Relational Cache Analysis for Static Timing Analysis

Sebastian Hahn
Saarland University, Saarbrücken, Germany

Daniel Grund
Saarland University, Saarbrücken, Germany

***Abstract*—Static cache analysis is an indispensable part of static timing analysis, which is employed to verify the timing behaviour of programs in safety-critical real-time systems.**

State-of-the-art cache analyses classify memory references as 'always hit', 'always miss', or 'unknown'. To do so, they rely on a preceding address analysis that tries to determine the referenced addresses. If a referenced address is not determined precisely, however, those cache analyses cannot predict this reference as hit or miss. On top of that, information about other cache contents is lost upon such references.

We present a novel approach to static cache analysis that alleviates the dependency on precise address analysis. Instead of having to argue about concrete addresses, we only need to argue about relations between referenced addresses, e.g. 'accesses same memory block' or 'maps to different cache set'. Such relations can be determined by congruence analyses, without precise knowledge about the actual addresses. The subsequent cache analysis then only relies on relations to infer cache information and to classify references.

One advantage of this approach is that hits can be predicted for references with imprecisely determined addresses; even if there is *no* information about accessed addresses. In particular, this enables the prediction of hits for references whose addresses depend on an unknown stack pointer or even depend on the program input. Relational cache analysis is always at least as precise as the corresponding state-of-the-art cache analysis. Furthermore, we demonstrate significant improvements for three classes of program constructs.

I. Introduction

In hard real-time systems, operations are subject to timing constraints, i.e. there are operational deadlines from events to system responses. To show the correctness of such systems, one must therefore derive guarantees on the timeliness of reactions. Thereby, one fundamental problem is to characterise the execution time of programs [1].

The execution time of a program depends on the program's input as well as on the underlying hardware architecture. Modern microprocessors feature many components that speed up computation by speculation, e.g. branch predictors, caches, and pipelines. In such a setting, the large number of inputs and initial hardware states as well as the complexity of such processors preclude exhaustive techniques to determine exact best- and worst-case execution times. Instead, static methods that compute lower and upper bounds on the execution times have to be used.

One hardware feature that has a particularly strong influence on program execution time is caching. A cache is a small and fast memory that aims at bridging the latency gap between the processor and the main memory. To do so, it stores memory blocks that are likely to be accessed in the near future. If this mechanism works well, it reduces the average latency of memory accesses by at least one order of magnitude: If an accessed memory block resides in the cache, which is called cache hit, the memory access can be serviced within one cycle. In the other case, which is called a cache miss, a transfer from main memory may take 100 cycles.

Due to the strong influence of caches on execution time, precise analysis of cache behaviour is mandatory in static timing analysis. The goal of static cache analysis is to predict as many hits and misses as possible. The more memory accesses are predicted to result in cache hits, the better is the upper bound on the program's execution times. Likewise, the lower bound is tighter the more misses are predicted.

Most prior works on cache analysis, e.g. [2], [3], [4], [5], are similar in that they all take the following two-step approach: First, an address analysis computes for each memory reference an approximation to the set of addresses accessed by that reference. In a second step, the actual cache analysis uses this address information to infer information about cache contents.

After a recap of cache notions and an explanation of a state-of-the-art cache analysis in Section II, we will illustrate the shortcomings of such prior approaches in Section III. If the address analysis does not exactly determine the accessed memory blocks, uncertainty about accessed blocks translates into uncertainty about cache contents and ultimately shows up as an inferior performance guarantee. The reason, in short, is that the cache analyses attach information to memory blocks in order to represent their information about cache contents. If a block is unknown, information cannot be attached and is lost. Although one can displace parts of this problem with highly context-sensitive analyses, there is a more efficient, more elegant way.

Our key contribution, which we outline in Section IV, is a novel approach to static cache analysis that lowers the requirements for precise cache analysis. To solve the intangibility problem described in the previous paragraph, we introduce *the concept of symbolic names*. Symbolic names abstract from addresses such that one can argue about memory references even if the accessed addresses are unknown. Based upon this abstraction, we present two modules that make up our analysis.

