the communication bandwidth to memory increases contention resulting in greater cost of cache misses. We present a sound technique for cache behavior analysis that handles instruction and data caches as well as a variety of multi-level cache policies. The resulting analysis is applicable to current general-purpose processors. Our technique relies on a new abstraction, live caches which model relationships between cache levels to improve accuracy of multi-level cache analysis. In an existing many-core cache configuration, live caches improve L2 hit accuracy by an average of 5.7%. Among others, this reduces the upper bound on memory accesses for worst case execution time (WCET) by an average 6.4%.

Keywords

abstract interpretation, cache behavior, program analysis, multi-level cache

Disciplines

Programming Languages and Compilers

A Theory of Reads and Writes for Multi-level Caches

Tyler Sondag and Hridesh Rajan

TR #09-20b
Initial Submission: April 1, 2010.

Keywords: static analysis, abstract interpretation, cache, behavior

CR Categories:
D.4.8[*Programming Languages*] Performance - Modeling and prediction
D.3.1[*Programming Languages*] Formal Definitions and Theory - Semantics
F.3.2[*Theory of Computation*] Semantics of Programming Languages - Program analysis

Submitted.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

An Abstract Domain for Multi-level Caches

Tyler Sondag and Hridesh Rajan

Iowa State University, Ames, IA 50011,
{sondag,hridesh}@cs.iastate.edu

Abstract. Many program analyses and optimizations rely on knowledge of cache behavior. The precision of the underlying cache model is increasingly important with the recent uptake of multi-core and many-core architectures for two reasons. First, per-core cache sizes generally decrease as the number of cores becomes large resulting in more cache misses. Second, large scale sharing of the communication bandwidth to memory increases contention resulting in greater cost of cache misses. We present a sound technique for cache behavior analysis that handles instruction and data caches as well as a variety of multi-level cache policies. The resulting analysis is applicable to current general-purpose processors. Our technique relies on a new abstraction, *live caches* which model relationships between cache levels to improve accuracy of multi-level cache analysis. In an existing many-core cache configuration, live caches improve L2 hit accuracy by an average of 5.7%. Among others, this reduces the upper bound on memory accesses for worst case execution time (WCET) by an average 6.4%.

1 Introduction

Emerging multi-core [13] and many-core [5] architectures such as the TILE64 processor [11] pose new problems and make some existing problems more important. One problem that is becoming increasingly important is classifying read/writes into those from the cache vs. those from the memory. This is due to two reasons. First, cache sizes are decreasing. Each core in a many-core architecture generally has smaller cache sizes than those in single or multi-core processors. For example, the TILE64 processor has 8Kb and 64Kb of L1 and L2 data cache respectively, whereas the Intel's Core 2 Duo processor, has 32Kb and 4Mb of L1 and L2 data cache respectively. This reduction in cache size directly translates to increase in cache miss rate [19]. Second, cost of cache misses is increasing due to memory bus contention, which is primarily because memory bus bandwidth is not keeping up with the number of cores in a CPU [9, 21]. Thus, frequent main memory access by many cores may easily congest the bus. As a result, cache analysis useful for multi-core and many-core architectures must:

- **Improve precision**, because classifying an L2 hit (cost$\simeq$15 cycles [31]) as a memory access (cost$\simeq$500 cycles [31]) may drastically increase computed upper bounds.
- **Handle cache hierarchies**, because for analysis and optimizations it is vital to reduce both low level and high level cache misses [9, 21]. Increasing L1 misses might be acceptable if more L2 hits are achieved, thereby keeping memory references on chip.
- **Handle Write-back**, because data cache levels with write-back will impact the behavior for unified (instruction+data) cache levels and thus is necessary for soundness.

- **Handle multi-level cache policies**, because a variety of multi-level cache policies exist in practice (ex: inclusive [16] in the core i7 [17], exclusive [32] in the AMD Phenom II [2], and mainly-inclusive in the Intel Core 2 Duo)

Our contribution is a multi-level cache analysis which addresses these requirements. Key to our multi-level cache analysis is the notion of *live caches*. A Live cache is an abstraction that captures information about memory blocks in the cache hierarchy which were lost in the previous multi-level analysis [15]. Therefore, live caches increases the precision over the previous work. Furthermore, live caches give more precise characterization of write-back behavior. The technical underpinnings of this work include:

- the definition of live caches and illustration of their behavior,
- an abstract interpretation for multi-level instruction and data cache analysis,
- proof of soundness of live caches which enables our abstract interpretation, and
- empirical results which show a 5.7% increase in known L2 cache hits for the mainly-inclusive cache policy.

This precision improvement benefits analyses and optimizations that rely on cache behavior model, e.g., we found that applying our technique reduces upper bound on cache and memory access times by 6.4% for benchmarks presented in previous work [15].

2 Background on Cache Analysis as Abstract Interpretation

In this section we give a brief overview of a typical cache analysis using abstract interpretation starting with basic terminology for caches, their structure and behavior.

2.1 Caches

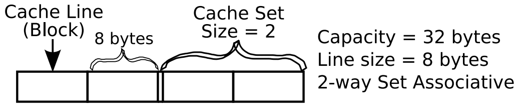


Fig. 1. A cache with two sets. A block in the first half of memory can only exist in the first set. Each set may hold two blocks (2-way set associative).

Cache Terminology . Basic terms are illustrated in Figure 1 and described below.

- Lx "level x" specifies the cache level we are referring to. Lower levels are closer to the processor and thus faster, but smaller.
- $\mathit{capacity}_x$ denotes the size in bytes of the Lx cache.
- $\mathit{line\ size}_x$ (block size) is the number of bytes loaded into the Lx cache upon a miss. We assume that as we go farther from the CPU, the line size is non-decreasing ($\mathit{line\ size}_x \leq \mathit{line\ size}_{x+i}$).
- Block refers to a segment of memory. As viewed from the Lx cache, there are $\mathcal{M}/\mathit{line\ size}_x$ blocks of memory, where $\mathcal{M}$ is the capacity of memory.
- Associativity: the number of cache lines a memory block may reside. For example, in a 2-way set associative cache, a block may be in one of two cache lines. A 1-way associative cache is direct mapped and $\mathit{capacity}_x/\mathit{line\ size}_x$-way is fully associative.

- Hit/Miss: When a memory location is in a cache level, it is a hit; otherwise, a miss.
- Write-back: It is a common technique used to reduce memory accesses when a block in cache is modified. With write-back, when a modified block is evicted from cache, the contents are written to the next cache level or memory. Write-back is common in most data caches such as those in the AMD Phenom II [2], the Intel Core 2 Duo, the Intel Core i7 [17], etc.
- Replacement strategy: A technique to determine which block to replace when a cache is full and an additional block is needed. Like previous work [3, 15], we use the least recently used (LRU) policy where the block accessed least recently is removed.

2.2 Abstract Interpretation based Cache Analysis

Figure 2 illustrates an abstract interpretation of cache behavior for a two level cache hierarchy. First level is a split (instruction separate from data) 2-way associative cache. Second level 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 this example we assume a mainly-inclusive cache policy. 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.

```
1 A input(*x)
2 A if(!*x)//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     *x += arr[i];
8 B output(*x);
```

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

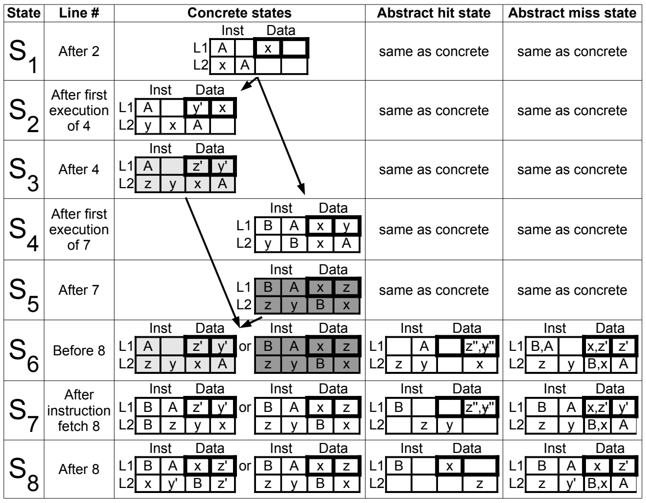


Fig. 2. Abstract Interpretation for cache behavior. *Left:* code example. Letters after line numbers indicate instruction cache block containing the code on that line. Table shows variable to cache block mapping *Right:* cache contents for each state. Loop unrolling is assumed.

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). Similar to previous work [3, 15], 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.

After analyzing lines 1-2, state S_1 results. L1 shows instruction cache block A as the most recently used instruction cache block and the block containing `x` as the most recently used data cache block. L2 shows that x was the most recently accessed block.

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 and 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 data cache block. The y' notation means that the block y is dirty (modified). Tracking actual values for variables is not necessary as it doesn't influence the hit/miss behavior. Values can be tracked separately and simplified by eliminating cache models. Next, after all executions of the loop, S_3 shows that L1 data cache now contains the entire array `arr` (block z contains the final two values) and that these values are dirty (but not yet written back to L2).

Next, we analyze the loop on lines 6-7. Note that the state S_4 corresponding to this line 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 `x`. Thus, after the first iteration S_4 shows us that x is the most recently used block, but block y has been loaded. Further, before these data locations were referenced, instruction block B was loaded. 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 cache.

Join function . When the analysis looks at line 8, it sees 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 2 shows the merged states for a "hit" and a "miss" analysis, both of which are sound. 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. Note that in S_5, z is clean but in S_3 it is dirty. Thus, in S_6, z'' denotes that z may be dirty. Also, notice that y does not exist in L1 in S_5, however, in S_3 it does exist and is dirty. Thus, in S_6, y should not exist. 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 $\cancel{y}$) but is kept in the state so higher levels can be modeled in a sound way.

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. If either block is dirty, the resulting block is dirty. For example, even though z is clean in S_5, since it is dirty in S_3, it is dirty in S_6.

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 updating LRU order and evicting A. 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.

Next, block x is accessed resulting in an update to the data cache as shown in S_8. For the hit analysis, two potentially dirty blocks are evicted (z and ~~y~~). Thus, cache is 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.

2.3 Problems with Previous Analysis Techniques

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 2.

Cache hierarchy . Consider the position of instruction block A in state S_1 of Figure 2. 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 . To illustrate write-back, let us revisit the transition from S_7 to S_8 of the hit analysis in Figure 2 where potentially dirty blocks are evicted from L1 cache. As a result of the potential eviction, L2 is updated forcing out block y. Without considering write-back, the analysis would report that y must hit which is not sound. Thus, it is crucial to handle write-back properly, especially in the presence of unified cache levels since data cache write-backs to L2 impact the instruction cache blocks in L2.

Precision . Notice the existence of block x in both potential concrete states of S_7 in Figure 2. 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.

Cache Hierarchy Types . Since we are dealing with multi-level caches, we must consider the different types of multi-level caches. There are three common types of cache hierarchies: inclusive, exclusive, and mainly-inclusive. We now briefly define each of these types of hierarchies and illustrate them in Figure 3.

- **Inclusive:** If Lx and Lx+1 are inclusive, then all blocks in Lx must be contained in Lx+1. Informally, we can think of this as Lx $\subset$ Lx+1. Inclusive caches are beneficial for shared caches since consistency is easy to maintain [16]. Inclusive policies are generally favored for caches shared among multiple cores (such as the L3 in the Intel Core i7) since consistency is more easily maintained.
- **Exclusive:** If Lx and Ly are exclusive, then they contain no common references. Informally, Lx $\cap$ Ly $= \phi$. Exclusive caches generally perform better with small caches since they increase the amount of memory that can be held in cache [32]. It has been shown that as cache size decreases, exclusive policies generally perform best [32]. This is likely due to the fact that exclusive caches effectively increase the amount of data that can be held in a cache.

- **Mainly-inclusive:** Mainly-inclusive caches are similar to inclusive caches. The difference is that blocks are evicted based on the current level only. Thus if a block is in Lx it doesn't mean that it is in Lx+1 (if these two levels are mainly-inclusive).

Thus, since the exclusive policy is good for small caches, if we have many cores with small caches, it may be worthwhile to consider exclusive on-chip caches for each processor. Further, since inclusive caches are good for shared caches and cache consistency, it may be desireable to add larger inclusive caches shared by many cores.

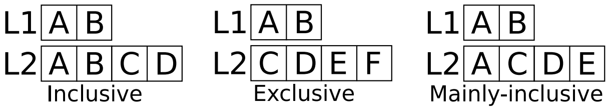


Fig. 3. Example of each type of hierarchy policy.

Previous work [15] developed a technique to analyze multi-level mainly-inclusive caches based on the original single level technique [3]. Since this technique is designed for mainly-inclusive instruction caches, it is unable to soundly analyze a variety of cache hierarchies that exist in practice such as those in the Intel Core i7 [17] and the AMD Phenom II [2] which contain inclusive and exclusive policies respectively.

3 Multi-level cache analysis with live caches

Our analysis is an abstract interpretation [10] for the domain of multi-level caches. Like previous work [3, 14, 15], the goal is to determine which memory references will be hits and which will be misses. Unlike previous work, we analyze both instruction and data caches and use *live caches* to improve precision and handle write-back.

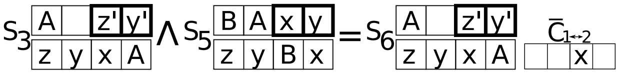


Fig. 4. x receives a better age in Live Cache than L2

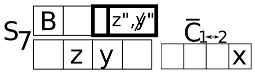


Fig. 5. x hits live cache

3.1 Adding Live Caches to the hierarchy

A live cache captures information accross cache levels. Figures 4 and 5 show states S_6 and S_7 from Figure 2 extended with live caches. The live cache corresponding to L1 data cache and L2 is denoted as $\bar{C}_{1\leftrightarrow 2}$. In Figure 4, block x is 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 takes the second to last position (worst case) in the live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$.

To maintain soundness, a live cache is updated whenever either of its corresponding levels, more specifically sets, is updated. For example, recall that the transition from S_6 to S_7 in Figure 2 involved an update of both L1 instruction cache and L2 cache. Thus, the corresponding live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ must also be updated. Figure 5 illustrates this update. Here block x is moved to the last position in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$.

Size of live caches The size of the live cache is equal to the larger of the two caches (because of the way write-back is handled). For example, in Figure **??**, $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ has four lines which is the size of L2. Since most processors have set associative caches, we need to consider how live caches work with sets.

Number of live caches Recall from our previous example that we used a live cache to capture a block in two different cache levels. Thus, for any block that can fit into either cache, we need to be able to capture this relationship. Therefore, for any two cache sets in Lx and Ly that may possibly share the same memory reference, we need a corresponding set in the live cache $\bar{\mathcal{C}}_{x\leftrightarrow y}$. In total, we have $\max(\text{num\_sets}_x, \text{num\_sets}_y)$ sets in $\bar{\mathcal{C}}_{x\leftrightarrow y}$ where num_sets$_x$ represents the number of sets in Lx cache. This is because in practice we always have that if two sets may contain common references, then one set is a subset of the other.

In Figure 6, L1 has two sets of two blocks each and L2 has four sets of four blocks each. Suppose there are 16 blocks of memory as viewed from these cache levels. The blocks that can fit in each set are specified above the set in the figure. For example, suppose block 2 is in the live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$. Block 2 can exist in the first set of L1 ($\mathcal{S}_{1,1}$) and the first set of L2 ($\mathcal{S}_{2,1}$). This block, 2, must be in the set in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ that corresponds to these two sets, the first set in $\bar{\mathcal{C}}_{1\leftrightarrow 2}$ which is denoted $\bar{\mathcal{S}}_{1,1\leftrightarrow 2,1}$. Then, whenever either $\mathcal{S}_{1,1}$ or $\mathcal{S}_{2,1}$ is updated, $\bar{\mathcal{S}}_{1,1\leftrightarrow 2,1}$ is updated.

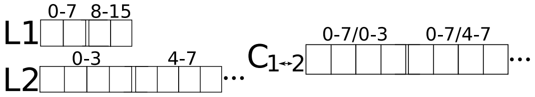


Fig. 6. Live cache size

Since we have a live cache for each pair of cache levels, we have $\binom{\mathcal{N}}{2}$ live caches where $\mathcal{N}$ is the number of cache levels. To handle write-back (and exclusive caches), we have an additional live cache for each cache level. In practice, the number of cache levels in a hierarchies is small (typically 2 or 3 levels). Thus, the space needed for live caches is small.

3.2 Benefits of Live caches

Improved precision . Recall in the example from Figure 2 that when reading x on line 8, the hit analysis could not guarantee a hit even though x is in either concrete state in S_7. Now, consider the read to x again but using the state S_7 augmented with live caches from Figure 4. Since x is in the live cache $\bar{\mathcal{C}}_{1\leftrightarrow 2}$, 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 . As mentioned previously, write-back introduces new behavior that must be modeled. Thus, our analysis models this behavior. Figure 2 shows how write-back is handled without live caches, however, live caches also play a role in giving more accurate knowledge of cache states. Consider the join for S_6 where L2 location for y is determined purely on the references in L2. Since y is dirty in S_3, we know it will be written back to L2 upon eviction. Thus, we can take the location of y in L2 from S_5 which gives an equal or better location than the typical join. The result is that the analysis will report an L2 hit as an upper bound. An L2 hit is a sound approximation, however, the access may also be an L1 hit in the case of S_3. Thus, we use live caches to more accurately classify this block as illustrated in Figure 7. In this example, we have that y exists in the live cache corresponding to L2, $\bar{\mathcal{C}}_{2\leftrightarrow 2}$ (unlike previous live caches for to two levels). The resulting state is more accurate since blocks in L2 must be in L2 and blocks in $\bar{\mathcal{C}}_{2\leftrightarrow 2}$ are in L2 in the worst case. Since the live cache only corresponds to L2, it is only updated whenever L2 is updated. Thus y exists in $\bar{\mathcal{C}}_{2\leftrightarrow 2}$ for the same period as it would in L2 in the hit analysis.

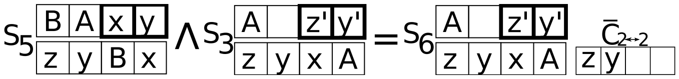


Fig. 7. Live cache join with write-back.

Hierarchy . As discussed previously, it is essential for a sound analysis to model both instruction and data caches due to their impact on each other in unified cache levels. Thus, our analysis simultaneously analyzed both instruction and data caches. Furthermore, the hierarchy as a whole contains more information than the individual levels do in isolation. Therefore, our analysis introduces live caches which capture additional information available in the hierarchy as shown in Figure 4.

Cache Hierarchy Types . We now define the theory of our multi-level cache analysis. As is usual for an abstract interpretation based analysis [3], we first define theconcrete behavior of different types of cache hierarchies (Section 4). Then (Section 5) we define the abstraction of the concrete behavior for multi-level caches and the formalism behind live caches.

4 Concrete Cache Semantics

In this section, we define the concrete semantics for caches. Here, we focus on mainly-inclusive caches, a common multi-level cache policy (Pentium II, III, 4, etc). If Lx and Lx+1 are mainly-inclusive, this means that most blocks in Lx are also in Lx+1. Mainly-inclusive cache levels use their own local policy to determine loads and evicts from cache. When evicting a block, next block according to the policy (LRU) is chosen and the next cache level is notified if the block is dirty. The advantage with mainly-inclusive caches is that they are easy to implement since each layer operates almost independently. An example state of a mainly-inclusive cache is shown on the right side of Figure 3. It is important to note that this is not the way things physically happen in the cache but is the *observable behavior* which suffices for the model. The observable behavior refers to the LRU order and when blocks are evicted from the various cache levels.

4.1 Notation

Figure 8 shows our notation for concrete semantics. It is inspired from Ferdinand and Wilhelm [12], but extends it to multi-level caches.

n_x:	The **number of blocks** that fit into the Lx cache at one time. $n_x = \textit{capacity}_x / \textit{line size}_x$.
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 $\{m_{1,x}, \cdots, m_{\mathcal{M}/\textit{line size}_x,x}\}$, 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**.
$\textit{adr}_x$:	$\textit{adr}_x : M_x \rightarrow \mathbb{N}$ function **mapping memory blocks** (as viewed from Lx cache) **to their start address** and is defined as $\textit{adr}_x(m_{i,x}) = n$ where $n = \lfloor i(\textit{line size}_x) \rfloor$.
$\textit{set}_x$:	$\textit{set}_x : M_x \rightarrow C_x$ **maps blocks to cache sets**. $\textit{set}_x(m_{i,x}) = S_{j,x}$, where $j = \textit{adr}_x(m_{i,x}) \bmod (n_x/A_x) + 1$.
$\textit{blk}_x$:	$\textit{blk}_x : \mathbb{N} \rightarrow M_x$ **maps addresses to blocks**. $\textit{blk}_x(n) = m_{i,x}$ where $i = \lceil n / \textit{line size}_x \rceil$.
δ:	$\delta : C_x \times S_{i,x} \rightarrow \{\textit{true}, \textit{false}\}$ True if the input block is **dirty**.
E_x:	set of caches **exclusive** with the Lx cache (C_x).
$\in'$:	Used for cache levels with different line sizes. e.g.: $m_{i,x} \in' m_{i',x+1}$ means that all of $m_{i,x}$ is in $m_{i',x+1}$ Formally, $m_{i,x} \in' m_{j,y} \Leftrightarrow \textit{adr}_y(m_{j,y}) \leq \textit{adr}_x(m_{i,x}) \leq \textit{adr}_y(m_{j+1,y}), x < y$
rem:	$\textit{rem} : C_x \times M_x \rightarrow C_x$ Used to **remove a block** from a cache set. Suppose $S_{i,x}(l^i_{k,x}) \rightarrow m_{j,x}$. Then, $S'_{i,x} \mapsto \textit{rem}(S_{i,x}, m_{j,x}) \Rightarrow S'_{i,x}(l^i_{k,x}) \rightarrow I$.

Fig. 8. Notation for Concrete Semantics

4.2 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 ($\mathcal{U}\ :\ H \times M_x\ \rightarrow\ H$) for handling individual cache sets is borrowed from previous work [12]. It takes a cache level and a block and returns the updated cache as follows. given in Equation 1. 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 .

Equation 1 Single set update function [12]

$$\mathcal{U}(C_x, m_{j,x}) = \begin{cases} l^i_{1,x} \mapsto m_{j,x}, & \text{if } \exists h, i : \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k-1,x}) | k \in \{2, \cdots, h\}, & S_{i,x}(l^i_{h,x}) = m_{j,x} \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k,x}) | k \in \{h{+}1, \cdots, A_x\} & \\ & \\ l^i_{1,x} \mapsto m_{j,x}, & \text{otherwise} \\ l^i_{k,x} \mapsto S_{i,x}(l^i_{k-1,x}) | k \in \{2, \cdots, A_x\} & \textit{set}_x(m_{j,x}) \rightarrow S_{i,x} \end{cases}$$

We now define our multi-level updates for each policy in terms of this set update function. 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.

Mainly Inclusive A technique based on mainly-inclusive caches can rely heavily on previous work since the replacement policies for each level are independent [31]. Such a technique was defined in previous work for instruction caches [15]. Nevertheless, we re-define the semantics to include multiple levels and write-backs in order to make handling combinations with multiple types (inclusive/exclusive) of caches easier and to incorporate our novel techniques for improved accuracy.

We now define the update function, $\mathcal{R}^{\textit{mainly}}_x : H \times M'_x \rightarrow H$ 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 in Equation 2. 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 4.3). 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.