The *congruence analysis module*, presented in Section V, is responsible for the determination of relations between

pairs of symbolic names. We formalise its interface, i.e. we define the meaning of congruence information, and we suggest several analyses that can be used to compute such information.

The *relational cache analysis module*, formalised in Section VI, infers cache information and classifies memory references. The information about cache contents is represented using symbolic names and additional information attached to them. The analysis itself only relies on the relations provided by the congruence analysis module.

In Section VII we discuss related work and in Section VIII we evaluate our approach by comparing it against a state-of-the-art cache analysis. For three classes of programs, each exemplified with one representative program, and over a range of cache configurations, our approach can classify up to 24% more references as always hit. The key advantage of the relational approach is that it can predict hits where prior analyses fail to do so, e.g. for a certain class of array accesses, for stack-relative accesses, and even for accesses whose addresses depend on the program input.

We close with conclusions and an outlook on future work triggered by our new approach.

II. Foundations

A. Cache Parameters and Operation

A cache is a small but fast memory that transparently buffers memory contents in order to bridge the latency gap between the processor and the large but slow main memory. For efficiency reasons this buffering is restricted by the following scheme.

A k-*way set-associative cache* consists of *cache sets*, each consisting of k equally-sized *cache lines*. k is also called the *associativity* of the cache.

Main memory is logically partitioned into equally-sized *memory blocks*. Each block can only be cached as a whole and can only be cached in a cache line of one particular cache set that is determined by the address of the block. Usually, the block size and the number of cache sets are chosen as powers of two such that an address can be easily partitioned into 3 parts: The least significant bits, called the *offset*, determine the location of a byte within a cache line. The next, more significant bits, called the *index*, determine the number of the cache set in which a memory block may be cached. The most significant bits, called *tag*, are stored along with the data to uniquely identify it.

When processing a memory access, the cache logic determines whether the requested data is stored in the cache or not. Given an address, its index part tells in which cache set to look and a comparison of the tag part determines whether the block is actually contained. If it is, this is called a *cache hit* and the requested part of the block, which is determined by the offset, is returned to the processor. Otherwise, this is called a *cache miss* and the requested block is first fetched from main memory and stored in the cache set.

The cache line to hold the newly fetched block is determined by the *replacement policy* of the cache. We will focus on the Least Recently Used (LRU) policy that replaces the block that has not been accessed for the longest time.

B. Static Cache Analysis

State-of-the-art cache analysis comprises two parts: First, an address analysis, e.g. an interval analysis [6], determines for each memory reference an approximation to the set of addresses it accesses. Then, the cache analysis uses that information to infer information about cache contents. To represent the contents of a cache set, analyses attach information to memory blocks. We also say that they use memory blocks as *abstract cache elements*.

In the following, we summarise the must- and may-cache analyses of Ferdinand et al. [4]. The information they attach to memory blocks is information about their age.

Definition II.1 (Age)**.** The *age* of a memory block b is the number of distinct blocks that have been accessed since the last access to b and map to the same cache set as b.

As the LRU policy always retains the k most recently used blocks in each cache set, a block is evicted when its age reaches k, the associativity.

The must-cache analysis (may-cache analysis) infers upper (lower) bounds on the age of blocks at each program location. These bounds are valid for all partial executions up to the respective program point. Using this approximate information, a memory reference can thus be classified as always hit (always miss): The must-cache analysis can predict hits for a block, if its age bound is less than k. Analogously, the may-cache analysis can predict misses for blocks with a lower age bound of at least k. And there is the inconclusive case, unknown, where $k-1$ and k are included between lower and upper bound.

As an example consider the actions of Ferdinand's must-cache analysis. At the program point following an access to block b, the age bound of b is (re)set to zero. If an access to another block *might* increase the age of b the bound is increased. At control-flow merge points, each block is assigned the *maximum* of its age bounds at the preceding program points. The "might" and "maximum" ensure that the attached value is indeed always an upper bound. For details please see [4] or the up-to-date formalisation in [7].