Inclusive Suppose Lx and Lx+1 are inclusive. This means that Lx+1 that must contain all memory references in Lx. For example, if we are dealing with an inclusive L2 cache, then we can think of L1 $\subseteq$ L2. That is, $x\ \in'$ L1 $\Rightarrow$ $x\ \in'$ L2. An example of two

Equation 2 Mainly inclusive concrete update function

$$\mathcal{R}_x^{mainly}(m_{j,x}) = \begin{cases} \mathcal{U}(C_x, m_{j,x}) & \text{if } \exists i, h : S_{i,x}(l_{h,x}^i) = m_{j,x} \\ \\ C_y \mapsto \mathcal{R}_y^?(m_{j',y}) & y = x+1, \text{ if } \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x}, \ \nexists h : \\ \mathcal{U}(C_x, m_{j,x}) & S_{i,x}(l_{h,x}^i) = m_{j,x} \wedge \neg\delta(S_{i,x}, l_{A_x,x}^i), m_{j,x} \in' m_{j',y} \\ \\ C_y \mapsto \mathcal{W}_{S,y}(S_{k,y}) & y = x+1, \text{ if } \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x}, \ \nexists h : \\ C_y \mapsto \mathcal{R}_y^?(m_{j',y}) & S_{i,x}(l_{h,x}^i) = m_{j,x} \wedge \delta(S_{i,x}, l_{A_x,x}^i), m_{j,x} \in' m_{j',y}, \\ \mathcal{U}(C_x, m_{j,x}) & S_{i,x}(l_{A_x,x}^i) \rightarrow m_{e,x}, \mathit{set}_y(m_{e,x}) \rightarrow S_{k,y} \end{cases}$$

inclusive levels is shown in the left of Figure 3. If a block is evicted from L1 we need not notify L2 unless it is dirty. However, if L2 evicts a block, all blocks in L1 that are subsets of the block must also be evicted from L1. This can be multiple blocks if the line size of L1 is smaller than the line size of L2. This problem is due to the fact that in typical inclusive caches, the L1 cache is not considered when evicting blocks from the L2 cache [26]. Inclusive caches are beneficial for shared caches since maintaining consistency is easy [16].

Given its similarity to the mainly-inclusive policy, our semantics are similar with a few exceptions. Anytime a block is evicted, we must look at lower inclusive levels to see if we must evict anything. Also, upon such an eviction, we must look at the state of the blocks (dirty/clean) to see if higher levels need to be notified.

We now formally define the read function for inclusive cache levels. $\mathcal{R}_x^{incl} : H \times M_x' \rightarrow H$ which takes a cache level and a reference to a memory block and returns the updates cache level. This function is formally defined in Equation 3. 1) The first case is when the block is already in the cache set. In this case, we simply update the LRU order of the current cache level. 2) The second case is when the block is not already in the current cache level and the block being evicted is clean and not in a lower cache. In this case, we update the LRU order of the current cache and issue a read to the next cache level for the same block. 3) The third case is when the block is not already in the current cache level and the block being evicted is clean, but is in a lower cache. In this case, we update the LRU order of the current cache and issue a read to the next cache level for the same block. Then, for all sub-blocks of the evicted block in lower inclusive levels, we remove these blocks. 4) The fourth case is when the block is not already in the current cache level and the block being evicted is dirty and not in a lower cache. In this case, we update the LRU order of the current cache and issue a read to the next cache level for the same block. We also mark the evicted block as dirty in the next cache level. 5) The final case is when the block is not already in the current cache level and the block being evicted is dirty (either in this cache or a lower cache) and exists in a lower cache. In this case, we update the LRU order of the current cache and issue a read to the next cache level for the same block. We also mark the evicted block as dirty in the next cache level. Then, for all sub-blocks of the evicted block in lower inclusive levels, we remove these blocks.

We now prove that the inclusive nature of the cache is preserved with this update function.

Equation 3 Inclusive concrete update function

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

$$\left\{\begin{array}{lr}
\mathcal{U}(C_x, m_{j,x}) & \text{if } \exists i,h : S_{i,x}(l^i_{h,x}) = m_{j,x} \\
 & \\
 & \text{if } \textit{set}_x(m_{j,x}) \rightarrow S_{i,x} m_{j,x} \in' m_{j',x+1} \\
C_{x+1} \mapsto \mathcal{R}^?_{x+1}(m_{j',x+1}) & \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x} \wedge \neg\delta(S_{i,x}, l^i_{A_x,x}) \\
\mathcal{U}(C_x, m_{j,x}) & S_{i,x}(l^i_{A_x,x}) \rightarrow m_{e,x} \\
 & \nexists S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x} \\
 & \\
 & \text{if } \textit{set}_x(m_{j,x}) \rightarrow S_{i,x} \\
C_{x+1} \mapsto \mathcal{R}^?_{x+1}(m_{j',x+1})) & \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x} \wedge \neg\delta(S_{i,x}, l^i_{A_x,x}) \\
\mathcal{U}(C_x, m_{j,x}) & m_{j,x} \in' m_{j',x+1} \\
\forall S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x}, & S_{i,x}(l^i_{A_x,x}) \rightarrow m_{e,x} \\
\textit{rem}(S_{i',x-z}, S_{i',x-z}(l^{i'}_{k,x-z})) & \exists S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x} \\
 & \wedge\neg\delta(S_{i',x-z}, l^{i'}_{k,x-z}) \\
 & \\
 & \text{if } \textit{set}_x(m_{j,x}) \rightarrow S_{i,x} \\
C_{x+1} \mapsto \mathcal{W}_{C,x+1}(m_{e',x+1}) & \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x} \wedge \delta(S_{i,x}, l^i_{A_x,x}) \\
C_{x+1} \mapsto \mathcal{R}^?_{x+1}(m_{j',x+1}) & m_{j,x} \in' m_{j',x+1} \\
\mathcal{U}(C_x, m_{j,x}) & S_{i,x}(l^i_{A_x,x}) \rightarrow m_{e,x}, m_{e,x} \in' m_{e',x+1} \\
 & \nexists S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x} \\
 & \\
C_{x+1} \mapsto \mathcal{W}_{C,x+1}(m_{e',x+1}) & \text{if } \textit{set}_x(m_{j,x}) \rightarrow S_{i,x} \\
C_{x+1} \mapsto \mathcal{R}^?_{x+1}(m_{j',x+1})) & \nexists h : S_{i,x}(l^i_{h,x}) = m_{j,x}, m_{j,x} \in' m_{j',x+1} \\
\mathcal{U}(C_x, m_{j,x}) & S_{i,x}(l^i_{A_x,x}) \rightarrow m_{e,x}, m_{e,x} \in' m_{e',x+1} \\
\forall S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x}, & \exists S_{i',x-z}, l^{i'}_{k,x-z} : S_{i',x-z}(l^{i'}_{k,x-z}) \in' m_{e,x} \\
\textit{rem}(S_{i',x-z}, S_{i',x-z}(l^{i'}_{k,x-z})) & \wedge(\delta(S_{i,x}, l^i_{A_x,x}) \vee \delta(S_{i',x-z}, l^{i'}_{k,x-z}))
\end{array}\right.$$

Theorem: If $C_x \subseteq C_y$, $C_x, C_y \in H$, and C_x, C_y are inclusive, then $C'_x \subseteq C'_y$ where $C'_x, C'_y \in H'$ such that $\mathcal{R}(H, m_{j,1}) \rightarrow H'$.

Proof: We can safely ignore cases which do not result in accesses to C_x since these cases to not change the contents of C_x and C_y. Thus, we show that for each case in the update function for C_x that in the resulting cache hierarchy, C_x and C_y are still inclusive. That is, if $m_{k,x} \in C_x$ then $m_{k,x} \in' C_y$.

- Case 1: Trivial since the cache contents do not change.
- Case 2: WTS $m_{j,x} \in C'_y$.
  We have that $m_{j,x} \in' m_{j',y}$. We also have that $\mathcal{R}(C_y, m_{j',y}) \rightarrow C'_y$. Clearly this read ensures that $m_{j',y} \in C'_y \Rightarrow m_{j,x} \in' C'_y$.
  WTS $m_{e,x} \notin' C'_{x-z} \in E_x$.
  Trivial since this is not possible.
- Case 3: WTS $m_{j,x} \in C'_y$. See Case 2.
  WTS $m_{e,x} \notin' C'_{x-z} \in E_x$.
  Trivial since we remove all such blocks.

– Case 4: Same as Case 2. Write doesn't effect higher caches.
– Case 5: Same as Case 3. Write doesn't effect higher caches.

Therefore we have shown that our concrete semantics are correct in that the inclusive properties of the cache levels are preserved.

Exclusive When we refer to exclusive cache levels, we mean cache levels that must not contain any common memory references [18]. An example of an exclusive cache is shown in the middle of Figure 3. Suppose the set of caches $E_i = \{C_i, C_{i+1}, \cdots, C_{i+n}\}$ are exclusive. Then, $\forall C_j, C_k \in E_i$ s.t. $j \neq k$, we have $C_j \cap C_k = \phi$ where C_j refers to the set of memory blocks held in cache level j. Thus, when we load a block into Lx from Lx+1, we must remove the block from Lx+1. All exclusive levels must have the same line size. For small caches, exclusive caches perform better [31].

We now consider some examples of exclusive cache behavior. First, consider a simple example where each cache level has the same number of sets ($n_i/A_i = n_j/A_j \; \forall C_i, C_j \in E$ s.t. $i \neq j$). In this case, we can think of the exclusive hierarchy as a single "virtual" cache (where the lower cache levels are more recently used in the LRU order). The update function then simply looks at this virtual cache and uses the standard update function ($\mathcal{U}$). An example of this is shown in Figure 9.

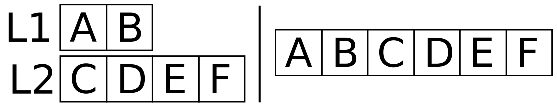


Fig. 9. Exclusive cache view: The left side shows a more realistic representation whereas the right side shows a more virtual representation.

If the different exclusive levels have different numbers of sets, updating becomes more tricky. In the example in Figure 10, we have that the block being read into L1 and the block being evicted from L1 can not exist in the same set in L2. Thus, we may not simply perform a swap or consider these caches as a large virtual cache. Informally, we can think of a read of m_r to exclusive cache Lx as follows. If m_r exists in a higher level of exclusive cache, say L$x + k$, remove m_r from L$x + k$ and replace it with I, the

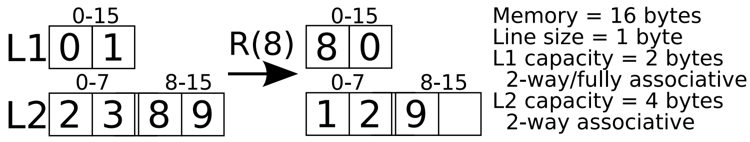


Fig. 10. Exclusive update where swap of evicted and read block is not possible. Addresses above sets represent the range of possible addresses. Addresses inside lines represent the current block.

empty block. Then, load m_r into Lx. Then, issue a read in Lx+1 for the block evicted from Lx. Continue in this fashion, moving each evicted block to the next cache level until L$x + k$ is reached.