Finally, note that it is current practice to analyse cache sets independently of each other, i.e. a cache analysis is constructed as a Cartesian product of cache set analyses. As each access can only affect a single cache set, this seems like a reasonable approach. If the addresses are not precisely determined, however, it may not be clear which cache set is affected.

```
int pending = 1;

void swap (int *a) {
  if (pending) {
    int tmp1 = *a;
    int tmp2 = *(a + 1);
    *a       = tmp2;
    *(a + 1) = tmp1;
  }
  pending = 0;
}
```

Analysis setting:

- 2 cache sets
- 4-way associative
- line size = `sizeof(int)`
- LRU replacement
- `tmp`s in registers
- stack pointer unknown

Figure 1. Example program and analysis setting.

III. Motivation

In this section, we explain the drawbacks of state-of-the-art cache analyses. In particular we shed light on the problems that arise if referenced addresses are not or cannot be determined precisely.

Consider the program and the cache configuration in Figure 1. In any execution where `pending` is `1`, the write access to `*a` will be a cache hit: The cache is 4-way associative and after the preceding read access to `*a` only one other block was accessed, that at address `*(a+1)`.

Now, consider the analysis results of [4]. Assume that the address `*a` is unknown to the analysis. Any sound analysis must therefore assume that *any* memory block might be accessed upon the read from `*a`. Hence, the analysis can *not* reset the age bound of *any particular* block to zero. As a consequence, the analysis cannot guarantee that the succeeding write to `*a` is a cache hit.

Observation III.1 (Intangibility)**.** Cache analyses that use memory blocks as abstract cache elements cannot model accesses to memory blocks with imprecisely determined addresses. In particular, they cannot predict hits for such accesses.

Now consider the accesses to `pending`. After the read of `pending`, its block has age zero. In the `if` branch, five accesses take place: The read of the value of parameter `a` from the stack, followed by two reads and two writes to `*a` and `*(a+1)`. The read of `*a` and the write to `*a` access the same memory block. Analogously for `*(a+1)`. Thus, three distinct blocks get accessed and `pending` is still cached after these accesses. Its block has age three.

Now, consider the analysis results. As `pending` is a global variable, its address is statically known. After the read of `pending` the analysis can guarantee age zero for the block containing `pending`. After the read of `*(a+1)` the age bound of `pending`'s block is three as three different blocks have been accessed. As we have seen above, the analysis cannot show that the write accesses are hits. Hence, the analysis has to take two further cache misses into account, which increase the age bound of `pending` to five. As a consequence, the write to `pending` cannot be guaranteed to be a cache hit either.

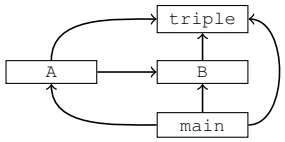


Figure 2. Call graph.

Observation III.2 (Excessive Information Loss)**.** In presence of memory accesses with imprecisely determined addresses, cache analyses that use memory blocks as abstract cache elements excessively lose information. For instance, they age cached elements overly pessimistically. In particular, m accesses to the same but unknown memory block increase the age bound of cached elements by m.

A frequently occurring source for imprecisely determined addresses are stack-relative accesses. To be able to exactly determine the address of a stack-relative access, the value of the stack pointer must be known. Its value, however, depends on the call stack of the current function.

For example, consider the call graph in Figure 2 and the function `triple` in Figure 9 on page 9, which makes heavy use of local variables. In order to precisely analyse the stack-relative accesses to those variables, prior cache analyses need to distinguish all call sites: `main`, `main.A`, `main.A.B` and `main.B`. In each of these analysis contexts, the stack pointer can be precisely determined, which is a prerequisite for prior cache analyses to be precise. Another form of context sensitivity typically employed by cache analyses is virtual loop peeling. Similarly to call sites, it allows to precisely determine addresses that depend on the loop iteration. However, high context sensitivity comes at the cost of increased analysis time—or simply is infeasible, e.g. in deeply-nested loops.