We now formally define the read function for exclusive cache levels. $\mathcal{R}_x^{excl} : H \times M_x' \rightarrow H$ which takes a cache level and a reference to a memory block and returns the updated cache level. The formal definition is given in Equation 4. It is important to note that since all cache levels which are exclusive with one other have the same line size, we have that $m_{j,x} = m_{j,x+k}$ where $C_x, C_{x+k} \in E_x$. 1) The first case is when a block is already in the cache set. In this case, we only update the LRU order of the current cache level. 2) The second case is when the block is in a higher level of exclusive cache (Lx+k). In this case, we must first remove the block from the higher level of cache (Lx+k). Then, each level between the current level and the higher level (Lx and Lx+k resp.) must evict the LRU block to the next highest level. Finally, the read block is added to the current level (Lx) and the LRU order is updated. 3) The third case is when the block is not in a higher level of exclusive cache. In this case, we issue a read to the next highest level of cache that is not in the exclusive set (if all levels are exclusive then this is main memory). Then, the block is read into the current level (Lx) and the LRU order is updated. Finally, the evicted blocks of each higher exclusive level are read into the next cache level recursively. 4) The fourth case is when the next cache level is not an exclusive cache and the evicted block is clean. First, we issue a read to the block for next level of cache. Then, the LRU order of the current level is updated. 5) The final case is when the next cache level is not an exclusive cache and the evicted block is dirty. First, we write the evicted block as dirty to the next level of cache. Then,

Equation 4 Exclusive concrete update function

$$\mathcal{R}_x^{excl}(m_{j,x}) = \begin{cases} \mathcal{U}(C_x, m_{j,x}) & \text{if } \exists i, h : S_{i,x}(l_{h,x}^i) = m_{j,x} \\[2ex] \begin{array}{l} \mathit{rem}(S_{n,x+k}, m_{j,x}) \\ \mathcal{U}^{excl}(C_x, m_{j,x}) \end{array} & \begin{array}{r} \text{if } \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x}, \nexists h : S_{i,x}(l_{h,x}^i) = m_{j,x} \\ \exists h, n, k : C_{x+k} \in E_x, S_{n,x+k}(l_{h,x+k}^n) = m_{j,x} \end{array} \\[4ex] \begin{array}{l} \mathcal{U}^{excl}(C_x, m_{j,x}) \\ \mathcal{R}_{x+z}^{?}(m_{j',x+z}) \end{array} & \begin{array}{r} \text{if } C_{x+1} \in E_x,\ \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x}, \\ \nexists h : S_{i,x}(l_{h,x}^i) = m_{j,x},\ \nexists h, n, k : C_{x+k} \in E_x, \\ S_{n,x+k}(l_{h,x+k}^n) {=} m_{j,x}, \\ \exists z : C_{x+z} \notin E_x,\ C_{x+z-1} \in E_x,\ m_{j,z} \in' m_{j',x+z} \end{array} \\[6ex] \begin{array}{l} \mathcal{U}(C_x, m_{j,x}) \\ \mathcal{R}_{x+1}^{?}(m_{j',x+1}) \end{array} & \begin{array}{r} \text{if } C_{x+1} \notin E_x,\ \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x} \\ \nexists h : S_{i,x}(l_{h,x}^i) = m_{j,x}, \\ m_{j,x} \in' m_{j',x+1},\ \neg\delta(S_{i,x}(l_{A_x,x}^i)) \end{array} \\[4ex] \begin{array}{l} \mathcal{U}(C_x, m_{j,x}) \\ \mathcal{W}_{S,x+1}(S_{i,x}(l_{A_x,x}^i)) \\ \mathcal{R}_{x+1}^{?}(m_{j',x+1}) \end{array} & \begin{array}{r} \text{if } C_{x+1} \notin E_x,\ \mathit{set}_x(m_{j,x}) \mapsto S_{i,x} \\ \nexists h : S_{i,x}(l_{h,x}^i) = m_{j,x} \\ m_{j,x} \in' m_{j',x+1},\ \delta(S_{i,x}(l_{A_x,x}^i)) \end{array} \end{cases}$$

we issue a read to the block for next level of cache. Finally, the LRU order of the current level is updated.

This function relies on another function, $\mathcal{U}^{\mathit{excl}}$ for recursively updating the LRU order of the exclusive cache levels. Formally, $\mathcal{U}^{\mathit{excl}} : H \times M' \rightarrow H$ and is defined in Equation 5.

Equation 5 Exclusive recursive update function

$$\mathcal{U}^{\mathit{excl}}(C_x, m_{j,x}) = \begin{cases} \begin{array}{l} \mathcal{U}(C_x, m_{j,x}) \\ \mathcal{U}^{\mathit{excl}}(C_{x+1}, m_{e,x}) \end{array} & \begin{array}{r} \text{if } C_{x+1} \in E_x \\ \mathit{set}_x(m_{j,x}) \rightarrow S_{i,x} \\ S_{i,x}(l^i_{A_x,x}) \rightarrow m_{e,x} \end{array} \\ & \\ \mathcal{U}(C_x, m_{j,x}) & \text{otherwise } (C_{x+1} \notin E_x) \\ & \end{cases}$$

We now prove that the exclusive nature of the cache levels is preserved with this update function.

Theorem x: Suppose H contains a set of exclusive caches, E, which is correct. Also, suppose $\nexists z$ s.t. $m_{j,x} \in' C_z \in E$. Then $\mathcal{U}^{\mathit{excl}}(C_x, m_{j,x}) \rightarrow H'$ where H' is correct.

Proof: Suppose we have H s.t. $\forall x, y : C_x, C_y \in E \subseteq H, C_x \cap C_y = \phi$. Then, $\mathcal{U}^{\mathit{excl}}(C_x, m_{j,x}) \rightarrow H'$

- Case 1: Since $m_{j,x} \notin C_x$, we evict a block, $m_{e,x}$ to C_{x+1}. Since H was correct and $m_{e,x} \in' C_x$, After $\mathcal{U}(C_x, m_{j,x})$, $m_{e,x} \notin C'_x$. Thus, $\forall z$ s.t. $C_z \in E, m_{e,x} \notin C'_z$
- Case 2: We can safely add $m_{j,x}$ since it is not in any other exclusive levels. This means we will evict a block, $m_{e,x}$ which is acceptable since it will not effect any other levels of exclusive cache and removing a block can not break exclusiveness.

Theorem y: Suppose E is a set of exclusive caches. So, $\forall C_x, C_y \in E \subseteq H$, we have that $C_x \cap C_y = \phi$. Then, $\mathcal{R}(H, m_{j,x}) \rightarrow H'$ where $\forall C_y \in E, x \leq y$ and $\forall C_x, C_y \in E$, we have that $C'_x \cap C'_y = \phi$, where $C'_x, C'_y \in H'$

Proof: Suppose we have a read, $\mathcal{R}(H, m_{j,x}) \rightarrow H'$. We are to show (WTS) that $C'_x \cap C'_y = \phi : C'_x, C'_y \in E \subseteq H'$. We show that for each case (each case in the update function) that in the resulting cache hierarchy, the exclusive sets are still exclusive.

- Case 1: Trivial since the cache contents do not change.
- Case 2: Since H was exclusive and $m_{j,x} \in' C_{x+k}$, then $\mathit{rem}(S_{n,x+k}, m_{j,x}) \Rightarrow \forall C_y \in E, m_{j,x} \notin' C_y$. Thus, by Therem x, H' is correct. Note that we can safely assume that $m_{j,x}$ is not in a lower level of exclusive cache since this function would never be called if this were the case.
- Case 3: We have that $\forall C_y \in E, m_{j,x} \notin' C_y$. Since $C_{x+z} \notin E$, we can ignore the read to it. Thus, by Therem x, H' is correct.
- Case 4: WTS $m_{j,x} \notin C'_y \in E, \forall y \neq x$.
  Suppose $m_{j,x} \in C'_{x-z} \in E$. Then, $\mathcal{R}^{excl}_{x-z}$ would have taken Case 1. Thus, this is a contradiction to being in this Case. Therefore, $m_{j,x} \notin C_{x-z} \in E$ Since the next level is not exclusive with the current level, we can safely evict the LRU block.

– Case 5: Same as Case 4 except write to non-exclusive cache.

4.3 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. $\mathcal{W}_{S,x}$ is formally defined in Equation 6.

Equation 6 Cache set write function

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

4.4 Reads/Writes

Now, we define reads and writes for entire cache hierarchies. In a cache hierarchy, cache levels may have different policies (inclusive, exclusive, or mainly-inclusive). Thus, we define our semantics to handle cases in which a variety of policies may be used for different cache levels.

First, we define a read to an address as $\mathcal{R} : \mathbb{N} \times \mathcal{P}(H) \rightarrow \mathcal{P}(H)$ which takes the address being read and a cache hierarchy in Equation 7. This function first determines the block containing the address in the L1 cache. Then, it finds the set which holds the block in the L1 cache. Finally, a read is issued to this set.

Equation 7 Cache read function

$$\mathcal{R}(n, H) = \begin{cases} \textit{blk}_1(n) \rightarrow m_{i,1} \\ \textit{set}_1(m_{i,1}) \rightarrow S_{j,1} \\ \mathcal{R}_1(S_{j,1}, m_{i,1}) \end{cases}$$

Next, we define a write to an address as $\mathcal{W} : \mathbb{N} \times \mathcal{P}(H) \rightarrow \mathcal{P}(H)$ which takes the address being written to and a cache hierarchy in Equation 8. Since writes are accesses, this function first updates the LRU order via the read function for L1 cache. Then, the function determines the block containing the address in the L1 cache. Next, it finds the set which holds the block in the L1 cache. Finally, the first block in the set is marked as dirty.

5 Abstract Cache Semantics

This section discusses our abstract semantics including notations, join and update functions, and hit/miss analysis.

Equation 8 Cache write function

$$\mathcal{W}(n,H) = \begin{cases} \mathcal{R}(n,H) \\ \mathit{blk}_1(n) \rightarrow m_{i,1} \\ \mathit{set}_1(m_{i,1}) \rightarrow S_{j,1} \\ \mathcal{W}_{S,1}(S_{j,1}) \end{cases}$$

5.1 Notation

The notations used to define and update abstract states is similar to the notations used for concrete states. For example, we use $\mathcal{C}_x$ to refer to the abstract Lx cache whereas C_x was used to refer to the concrete Lx cache. We use similar notation for other syntax.

Single block to Set . Since run-time path is unknown, lines may hold more than one value (illustrated in Figure 2). Therefore, the 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 $\mathcal{S}_{i,x} \in Cx$ and $\mathcal{S}_{k,y} \in Cy$. Then, to refer to this set in the live cache $\bar{\mathcal{C}}_{x\leftrightarrow y}$, 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 2), for precision, we like to know if a block *must* be dirty ($'$ in Figure 2). First, $\delta : M \times \mathcal{S} \rightarrow \{true, false\}$ takes a block and a cache line and returns true if any 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 . We define the abstract hierarchy state $\mathcal{H}$ by extending the concrete hierarchy state, H, to include $\mathcal{V} = \{\bar{\mathcal{C}}_{1\leftrightarrow 1}, \bar{\mathcal{C}}_{1\leftrightarrow 2}, \cdots, \bar{\mathcal{C}}_{\mathcal{N}\leftrightarrow\mathcal{N}}\}$, the set containing all live caches. Formally, $\mathcal{H} = \{\mathcal{C}_1, \cdots, \mathcal{C}_{\mathcal{N}}, \bar{\mathcal{C}}_{1\leftrightarrow 1}, \bar{\mathcal{C}}_{1\leftrightarrow 2}, \cdots, \bar{\mathcal{C}}_{\mathcal{N}\leftrightarrow\mathcal{N}}\}$.

5.2 Hit/Miss Analysis

As shown in Section 2, our approach consists of two parts, a hit (must) and miss (may) analysis [3,15]. 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 definitely hit in Lx,
- the access will definitely miss in Lx, or
- we do not know 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

- the access in worst case will hit Lx (when $m_{j,w} \in' \bar{\mathcal{C}}_{x\leftrightarrow x}$) or
- the access will either hit Lx or Ly, worst case it will hit Ly (when $m_{j,w} \in' \bar{\mathcal{C}}_{x\leftrightarrow y}$).

This new information results in a tighter upper bound since we classify additional accesses as hits. Formal teatment of miss analysis is omitted here, but included in our technical report [27].

Like previous work [12, 15] 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 2 (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).

5.3 Join function

The "join" function is used to combine states from two separate program paths. Since we are designing a sound analysis, this must be a worst case combination. This is illustrated in Figure 2. Compared to join functions used by previous work [15], this definition handles live caches and data caches soundly. As a result of these new join functions, we re-gain soundness for data caches as well as inclusive and exclusive caches. We also re-define the join function to include handling live caches properly.

Mainly-inclusive The join function for our hit analysis is defined in Equation 9. Here, $\mathcal{S}_{i,x} \in \mathcal{C}_x \in \mathcal{H}$ and $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k} \in \bar{\mathcal{C}}_{x\leftrightarrow y} \in \mathcal{H}$. Since $\bigwedge(H', H'') = \bigwedge(H'', H')$ 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.

- First case is for the actual abstract caches ($\mathcal{C}_x$) and is when two references exist on the same level. In this case (like previous work [3]), we take the later position ($\max(a, b)$) and update the dirty (δ) and may dirty ($\bar{\delta}$) states of the block.
- Second case deals with write-back when a dirty reference exists in one case (H') but not the other (H''). We must keep 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 (H). Thus, we use an empty block (I) that is marked maybe dirty $\bar{\delta}(I, l^x_{a,i})$.
- 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.
- Fourth case handles states where the block is in different levels 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.
- 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.