Observation III.3 (Need for Context Sensitivity)**.** Prior cache analyses need to be highly context sensitive in order to be precise, which incurs higher analysis times.

We outlined the key motivations for our work:

- Overcome the drawbacks inherent to cache analyses that are based on memory blocks as abstract cache elements.
- Reduce the requirements of cache analysis with regard to context sensitivity.

IV. Our Approach

As we have seen in Section III, using memory blocks as abstract cache elements has inherent drawbacks. Fortunately, it is not even necessary to know the precise blocks that get accessed. To predict a hit for instance, it is enough to show that the accessed block coincides with a cached one. The actual addresses are unimportant.

To be able to argue about memory references without needing to know the accessed addresses, we introduce the concept of symbolic names.

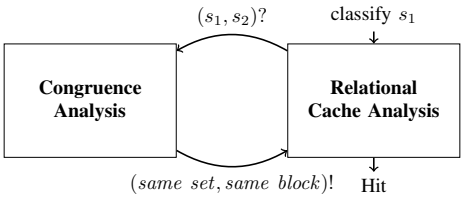


Figure 3. Interaction between congruence analysis and cache analysis.

Definition IV.1 (Symbolic Name)**.** A *symbolic name* is a name that uniquely identifies an occurrence of an address expression in a program. Note that each occurrence of an address expression in a program text corresponds to possibly many address computations during program execution.

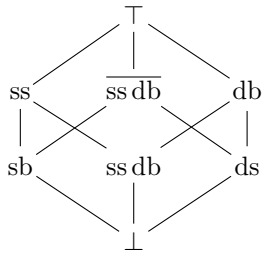
The interpretation of a symbolic name s for an occurrence o of an address expression is the following: At each point during program execution, s represents the address computed most recently at o. We call an access to the address most recently computed at o an access via symbolic name s. In short, symbolic names abstract from concrete addresses that get computed—much like program variables abstract from values that get assigned.

Instead of memory blocks, relational cache analysis uses symbolic names as abstract cache elements. That is, we attach cache information, e.g. age bounds, to symbolic names. By doing so, we avoid the problem described in Observation III.1. Even if the accessed address is unknown, we can attach information to the symbolic name that represents the address.

As already said, actual addresses are unimportant. To model cache operations, however, we need information about the relations between addresses: Do two addresses correspond to the same block? Or do they correspond to different blocks? And if the blocks are different, do those blocks map to the same cache set or to different cache sets? Answers to those questions are necessary to classify references and to avoid the problem described in Observation III.2 when computing the effect of a cache access.

Our approach comprises two modules: First, the *congruence analysis module* whose sole purpose is to compute relations between symbolic names, or more precisely, relations between the addresses represented by symbolic names. Second, the *relational cache analysis module*. With the help of the congruence analysis results, it computes approximations to cache contents, which are expressed using symbolic names.

Before we formalise the two modules in the next two sections, consider Figure 3. It demonstrates an interaction between the relational cache analysis and the congruence analysis: A client queries the cache analysis for a classification of the symbolic name s_1. For each symbolic name t that resides in the abstract cache, the cache analysis consults the congruence analysis for the relation between t and s_1. For $t = s_2$, the congruence analysis can guarantee that s_1 and s_2 represent the same memory block. The relational cache analysis can therefore conclude that the access via s_1 will result in a cache hit and can return this answer to the client.



Relation	Meaning
ss	same cache set
ds	different cache set
sb	same block
db	different block
$\mathrm{ss\,db}$	ss and db
$\overline{\mathrm{ss\,db}}$	ds or sb

Figure 4. Hasse diagram of the lattice of relations, $\mathcal{R}$.

V. Congruence Analysis

In this section we formalise the congruence analysis module. Its task is to answer questions about the relation between two symbolic names at a program point. First, we determine the kinds of relations between addresses that are interesting for cache analysis. Then, with the help of an underlying program semantics, we formalise their meaning, i.e. the meaning of congruence information. Finally, we enumerate analyses that can be used to obtain congruence information and explain how to do so.