Equation 9 Join function for mainly-inclusive policy (hit analysis)

$$\bigwedge_{\mathit{hit}}^{\mathit{mainly}} (\mathcal{H}', \mathcal{H}'') = \mathcal{H} \Rightarrow m_{j,w} \in' \mathcal{S}_{i,x}(l^i_{t,x})|$$

$$\left\{ \begin{array}{ll} t = \max(a,b) & \text{if } \exists a,b : m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x}), \\ \delta(m_{j,w}, l^i_{t,x}) = \delta'(m_{j,w}, l^i_{a,x}) \wedge \delta''(m_{j,w}, l^i_{b,x}) & \quad m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x}) \\ \bar{\delta}(m_{j,w}, l^i_{t,x}) = \bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}) & \end{array} \right. \tag{1}$$

$$, I \in' \mathcal{S}_{i,x}(l^i_{a,x})|$$

$$\left\{ \begin{array}{ll} \bar{\delta}(I, l^i_{a,x}) & \text{if } m_{j,w} \in' \mathcal{S}_{i,x}(l^i_{a,x}), \bar{\delta}'(m_{j,w}, l^i_{a,x}), \\ & \quad \nexists b, y, k : m_{j,w} \in' \mathcal{S}_{k,y}(l^k_{b,y}) \end{array} \right. \tag{2}$$

$$, m_{j,w} \in' \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{t,x,y})|x \leq y,$$

$$\left\{ \begin{array}{lrl} t = A_{x,y} - \min(A_x - a, A_y - b) & x < y \ \text{ if } \exists a,b : \neg\delta'(m_{j,w}, l^i_{a,x}), & \\ \neg\delta(m_{j,w}, \overline{l}^{i,k}_{t,x,y}), & m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x}), & (3) \\ \bar{\delta}(m_{j,w}, \overline{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}) & m_{j,w} \in' \mathcal{S}''_{k,y}(l^k_{b,y}) & \\ & & \\ t = A_{y,y} - (A_y - b) & x = y \ \text{ if } \exists a,b,w,p : \delta'(m_{j,w}, l^p_{a,w}) & \\ \delta(m_{j,w}, \overline{l}^{k,k}_{t,y,y}) = \delta''(m_{j,w}, l^k_{b,y}), & m_{j,w} \in' \mathcal{S}'_{p,w}(l^p_{a,w}), & (4) \\ \bar{\delta}(m_{j,w}, \overline{l}^{k,k}_{t,y,y}) & m_{j,w} \in' \mathcal{S}''_{k,y}(l^k_{b,y}) & \\ & & \\ t = \max(a,b) & \text{if } \exists a,b : & \\ \delta(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \delta'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \wedge \delta''(m_{j,w}, \overline{l}^{i,k}_{b,x,y}) & m_{j,w} \in' \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{a,x,y}), & (5) \\ \bar{\delta}(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, \overline{l}^{i,k}_{b,x,y}) & m_{j,w} \in' \bar{\mathcal{S}}''_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{b,x,y}) & \\ & & \\ t = A_{x,y} - \min(A_{x,y} - a, A_x - b) & \text{if } x < y \ \ \exists a,b : & \\ \neg\delta(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) & m_{j,w} \in' \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{a,x,y}), & (6) \\ \bar{\delta}(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}) & m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x}), \neg\delta''(m_{j,w}, l^i_{b,x}) & \\ & & \\ t = a & \text{if } x < y \ \ \exists a,b : & \\ \neg\delta(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) & m_{j,w} \in' \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{a,x,y}), & (7) \\ \bar{\delta}(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}) & m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x}), \delta''(m_{j,w}, l^i_{b,x}) & \\ & & \\ t = A_{x,y} - \min(A_{x,y} - a, A_y - b) & \text{if } \exists a,b : & \\ \delta(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \delta'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \wedge \delta''(m_{j,w}, l^k_{b,y}) & m_{j,w} \in' \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\overline{l}^{i,k}_{a,x,y}), & (8) \\ \bar{\delta}(m_{j,w}, \overline{l}^{i,k}_{t,x,y}) = \bar{\delta}'(m_{j,w}, \overline{l}^{i,k}_{a,x,y}) \vee \bar{\delta}''(m_{j,w}, l^k_{b,y}) & m_{j,w} \in' \mathcal{S}''_{i,y}(l^k_{b,y}) & \end{array} \right.$$

- 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.
- 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, 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.

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

Soundness . We now prove the soundness of the join function. To do so (for our hit analysis), we must show that any block in either input state is safely approximated in the output state. This means that for any sequence of reads and writes that the block in the output state is evicted no later than the block in either input state.

Theorem 1: Suppose $\mathcal{H}$’ and $\mathcal{H}$” are sound approximations of H' and H'' respectively (their concrete states). Then $\bigwedge(\mathcal{H}', \mathcal{H}'') = \mathcal{H}$ where $\mathcal{H}$ is a sound approximation of H.
Proof: We show for each case, the placement that the join function gives the block in the resulting state $\mathcal{H}$. We then show how this placement is a safe approximation of either case ($\mathcal{H}'$ and $\mathcal{H}''$).

1. Suppose $m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x})$ and $m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x})$
   Without loss of generality (WLOG), assume $a \geq b$. Then resulting block is in $l^i_{a,x}$
   (a) Suppose $\delta'(m_{j,w}, l^i_{a,x}) \wedge \delta''(m_{j,w}, l^i_{b,x})$. Then, $\delta(m_{j,w}, l^i_{a,x})$. Since $a \geq b$, then $A_x - a \leq A_x - b$. Thus, in either case, $m_{j,w}$ is in a later or equal position of the same level. Thus, $m_{j,w}$ will leave Lx at least as early as either case. Both cases have $m_{j,w}$ as dirty, so resulting block is same as previous.
   (b) Suppose $\neg(\delta'(m_{j,w}, l^i_{a,x}) \vee \delta''(m_{j,w}, l^i_{b,x}))$. Then, $\neg\delta(m_{j,w}, l^i_{a,x})$. Since neither case has $m_{j,w}$ as dirty, the resulting block is the same as previous. Like the previous case, in the resulting state, $m_{j,w}$ has either the same position or is closer to being evicted than the two states. Thus, it will leave Lx no later than either case.
   (c) Suppose exactly one version is dirty. WLOG $\delta'(m_{j,w}, l^i_{a,x}) \wedge \neg\delta''(m_{j,w}, l^i_{a,x})$. Then, $\neg\delta(m_{j,w}, l^i_{a,x}) \wedge \bar{\delta}(m_{j,w}, l^i_{a,x})$. Like previous cases, $m_{j,w}$ has either the same position or is closer to being evicted than the two states. Thus, it will leave Lx no later than either case. Upon eviction, we update the LRU order of Lx+1 with an empty block. Thus, if $\delta(m_{j,w}, l^i_{a,x})$, we still update the LRU order as before. If $\neg\delta(m_{j,w}, l^i_{a,x})$, since we have updated the order with the empty block, our assumption is still safe. Thus, the higher level is handled safely.
2. Suppose $m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x})$ and $\exists k, y, b : m_{j,w} \in' \mathcal{S}''_{k,y}(l^k_{b,y})$. WLOG, assume $x < y$.
   (a) $\neg(\delta'(m_{j,w}, l^i_{a,x}) \wedge \delta''(m_{j,w}, l^k_{b,y})) \wedge \neg(\bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^k_{b,y}))$. Then $m_{j,w}$ takes the position $\min(A_x - a, A_y - b)$ places away from being evicted in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ Since $A_x - a \geq \min(A_x - a, A_y - b) \leq A_y - b$, in $\mathcal{H}$, $m_{j,w}$ has the worst case proximity to eviction. Thus after $\min(A_x - a, A_y - b)$ accesses to either $S_{i,x}$ or $S_{k,y}$, $m_{j,w}$ is evicted from cache altogether. Thus, it will leave $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ no later than it will leave Lx or Ly in either case. While it is in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$, it is classified as either a hit to Lx or Ly which is the case since

it is still in either Lx or Ly. Since neither block may be dirty, higher levels are handled safely.

(b) $\bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^k_{b,y})$. This is the same as the previous case except that $m_{j,w}$ may be dirty. Suppose $\bar{\delta}'(m_{j,w}, l^i_{a,x})$. Thus, by the second rule of the join, $I \in \mathcal{S}_{i,x}(l^i_{a,x})$ and $\bar{\delta}(m_{j,w}, l^i_{a,x})$. Thus, when it is evicted, we update the order of $\mathcal{S}_{set_{x+1}(m_{j,w}),x+1}$ with the empty block. If $\delta(m_{j,w})$, then we properly update $S_{set_{x+1}(m_{j,w}),x+1}$. If $\neg\delta(m_{j,w})$, then we have not assumed that $m_{j,w}$ will be updated in $S_{set_{x+1}(m_{j,w}),x+1}$, thus we are safe. The case for $\bar{\delta}''(m_{j,w}, l^k_{b,y})$ is similar. (This case includes $\delta''(m_{j,w}, l^k_{b,y})$ since $\delta''(m_{j,w}, l^k_{b,y}) \Rightarrow \bar{\delta}''(m_{j,w}, l^k_{b,y})$)

(c) $\delta'(m_{j,w}, l^i_{a,x})$. In this case, the block in the lower level is dirty . Thus, we take the position $t = A_{y,y} - (A_y - b)$ in $\bar{\mathcal{S}}_{y,k\leftrightarrow y,k}$. If $\mathcal{H}'$ is the case, then this is a safe upper bound since $m_{j,w} \in \mathcal{S}_{i,x}(l^i_{a,x})$ and $x < y$. Once $m_{j,w}$ is evicted from $\mathcal{S}_{i,x}$, it is written to $\mathcal{S}_{set_{x+1}(m_{j,w}),x+1}$. If $x+1 < y$, then $m_{j,w} \in \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ is still a safe upper bound since Lx+1 hit is better than Ly hit. If $x+1 = y$, then $m_{j,w} \in \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ is still a safe upper bound since $m_{j,w} \in \mathcal{S}_{i,y}(l^k_{1,y})$

3. Suppose $m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x})$ and $\nexists k, b, y : m_{j,w} \notin' \mathcal{S}''_{k,y}(l^k_{b,y})$

(a) $\neg\bar{\delta}'(m_{j,w}, l^i_{a,x})$. The block does not exist in the joined state and is thus not reported as a hit which is sound.

(b) $\delta'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}'(m_{j,w}, l^i_{a,x})$. Therefore, by the second rule of the join, $I \in \mathcal{S}_{i,x}(l^i_{a,x})$ and $\bar{\delta}(m_{j,w}, l^i_{a,x})$. Thus, when evicted, we update the order of $\mathcal{S}_{set_{x+1}(m_{j,w}),x+1}$ with I If $\delta(m_{j,w})$, then we properly update $S_{set_{x+1}(m_{j,w}),x+1}$. If $\neg\delta(m_{j,w})$, then we have not assumed that $m_{j,w}$ will be updated in $S_{set_{x+1}(m_{j,w}),x+1}$, thus we are safe.

4. Suppose $m_{j,w} \in' \bar{\mathcal{S}}'_{x,i\leftrightarrow y,k}(\bar{l}^i_{a,x})$. WLOG, assume $x < y$.

(a) $m_{j,w} \in' \bar{\mathcal{S}}''_{x,i\leftrightarrow y,i}(\bar{l}^k_{b,y})$. In this case, $m_{j,w}$ takes the position $\min(A_{x,y} - a, A_{x,y} - b)$ places from eviction in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$. Since $A_{x,y} - a \geq \min(A_{x,y} - a, A_{x,y} - b) \leq A_{x,y} - b$, in $\mathcal{H}$, $m_{j,w}$ has the worst case proximity to eviction. Thus after $\min(A_{x,y} - a, A_{x,y} - b)$ accesses to either $S_{i,x}$ or $S_{k,y}$, $m_{j,w}$ is evicted from cache altogether. Thus, it will leave $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ no later than it will leave Lx or Ly in either case. While it is in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$, it is classified as either a hit to Lx or Ly which is the case since it is still in either Lx or Ly.