A. Relations

As already hinted at, in cache analysis one is interested in whether an accessed memory block and a cached memory block (a) are the *same block*, in order to predict hits; (b) map to the *same cache set* but are *different blocks*, in order to argue exertion of influence; or (c) map to *different cache sets*, in order to exclude possible eviction. Hence, we define the relations sb, $\mathrm{ss\,db}$, and ds that correspond to the three cases above. Due to their incompleteness, sound static analyses usually cannot exactly determine program properties. Therefore we also introduce approximations to the above relations between blocks, which are disjunctions of the three basic relations. Figure 4 explains all those relations and shows their ordering in the lattice $\mathcal{R}$. The relation $\overline{\mathrm{ss\,db}}$, for instance, means either sb or ds. As we will see later, $\overline{\mathrm{ss\,db}}$ can still be useful in cache analysis to exclude potential evictions.

B. Intuition

Consider an *execution trace* reduced to its address computations and memory accesses. In such traces, $\langle s \mapsto u \rangle$ stands for an address computation with result u that was computed at the address expression with symbolic name s. And $\langle s \rangle$ stands for a memory access to the address computed most recently at the address expression with symbolic name s. For instance, take

$$\langle s_2 \mapsto 5 \rangle \circ \langle s_1 \mapsto 5 \rangle \circ \langle s_1 \rangle \circ \langle s_2 \rangle \circ \langle s_1 \mapsto 6 \rangle \circ \langle s_1 \rangle \circ \langle s_2 \rangle,$$

which contains accesses to the addresses 5, 5, 6, and 5. The relation of two symbolic names is the relation between the addresses that have been computed most recently at those symbolic names. For instance, the relation between s_1 and s_2 before the first access via s_1 is the relation between addresses 5 and 5, namely same block (sb). The relation between s_1 and s_2 before the second access via s_1 is the relation between addresses 6 and 5. This relation depends on the cache configuration and is either same block (sb) or different set (ds).

C. The Meaning of Congruence Information

Let $\mathcal{N}$ denote the set of symbolic names and let $\mathcal{R}$ be the set of relations defined above. Conceptually, we model the congruence information as one function

$$cgr_v : \mathcal{N} \times \mathcal{N} \to \mathcal{R},$$

per program location v. We now give meaning to cgr_v by relating it to the traces through a program. First, the relation between two addresses a and b is given by

$$rel(a,b) = \begin{cases} \text{sb} & : mb(a) = mb(b) \\ \text{ss db} & : mb(a) \neq mb(b) \wedge cs(a) = cs(b) \\ \text{ds} & : cs(a) \neq cs(b) \end{cases}$$

where $mb(b)$ denotes the memory block containing b and $cs(b)$ denotes the cache set b maps to. For a (partial) execution trace τ and $s \in \mathcal{N}$, we define

$$last(\tau, s) = \begin{cases} \bot & : \tau = \epsilon \\ b_l & : \tau = \tau' \circ \langle s \mapsto b_l \rangle \\ last(\tau', s) & : \text{otherwise, where } \tau = \tau' \circ \langle . \rangle. \end{cases}$$

$last(\tau, s)$ returns the address that has been computed most recently at the symbolic name s on the trace τ; or $\bot$ if no address computation with symbolic name s occurs on τ.

For the $\bot$-cases we extend rel to $\widehat{rel}$, which is defined as

$$\widehat{rel}(a,b) = \begin{cases} \bot & : a = \bot \vee b = \bot \\ rel(a,b) & : \text{otherwise.} \end{cases} \quad (1)$$

Usually, there are several traces through a program up to a specific program location v. Therefore, we have to compute congruence information that safely approximates congruences on all such traces. Let $\mathcal{T}_{P,v}$ be the set of traces through the program P up to the program location v. A function cgr_v safely approximates congruence information at program point v if for all symbolic names s and t as well as all traces $\tau \in \mathcal{T}_{P,v}$

$$cgr_v(s,t) \sqsupseteq \widehat{rel}(last(\tau, s), last(\tau, t)) \quad (2)$$

Note that cgr_v effectively needs to approximate relations only for traces on which both symbolic names occur. For other traces $\widehat{rel}$ is $\bot$, as defined in Equation 1, and $cgr_v(s,t) \sqsupseteq \bot$ is trivially satisfiable.