(b) $m_{j,w} \in' \mathcal{S}''_{i,x}(l^x_{b,i})$ and $\delta''(m_{j,w}, l^x_{b,i})$. Then, the position of $m_{j,w}$ in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,i}$ remains unchanged. In the case of $\mathcal{H}'$ this is safe since $\mathcal{H}'$ is safe. In the case of $\mathcal{H}''$, this is safe since $x < y$ (see argument in Case 2c)

(c) $m_{j,w} \in' \mathcal{S}''_{i,x}(l^x_{b,i})$ and $\neg\delta''(m_{j,w}, l^x_{b,i})$. In this case, $m_{j,w}$ takes the position $\min(A_{x,y}-a, A_x-b)$ places away from being evicted in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$. Since $A_{x,y}-a \geq \min(A_{x,y}-a, A_x-b) \leq A_x-b$, in the resulting state, $m_{j,w}$ has the worst case proximity to eviction. Thus after $\min(A_{x,y} - a, A_x - b)$ accesses to either $S_{i,x}$ or $S_{k,y}$, $m_{j,w}$ is evicted from cache altogether. Thus, $m_{j,w}$ will leave $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$ no later than it will leave Lx or Ly in either case. While $m_{j,w}$ is in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$, it is classified as either a hit to Lx or Ly which is the case since it is still in either Lx or Ly. When $\bar{\delta}''(m_{j,w}, l^x_{b,i})$, this is handled by Case 3b.

(d) $m_{j,w} \in' \mathcal{S}''_{k,y}(l^k_{b,y})$. In this case, $m_{j,w}$ takes the position $\min(A_{x,y} - a, A_y - b)$ places away from being evicted in $\bar{\mathcal{S}}_{x,i\leftrightarrow y,k}$. Since $A_{x,y}-a \geq \min(A_{x,y}-a, A_y-$

$b) \leq A_y$–b, in the resulting state, $m_{j,w}$ has the worst case proximity to eviction. Thus after $\min(A_{x,y} - a, A_y - b)$ accesses to either $S_{i,x}$ or $S_{k,y}$, $m_{j,w}$ is evicted from cache altogether. Thus, it will leave $\bar{\mathcal{S}}_{x,i \leftrightarrow y,k}$ no later than it will leave Lx or Ly in either case. While it is in $\bar{\mathcal{S}}_{x,i \leftrightarrow y,k}$, it is classified as either a hit to Lx or Ly which is the case since it is still in either Lx or Ly. When $\bar{\delta}''(m_{j,w}, l^y_{b,i})$, this is handled by Case 3b.

5. Otherwise, in all other cases, the $m_{j,w}$ does not exist in $\mathcal{H}$. Thus, it is reported as unsure which is sound for any case.

Formally, the join function for our miss analysis is defined in Equation 10

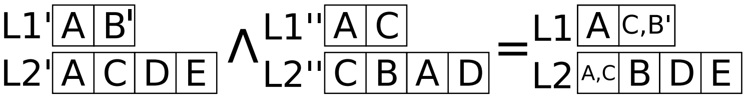


Fig. 11. Sample mainly-inclusive cache join for miss analysis.

Equation 10 Join function for mainly-inclusive miss analysis

$$\bigwedge_{miss}^{mainly}(\mathcal{H}', \mathcal{H}'') = \mathcal{H} \Rightarrow m_{j,w} \in' \mathcal{S}_{i,x}(l^i_{t,x})|$$

$$\begin{cases} t = \min(a, b) & \text{if } \exists a, b : m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x}), \\ \delta(m_{j,w}, l^i_{t,x}) = \delta'(m_{j,w}, l^i_{a,x}) \vee \delta''(m_{j,w}, l^i_{b,x}) & m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x}) \\ & \\ t = a, & \text{if } \exists a : m_{j,w} \in' \mathcal{S}'_{i,x}(l^i_{a,x}), \\ \delta(m_{j,w}, l^i_{t,x}) = \delta'(m_{j,w}, l^i_{a,x}) & \nexists b : m_{j,w} \in' \mathcal{S}''_{i,x}(l^i_{b,x}) \end{cases}$$

Inclusive Due to the similarity between inclusive and mainly-inclusive cache hierarchies, we use the same join functions for inclusive caches as we do for mainly-inclusive caches. With inclusive caches, we know know more about high levels of cache based on what we know about lower levels of cache. With inclusive cache levels a block in L1 implies that it is in L2. This means that the first case (Figure **??**) in which live caches improved performance no longer exists. However, the second case (Figure **??**) still applies and thus we still use live caches for inclusive caches.

Formally, the join function for our miss analysis is defined in Equation 11. Again, due to the similarity between inclusive and mainly-inclusive cache hierarchies, the inclusive join function is the same as the mainly-inclusive join function.

Equation 11 Join function for inclusive policy (miss analysis)

$$\overset{incl}{\underset{miss}{\bigwedge}} = \overset{mainly}{\underset{miss}{\bigwedge}}$$

Exclusive Exclusive caches present some very different behavior and must look at the hierarchy in order to determine higher level cache behavior. Like previous join functions, if a block exists in the same level in both cases, we simply take the LRU position for this level. However, with exclusive caches, it is much more frequent for a block to exist in two different levels in the two cases. When a block is evicted from an exclusive cache, it is put in the first location in the next exclusive cache level. Due to this behavior, when joining a block in two levels, we could just take the position in the higher level cache. This would be an upper bound on access time, however, to differentiate between cases where the block is definitely in the higher level and the case where we are not sure which level it is in, we make use of our live caches again. Figure 12 illustrates an example of the join function on a two level exclusive cache. We have that B which exists in L1 in both cases is in L1 in the resulting case. Then, A which is in L1 in the first case and L2 in the second case is in the live cache corresponding to L1 and L2. Since we model the behavior in this way, we can rely heavily on the join function for mainly-inclusive caches.

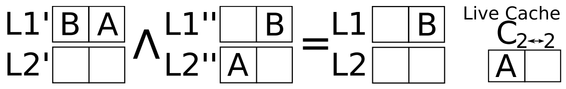


Fig. 12. Exclusive join

The join function for exclusive caches is formally defined in Equation 12. This new case is for when blocks exist in different levels. In the mainly inclusive case, we had

Equation 12 Join function for exclusive policy

$$\overset{excl}{\underset{hit}{\bigwedge}}(\mathcal{H}', \mathcal{H}'') = \overset{mainly}{\underset{hit}{\bigwedge}} \uplus$$

$$m_{j,w} \in' \bar{\mathcal{S}}_{x,i\leftrightarrow y,k}(\bar{l}^{i,k}_{t,x,y})|x \leq y,$$

$$\begin{cases} t = A_{x,y} - (A_x - b) & x = y, i = k, \\ \delta(m_{j,w}, \bar{l}^{i,i}_{t,x,x}) = \delta'(m_{j,w}, l^i_{a,x}) \wedge \delta''(m_{j,w}, l^i_{b,x}) & \text{if } \exists a, b, n, z : z < x,\ \mathcal{C}_x, \mathcal{C}_z \in E, \\ \bar{\delta}(m_{j,w}, \bar{l}^{i,i}_{t,x,x}) = \bar{\delta}'(m_{j,w}, l^i_{a,x}) \vee \bar{\delta}''(m_{j,w}, l^i_{b,x}) & m_{j,w} \in' \mathcal{S}'_{n,z}(l^n_{a,z}),\ m_{j,w} \in' \mathcal{S}_{i,x}(l^i_{b,x}) \end{cases}$$

a similar rule for when the block in the lower level was dirty. For exclusive caches, as illustrated in Figure 12, we have the same behavior even when the block is not dirty. Thus, we can relax the precondition to not look at the dirty/clean state of the block and only require that both levels are exclusive cache levels. In this case, the block is put in the live cache corresponding to the higher level. We then take the position of the block in the higher cache level.

The join function for our miss analysis is the same as the join function for mainly-inclusive caches. The join for our miss analysis is illustrated in Figure 13 Formally, the join function for our miss analysis is defined in Equation 13

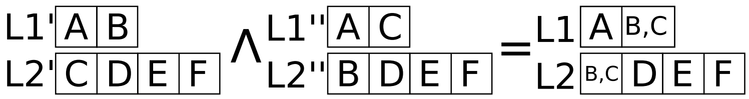


Fig. 13. Example of exclusive cache join for miss analysis.

Equation 13 Join function for exclusive policy (miss analysis)

$$\bigwedge_{miss}^{excl} = \bigwedge_{miss}^{mainly}$$

5.4 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.

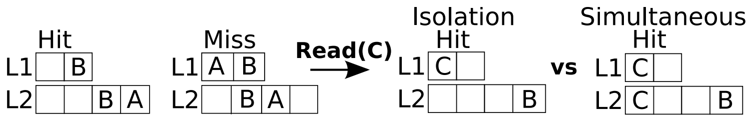


Fig. 14. Example simultaneous analysis.

Like previous work [15], we need to define a simultaneous hit and miss analysis so the hit analysis may consider the miss analysis in some cases. Figure 14 shows how performing each analysis (hit and miss) simultaneously can improve accuracy. When performing the hit analysis (first state), suppose we have a read to memory block C but do not know for sure that it is in the L1 cache. Since we desire a sound analysis, we must make safe assumptions. For this analysis, we must assume that the reference was not in the L1 cache. Thus, we must update the LRU order of the L2 cache. Suppose the reference was in the L1 cache. If this is the case, then C would not be updated in the LRU order of the L2 cache. Thus, we can not issue an update for L2 with C because we can not assume that it was a miss since this would not be sound. Thus, our safe estimate is that we have an L1 miss, update the LRU order of L2 cache but not with C.This is shown in the third state in Figure 14. In many cases, this is an overly safe estimation (no blocks would get loaded into higher cache levels). If we can determine that C must miss L1, then we can update L2 with C and improve accuracy. Recall that we have another analysis, our miss analysis. The current state for this miss analysis is shown in the second state of Figure 14. This miss analysis tells us that C will definitely miss L1. Thus, our hit analysis can update L2 with C. Therefore, we make use of this miss analysis to gain accuracy. The fourth state in Figure 14 shows the resulting state with a simultaneous analysis. Clearly this is an improvement over the isolated analysis.