D. Computation of Congruence Information

The congruence analysis module has a clearly defined interface: given two symbolic names and a program location v, it returns a relation between the two that is valid at v. The interface can be implemented by arbitrary analyses, which satisfy Equation 2, without having to change the relational cache analysis.

The congruence analyses do not depend on the cache analysis and thus they can be performed beforehand. However, we do not store the relations for pairs of symbolic names explicitly. Instead, a relation for a pair is calculated upon demand of the cache analysis by using the results of the various congruence analyses.

In the following, we list such analyses and go into detail on how information computed by these analyses can be used to obtain relations between symbolic names.

1) Interval Analysis: For each program location and storage location, e.g. registers or memory cells, interval analysis computes an interval $[l, u]$ that includes all values the storage location may possibly hold at that program location.

In the best case one can derive the precise addresses a and b for two address expressions with symbolic names s and t. In this case, precise congruence information according to $rel(a, b)$ is available. Other cases may only allow to obtain less precise relations. As there are many possibilities to derive relational information from intervals, we only give one more example: Let $[l_s, u_s]$ and $[l_t, u_t]$ be the intervals approximating the addresses for the symbolic name s and t, respectively. If $l_t \geq u_s + linesize$, one can derive the relation db between s and t.

2) Global Value Numbering: Global value numbering (GVN) [8] is a well-known technique for detecting redundant computations and can be implemented efficiently. It assigns each expression a value number. In case two expressions have the same value number, they provably compute the same value. Thus, one can use GVN to obtain the sb relation between two symbolic names if the value numbers of the corresponding address expressions are equal.

One can also obtain other relations by combining GVN with other analyses. For instance, let s and t be the symbolic names for the address expressions $B_1 + O_1$ and $B_2 + O_2$, respectively, where B_1, B_2 denote base registers and O_1, O_2 offsets. Furthermore assume that GVN finds out that B_1 and B_2 always evaluate to the same value, that the offsets O_1 and O_2 are known constants, and that $O_1 \leq O_2$.

- If the address represented by s is aligned to the beginning of a cache line and $O_2 - O_1 < linesize$, one can conclude that s and t access the same block.[1]

[1] Alignments may be required by the hardware or the ABI; additional ones may be guaranteed by the compiler; or they may be determined by one of the aforementioned static analyses. For instance, we assume that all variables are aligned to their respective size and that each memory access affects exactly one cache line.

- In case one does not have this particular alignment information but $O_2 - O_1 < linesize$ and there are at least 2 cache sets, one can derive the relation $\overline{\mathrm{ss}\,\mathrm{db}}$.
- If $linesize \leq O_2 - O_1 \leq linesize \cdot (nsets - 1)$, then s and t affect different cache sets and one can derive the relation ds.

3) Octagon Analysis: The octagon analysis computes constraints of the form $\pm x \pm y \leq c$ and $\pm x \leq c$, where x and y are registers and c is a constant. Let s and t be symbolic names and let B_1 and B_2 be the respective address expressions. If $linesize \cdot (nsets - 1) \geq |B_1 - B_2| \geq linesize$, one can derive the ds relation. Note that this formula has to be rewritten in order to fit the syntactical structure of the octagon constraints.

4) Value-Set Analysis: The value-set analysis of Balakrishnan et al. [9] approximates values by reduced interval congruences (RIC). An RIC can be represented by a 4-tuple (a, b, c, d) that denotes the values described by $\{a \cdot N + d \mid N \in [b, c]\}$. This domain is more expressive than the interval domain presented above. Further information about its use as congruence analysis can be found in [10].