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 int * a = (int *) malloc(sizeof(int));
2 *a = 5;
```

In this trivial example, we allocate some memory and set the pointer a to the address of this new memory. 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 can also solve this problem. Up to now, we had live cache sets that corresponded to two sets in different caches. We now add a live cache set ($\bar{\mathcal{S}}_{x,\forall\leftrightarrow x,\forall}$) for each live cache corresponding to a single level ($\bar{\mathcal{C}}_{x\leftrightarrow x}$). This new set corresponds to all live cache sets in this cache. Of course this means that references in this new set are likely to have a short lifetime since sets are typically small and we update the set anytime the cache level is updated. However, these sets catch cases like those illustrated previously which occur frequently in practice.

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 catch this important case. Furthermore, 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.

Even though blocks will be evicted from live cache quickly, since we have an additional set for each cache level, these blocks will remain in higher level caches long after they are evicted from the first level cache. This will result in previously lost memory references being classified as worst case higher level cache hits rather memory accesses.

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 in Equation 14. 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.

Equation 14 When live cache updates are called.

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

We define how live cache updates behave in Equation 15. This definition relies on update function, $\mathcal{U}$, similar to the update function for concrete cache sets presented in Equation 1. 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.

Equation 15 Update behavior for live caches

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

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

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

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

In the first case, since $x < y$ we just update the LRU order with the new set of blocks B. This is because we never have blocks that are definitely dirty in a live cache connected to two separate levels. In the second case, we take all blocks being evicted from the live cache and write them 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.

5.5 Termination of the Analysis

Here, we give an informal description of minor changes needed to ensure that the analysis terminates. Since our domain is finite, to prove that the analysis terminates, we must show that the join function is monotonic [15]. Initially, after viewing Figure 4, it may appear that the join function is not monotonic. However, only minor changes are necessary to ensure this. A formal proof of termination is trivial by enumerating all cases (since the domain is finite). An enumeration of a simple case without single level live-caches and only 2-way associative cache levels is shown in Figure 15. Given the length of such an enumeration for a realistic cache, we omit it and continue with the information description.

Recall that the live caches give some knowledge about a reference in the worst case. Thus, if a reference is in L1 for sure, we can still put it in any live cache corresponding to L1. Thus, when you consider Figure 4, in both of the cases being joined, x was actually in the live cache corresponding to L1 and L2 already. In S_5 it was in the second last position and in S_3 it was also in the second to last position. Thus, we are actually taking the worst case between these two and thus intuitively we can see that the join function is monotonic.

This actually makes our rules for join slightly more simple and only slightly complicates the semantics for reads. After the read each block needs to be copied to corresponding live caches in same proximity to eviction. For example, consider Figure 16. In this figure we first have some example state where A exists in both L1 and L2. Thus, in the middle state, we can add A to the live cache for L1 and L2 in the last and second to last position. Then, in the third state, we show how we can simply take the better of the two cases for simplicity. This is not required, but the later block containing A will have no effect on the analysis.

Summary . We have now defined our concrete and abstract cache semantics as well as the semantics for live caches. We have shown several ways in which live caches improve our analysis and shown that our analysis is sound in the presence of live caches.

6 Evaluation

We evaluate our technique in several ways. First, we compare our technique with existing techniques to determine which techniques consider various cases and architectures.

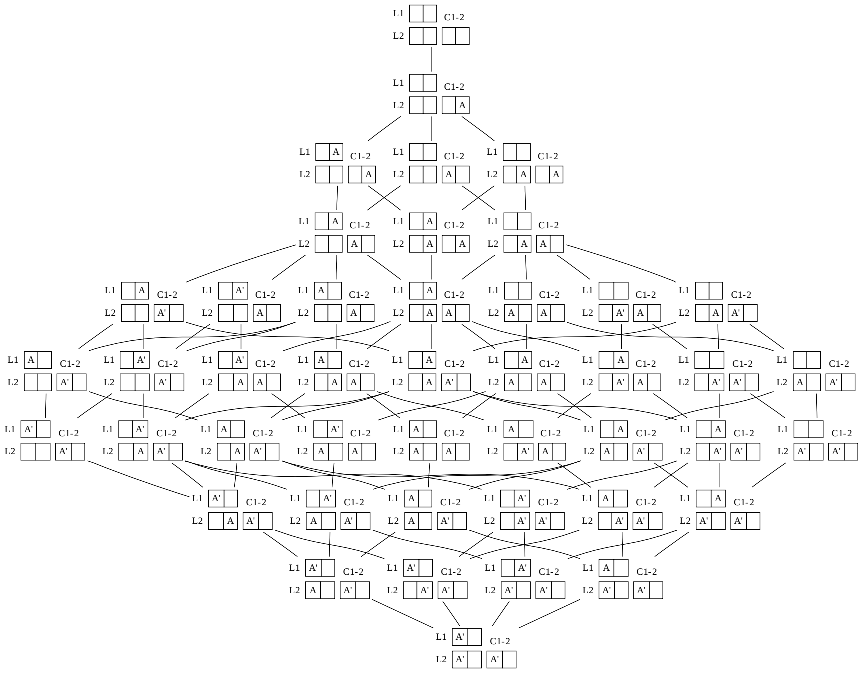


Fig. 15. Lattice with live caches for a simple case. Cases which do not exist are impossible.

This includes proofs of various properties and counter-examples to show where previous techniques lose their soundness for common situations. Next, we give examples where our technique improves upon the precision of previous work by leveraging more information available in the cache hierarchy. Then, we describe cases where previous techniques are not applicable but are analyzable via our technique.

6.1 Soundness/Unsoundness

For this evaluation, we consider previous techniques which produce sound results [3, 15,25].[1] Among these, only one technique [15] considers multi-level caches. Soundness may be necessary for a number of reasons. A sound technique will never give a false positive. When dealing with a sound technique, we know something about its behavior. An analysis that is not sound may behave in any way and we may not be able to trust its output. Soundness means that if our technique reports a hit or a miss for a reference, it is definitely a hit or a miss. This means that we will never fail to optimize code because we thought it was acceptable. For a problem like worst case execution time, a sound technique will never under-estimate the time taken for a memory access.

[1] We do not consider techniques which focus on other problems such as thread interference, pipelining, etc [6, 28–30] or those which limit their cache model ex: direct mapped only [20].

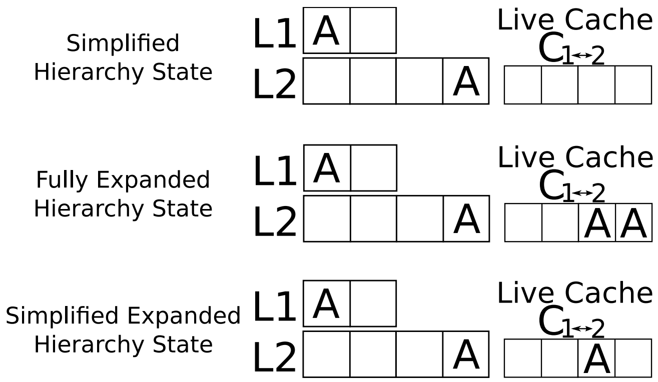


Fig. 16. Live cache behavior for monotone join.

Questions	**Description**	**Our**	**Hardy & Puauat**	**Alt *et al.***
Sound	Is the technique sound?	Yes	Yes	Yes
Multi-level	Handles caches with multiple levels	Yes	Yes	No
Policies	Which multi-level policies does it handle	All	Mainly-incl.	No
Policy combinations	Can levels have different policies?	Yes	No	No
Multi-level writeback	Handles write-backs in multi-level caches?	Yes	No	No
Lx or Ly hit	Captures references that exist in one of two levels?	Yes	No	No

Fig. 17. Comparison

Hierarchy policy Previous work [15] developed a technique to analyze multi-level mainly-inclusive caches based on a single level technique [3]. Since this was designed for mainly-inclusive instruction caches, it is unable to soundly analyze cache hierarchies that exist in practice such as those in the Intel Core i7 [17] and the AMD Phenom II [2].

Inclusive . With inclusive caches, previous work is not sound, even for analyzing the first cache level. This is because a block being evicted from a higher level cache may evict blocks from lower level caches [26]. This problem is due to the fact that in typical inclusive caches, lower level caches are not considered when evicting blocks from higher level caches [26]. For example, if a block is evicted from L2 cache, all blocks in L1 cache which are subsets of the block must also be evicted from L1. This can be multiple blocks if the *line size*$_1$ < *line size*$_2$.

Theorem 2: Previous work [15] is not sound for inclusive caches.

Proof: A counter-example is given in Figure 18. In this figure, we see that the update from the previous work [15] results in A being in L1 but not in L2. Thus, the analysis would report a hit for L1 but not L2. This is not possible for inclusive caches.

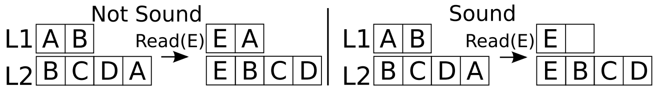


Fig. 18. Inclusive cache soundness counter-example.

Exclusive . Due to the radical differences between exclusive caches and mainly-inclusive caches, previous work is completely unable to analyze higher levels of exclusive caches. The ability to analyze exclusive caches is important since they frequently exist in practice (AMD Athlon, Phenom, etc.) [2].

Theorem 3: Previous work [15] is not sound for exclusive caches.

Proof: A counter-example is given in Figure 19. In this figure, we see that the update from the previous work [15] results in C being in both L1 and L2. Thus, the analysis would report a hit for both L1 and L2. This is not possible for exclusive caches since the levels may contain no common references.

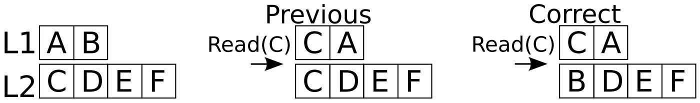


Fig. 19. Exclusive cache soundness counter-example.

Therefore, in order to analyze a wider range of systems, we develop semantics for all cache hierarchy types including exclusive and inclusive hierarchies as well as combinations of hierarchy types (Section 4 and Section 5). This is important since systems with various hierarchy policies exist in practice [2, 17].

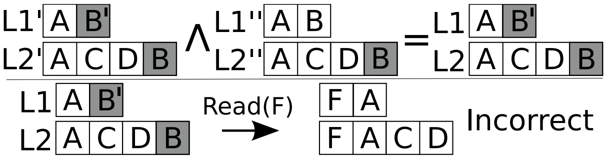


Fig. 20. Write-back soundness counter-example.

Write-back Since we are dealing with data caches, we must consider write-back. Previous work does not consider write-back [15]. When a dirty block is evicted from a cache level, the result is written to the next level of cache (or memory). For example, an evicted dirty block from L1 will be written to L2 and an evicted dirty block from the highest cache level will be written back to memory. Since this write-back is an access, it will cause the LRU order to be updated whenever a dirty block is evicted. In the presence of write-back, applying the single level join function repeatedly is not sound. For example Figure 20 illustrates a case for the miss analysis. After the join, since we are not considering write-back, we have the best case for each memory reference. Thus, in the joined state, we have that B is in the last position in the L1 cache. Next, after a read of a block the will definitely miss cache, F, we update both levels of cache which evicts both instances of B. However, since B was dirty in one of the cases, it may have been written back to L2 cache thus updating its order again and B might still be in L2. Thus, if the next statement was a reference to B, the previous analysis would report a definite miss. This is not sound since the first case in the join would result in an L2 hit.

6.2 Improved precision

To verify that live caches improve precision in practice, we compared two static analysis approaches, one with live caches and one without. Analysis implementation was done in our prototype tool which analyzes program binaries. Like previous work [15,25], we used the WCET benchmarks maintained by the Mälardalen WCET research group [1]. 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 compact representation of live caches will drastically reduce this overhead, however, producing such implementation was not the focus of this paper.

Improvement>10%	
Benchmark	**Improv.**
adpcm	15.3%
lcdnum	13.2%
duff	11.6%
insertsort	15.6%
lms	10.8%
minver	19.7%
qurt	22.6%
sqrt	17.6%
statemate	11.3%

0%<Improvement<10%	
Benchmark	**Improv.**
bs	5.0%
crc	7.1%
compress	9.4%
expint	3.0%
fft1	0.2%
janne_complex	2.9%
jfdctint	0.8%
ludcmp	0.7%
matmult	4.3%
minmax	2.9%
ns	7.8%

0% Improvement
bsort100
cnt
cover
edn
fac
fibcall
fir
fdct
loop3
nsichneu
qsort-exam
select

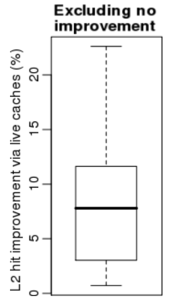


Fig. 21. L2 hit increase due to live caches. L1: 8kb 4-way, L2: 64kb 8-way. Left: improvement per benchmark, Right: boxplot for those benchmarks with non-zero improvement.

To see how well our analysis would perform for a realistic system, we used a cache configuration similar to that in the existing many-core Tile64 processor (L1: 8k, 4-way. L2: 64k, 8-way). The improvement in precision (for L2 hits) for each benchmark is shown in Figure 21. This data shows that introducing live caches into the analysis resulted in more than a 5.7% improved L2 cache hit rate for the analysis. For benchmarks reported in previous work [15] we observed an average improvement of 8.8% whereas the average improvement over all benchmarks with non-zero improvement was 9.1%. Overall, 63% of the benchmarks showed some improvement in precision. As the boxplot shows, there is a high variability between benchmarks (standard deviation of 6.2%).