VI. Relational Cache Analysis

In the last section, we have defined the meaning of relations, the validity of congruence information, and how this information can be computed. Now, we show how this information can be used to do precise cache analysis, also in presence of accesses with imprecisely determined addresses. The overall analysis is based on abstract interpretation [11]; see [10] for a complete formalisation with correctness proofs.

A. The Abstract Domain

First, we define which abstract cache information we want to maintain and how we can represent this information. Similar to Ferdinand's cache analysis, we maintain (upper) bounds on the age of cache elements, but we use symbolic names instead of memory blocks as abstract cache elements. Another difference is that we do not analyse cache sets independently: As we have no information about the absolute addresses, we do not know which cache sets are affected by accesses via a symbolic name s. Thus, the block represented by a symbolic name s might be contained in any cache set. No matter to which cache set the accessed address a maps to, the age of a during execution is bounded by the age bound of s. The set of abstract relational caches is defined by

$$\mathcal{C}^{\leq}_{rel} := \mathcal{N} \to \mathcal{AB} = \mathcal{N} \to \{0, \ldots, k-1, \infty\}.$$

$ab(s) = i$ denotes that a memory block represented by s has at most age i during execution. In case $ab(s)$ is ∞ the represented block is not guaranteed to still reside in the cache.

An abstract relational cache can be graphically represented by an LRU stack; an example is shown in Figure 5. The symbolic names in the i-th row are guaranteed to have at most age $i - 1$. Symbolic names that are not guaranteed to be in the cache are not visualised. The topmost symbolic names, for instance, have been referenced most recently. Note that several symbolic names can reside in the topmost row: Two symbolic names s and t can both have age zero if $cgr_v(s, t) = \mathrm{ds}$.

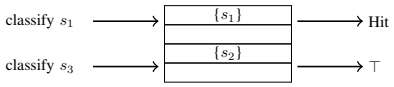
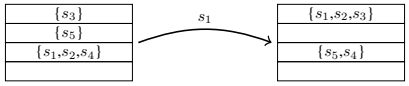


Figure 5. Classification of symbolic names s_1 and s_3.



$cgr_v(s_1, s_1) = \mathrm{sb}$, $cgr_v(s_1, s_2) = \mathrm{sb}$, $cgr_v(s_1, s_3) = \mathrm{ds}$,
$cgr_v(s_1, s_4) = \mathrm{ss}\,\mathrm{db}$, $cgr_v(s_1, s_5) = \mathrm{ss}$, $cgr_v(s_1, s_6) = \bot$

Figure 6. Update of an abstract cache upon a reference with symbolic name s_1.

B. Normalisation

Consider two different symbolic names s and t that are guaranteed by the congruence analysis to denote the same block. In such a case, the abstract domain maintains two age bounds, $ab(s)$ and $ab(t)$, which might differ. Intuitively, each age bound restricts the maximal age of both symbolic names. To normalise the representation of such equivalent information we introduce the function *effective age bound*, $eab : (\mathcal{N} \to \mathcal{AB}) \times (\mathcal{N} \times \mathcal{N} \to \mathcal{R}) \to (\mathcal{N} \to \mathcal{AB})$:

$$eab(ab, cgr_v) = \lambda s. \min\{ab(t) \mid cgr_v(s, t) = \mathrm{sb}\},$$

where $\min \emptyset = \infty$. Besides gaining precision, normalisation of age bounds simplifies operations on abstract elements and allows for a more compact abstract domain with the same expressive power. In the following we therefore assume that abstract elements are normalised. For details, see [10].

C. Classification Function

The classification function, illustrated in Figure 5, classifies memory references as always hit (H), always miss (M), or unknown ($\top$). With the domain and its meaning defined, it is straightforward to define a classification function. An access via a symbolic name s results in a hit if the age bound of s is less than ∞. We define $Class^{\leq}_{rel} : \mathcal{C}^{\leq}_{rel} \times \mathcal{N} \to \{\mathrm{H}, \mathrm{M}, \top\}$ as

$$Class^{\leq}_{rel}(ab, s) := \begin{cases} \mathrm{H} & : ab(s) < \infty \\ \top & : \text{otherwise} \end{cases}$$

D. Update Function

The update function describes the effect to an abstract relational cache ab upon a reference with symbolic name s.