Impact of program characteristics We performed a multiple linear regression analysis 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. These results are shown in Figure 22. Besides the characteristics shown in Figure 22, we also considered others such as number of loops, nesting of control structures, etc, but none of them were found to be strong

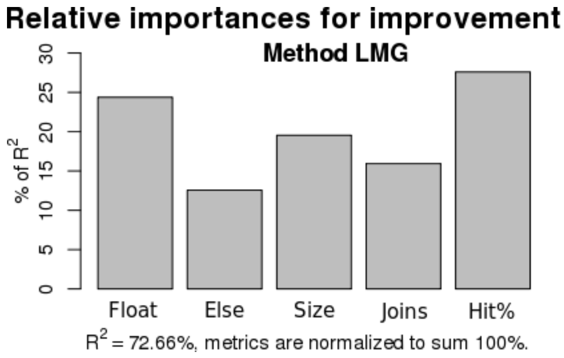


Fig. 22. Relative importance, Improv > 0

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 to see that as the program size increased precision improvement increased. This provides us with evidence that for programs larger than our current benchmarks, we are likely to 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) [7]. 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.

Impact of hardware configuration Besides the many-core 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 (Pentium Pro) to modern processors (Core 2). Our results show several noticeable trends which we now summarize.

First, we found a positive correlation between the size of L2 cache and performance 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 technique is useful for large caches as well. L1 cache sizes appear to have no significant impact on performance.

We found that associativity plays the largest role in impacting performance. Figure 23 shows average improvement for a variety of L2 associativity levels and linesizes. We see a clear negative correlation between associativity and improvement, however,

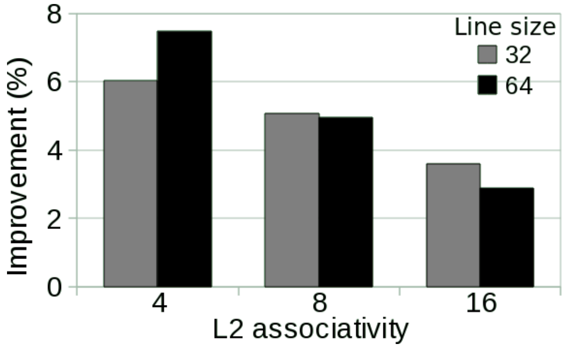


Fig. 23. Average improvement for varying L2 associativity and linesize (L1: 32k, 4-way. L2: 256k).

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.

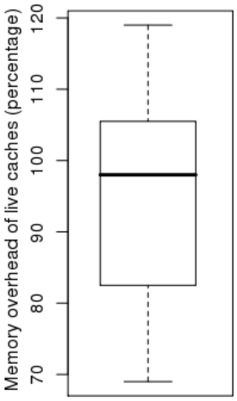
- Most programs see precision improvement.
- As program complexity and size increase, precision generally increases.
- Caches from small (Pentium Pro) to large (Core 2) see precision improvement.

6.3 Overhead

Here we give a brief theoretical discussion and empirical observations of overheads involved with live caches. Our current implementation is a nearly direct copy of the semantics presented here and is thus not optimized. Therefore, empirically measured overheads are likely highly inflated compared to those one could achieve in an efficient implementation. However, our results suggest that even in this simple implementation, scalability appears to be in line with an analysis with previous analysis.

Space As mentioned in Section 5, our approach using live caches, only requires $\binom{\mathcal{N}}{2}$ live caches at each state plus one additional live cache for each cache level. Since the number of cache levels is small (typically 2 or 3), the number of live caches needed is also small (ex: 3 for a 2 level hierarchy). Each of these live caches, $\bar{\mathcal{C}}_{x\leftrightarrow y}$, has $\max(A_x, A_y) * \max(n_x/A_x, n_y/A_y)$ total cache lines (typically n_y lines when $y \geq x$). Finally, the associativity, $A_{x\leftrightarrow y}$, of a live cache is equal to the higher of the levels it is associated with, $A_{x\leftrightarrow y} = \max(A_x = A_y)$. Typically, a live cache takes the same space as the higher associated level ($|\mathrm{L2}| = |\bar{\mathcal{C}}_{1\leftrightarrow 2}|$ where $|\mathrm{L}x|$ represents the size of $\mathrm{L}x$).

To empirically measure the space overhead involved with adding live caches to the cache analysis, we measured memory consumption (using Valgrind [22]) of the cache analysis with and without live caches. We observed that on average, live caches increase the memory consumption of the analysis by 95%. A box-plot of this data is shown



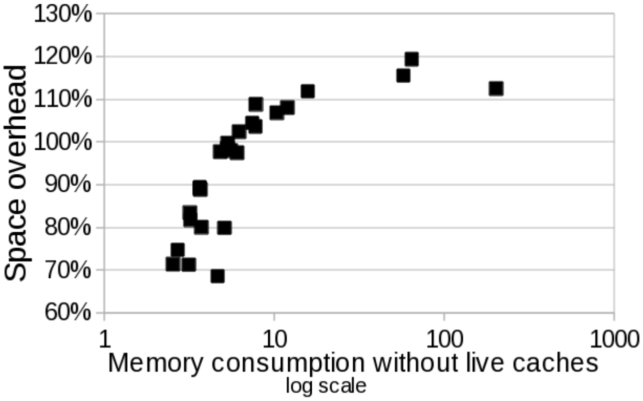


Fig. 24. Space overhead

Fig. 25. Memory use without live caches vs memory overhead

in Figure 24. This figure shows that overhead was fairly predictable being between roughly 85%-105% most of the time. A more compact representation of live caches will drastically reduce this overhead. For example, in the current implementation rather than having cache sets contain references to blocks. Currently, the blocks are duplicated for each level they are in. As a result, blocks in live caches are duplicates of blocks in standard cache levels. Nevertheless, the median memory consumption of the analysis was only 10.6MB with an inner quartile range of 6.9MB-16.1MB (std. dev. 85.4, min 4.4MB, max 429.7MB).

We found strong positive logarithmic correlations between space overhead and the following variables: number of static joins (counted joins in the code), accesses to L2 cache, and memory consumption of the analysis without live caches (illustrated in Figure 25). This means that, for example, programs with more joins have higher overhead but it appears to level off at some point (around 120% overhead). These logarithmic correlations are important since they suggest that our approach will scale as well as the analysis without live caches.

Time Overhead . To empirically measure the time overhead involved with adding live caches to the cache analysis, we measured the runtime of the cache analysis both with and without live caches. This does not include the time taken to determine which addresses each instruction accesses or other analysis (ex: control flow analysis). It only includes the time taken for the actual modeling of the cache behavior. For the WCET benchmarks, we observed that enabling live caches resulted in an average of 57.8% increased execution time for the analysis. A box-plot of the data is shown in Figure 26. This shows that most benchmarks had a fairly predictable overhead between roughly 45% and 70%. There were a couple of outliers wither higher overhead, but they were not terribly high (roughly doubled the analysis time). On average, the cache analysis took approximately 14 seconds with a median of only approximately 3 seconds.

The most interesting correlation observed was a strong positive linear correlation ($R^2 = 0.94$) between static joins (joins in the code) and run-time with live caches. This data is illustrated in Figure 27. This correlation is somewhat intuitive, but is interesting

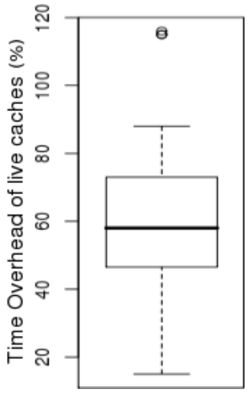


Fig. 26. Time overhead

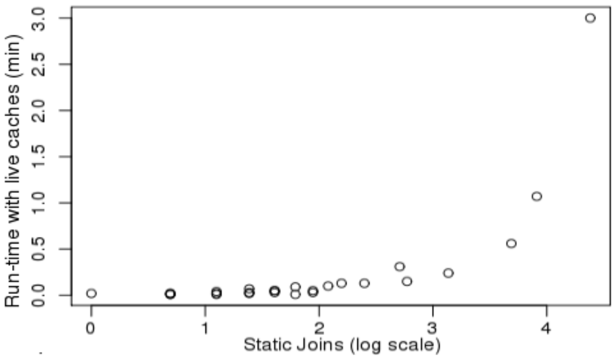


Fig. 27. Joins in the source code (log scale) vs analysis run-time

in that it allows us to reasonable estimate the run-time of the analysis by simply looking at the program.

When varying configurations, we found that analysis run-time had a positive linear correlation with both cache sizes and associativity (as expected). It is important to note that these sizes did not significantly impact the overheads.

7 Related Work

Existing work on cache behavior analysis can be divided into two categories: those that focus on instruction caches [3, 6, 15, 20, 29, 30] and those that focus on data caches [8, 23, 25, 28]. In contrast, our technique works for both instruction and data caches.

Most similar among instruction cache techniques is that of Hardy and Puaut [15] 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 this previous technique by capturing memory references that exist in different cache levels.

Another technique was proposed by Sen and Srikant [25] which uses abstract interpretation to predict data cache behavior. They make use of previous work to track addresses [4]. 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.* [3]. Like the work by Hardy and Puaut [15], 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.

Previous work exists to handle tracking address values in programs. For example, Balakrishnan and Reps [4] developed a framework to analyze memory accesses for security analysis. Similar techniques can be used to predict data cache behavior as shown by Sen and Srikant [25]. White *et al.*address the problem of classifying data references to cache lines [28], however, they do not deal with dynamic allocation which is handled soudnly with live caches. These techniques are complimentary to our work.

A technique for multi-level shared caches which considers interference [24] between threads was proposed by Yan and Zhang [30], but it doesn't handle both instruction and data caches. We don't consider interference but handle both type of caches.

8 Conclusion

Higher level cache behavior is increasingly important due to the widening gap between cache and memory speeds. Furthermore, newer architectures such as many-core systems typically have much smaller caches [11] and a higher penalty for higher level cache misses due to accessing shared memories [9, 21]. In this paper we showed examples where single level cache analysis techniques [3, 15, 25] fall short in accurately determining higher level cache behavior. These techniques lose precision when applied to multi-level caches and do not take into account realistic cache hierarchies which include split and unified cache levels as well as write-back which impacts higher level cache behavior. We have defined live caches for multi-level cache analysis which properly handles the entire cache hierarchy including write-back and captures information in the hierarchy which was lost with previous techniques. We have shown that our analysis with live caches is sound and improves precision by an average of 5.7% for L2 cache references. Furthermore, our analysis is scalable and sees improvements for all cache configurations tested ranging from small caches to large caches.

Acknowledgments

This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CAREER-08-46059. Comments and discussions with the members of the Laboratory for Software Design at Iowa State University were helpful in developing some of the ideas in this paper.

References

1. Wcet project / benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
2. Advanced Micro Devices, Inc. Family 10h AMD phenom^{TM}II processor product data sheet: Revision 3.04, February 2009. http://support.amd.com/us/Processor_TechDocs/46878_Phenom_ II_PDS_3.04_PUB.pdf.
3. Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS*. Springer-Verlag, 1996.
4. Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *In CC*. Springer-Verlag, 2004.
5. S. Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decade. *Tech. Report - Intel*, 2005.
6. C.A. Healy *et al.* Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1), 1999.

7. Jason F. Cantin and Mark D. Hill. Cache performance for selected spec cpu2000 benchmarks. *SIGARCH Comput. Archit. News*, 29(4):13–18, 2001.
8. Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, 2003.
9. Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *CCGRID*, pages 471–478, 2007.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. D. Wentzlaff *et al.* On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5), 2007.
12. Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3), 1999.
13. David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5), 2005.
14. Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 120–136. Springer, 2009.
15. Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.
16. Intel Corp. First the tick, now the tock: Next generation intel microarchitecture - nehalem, April 2009. http://www.intel.com/pressroom/ archive/reference/whitepaper_nehalem.pdf.
17. Intel Corp. Intel®64 and IA-32 architectures software developer's manual volume 1: Basic architecture, March 2009. http://www.intel.com/products/processor/manuals/.
18. N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *ISCA*, 1994.
19. K. Skadron *et al.* Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Trans. Comput.*, 48(11), 1999.
20. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3), 1999.
21. Lixia Liu, Zhiyuan Li, and Ahmed H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *ICS*, pages 359–367, 2008.
22. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.
23. Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. Static identification of delinquent loads. In *CGO*, page 303, 2004.
24. S. Lim *et al.* An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7), 1995.
25. Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.
26. Christopher Shannon, Mark Rowland, and Ganapati Srinivasa. Cache eviction technique for inclusive cache systems. In *US Patent No. 10,897,474 (US20070186045)*, 2004.
27. Tyler Sondag and Hridesh Rajan. A theory of reads and writes for multi-level caches. *Technical Report Iowa State University, Department of Computer Science, 09-20*, July 2009.
28. Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *RTAS*, 1997.
29. Jun Yan and Wei Zhang. WCET analysis of instruction caches with prefetching. In *LCTES*, 2007.
30. Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, 2008.

31. M. Zahran. Cache replacement policy revisited. In *WDDD held in conjunction with ISCA*, June 2007.
32. Ying Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *ISPASS*, 2004.