The function $U^{\leq}_{rel} : \mathcal{C}^{\leq}_{rel} \times \mathcal{N} \to \mathcal{C}^{\leq}_{rel}$ is defined as $U^{\leq}_{rel}(ab, s) :=$

$$\lambda t. \begin{cases} 0 & : c = sb \\ ab(t) & : c \in \{\mathrm{ds}, \overline{\mathrm{ss\,db}}\} \\ ab(t) & : c \sqsupseteq \mathrm{ss\,db} \wedge ab(s) \leq ab(t) \\ ab(t)+1 & : c \sqsupseteq \mathrm{ss\,db} \wedge ab(s) > ab(t) \wedge ab(t) < k-1 \\ \infty & : c \sqsupseteq \mathrm{ss\,db} \wedge ab(s) > ab(t) \wedge ab(t) \geq k-1 \\ \infty & : c = \bot \end{cases}$$

where $c = cgr_v(s, t)$.

For explanation, consider Figure 6. The symbolic names s_1 and s_2 are assumed to denote the same memory block, $cgr_v(s_1, s_1) = cgr_v(s_1, s_2) = \mathrm{sb}$. Thus their age bounds can be (re)set to zero upon the reference with s_1. s_4 denotes a different block that maps to the same cache set as the one represented by s_1, $cgr_v(s_1, s_4) = \mathrm{ss\,db}$. However, since the age bound of s_4 equals the one of s_1, s_4 does not need to be aged: Either the block represented by s_1 is younger than the block represented by s_4—s_4 is not affected by an access via s_1—or the block represented by s_4 is younger than the block represented by s_1—an access via s_1 cannot age s_4 beyond $ab(s_1)$ which is equal to $ab(s_4)$. Symbolic name s_5 denotes a block that maps to the same cache set as the one represented by s_1, $cgr_v(s_1, s_5) = \mathrm{ss}$. Since $ab(s_5) < ab(s_1)$, s_5 is conservatively aged by one. In case the age bound would already have been $k-1$, the element might be evicted from the cache. The symbolic name s_3 is guaranteed to map into a different cache set than s_1, $cgr_v(s_1, s_3) = \mathrm{ds}$. Therefore s_3 does not need to be aged. Note that in case one could only obtain $cgr_v(s_1, s_3) = \overline{\mathrm{ss\,db}}$, s_3 also does not need to be aged. Either s_1 and s_3 represent the same memory block, in which case the age bound of s_3 could be set to zero; or they represent memory blocks that map to different cache sets, in which case s_3 is not affected by an access via s_1. Finally, consider the case of symbolic name s_6, where $cgr_v(s_1, s_6) = \bot$. On all traces up to v there must be an address expression for s_1 because s_1 is just being referenced, but there is no address expression for s_6. Otherwise, by Equation 2, $cgr_v(s_1, s_6)$ could not equal $\bot$. As no address was computed for s_6, s_6 cannot have been referenced before, thus cannot be in the cache, and therefore its age bound is set to ∞.

For the sake of completeness, let us discuss the update function for address expressions. Let s be the symbolic name for the occurrence o of an address expression. During execution, upon an address computation at program location o, the meaning of s might change compared to previous address computations at o. Remember that s represents the address computed *most recently* at o. Thus, the cache information for s should be invalidated at o, i.e. the age bound of s should be set to ∞. However, one can show that the age bound of s at the program location o is ∞ anyway. Hence, invalidation is not necessary.

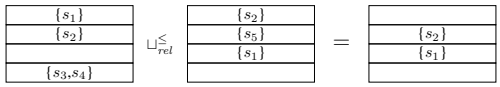


Figure 7. Two abstract caches and their join.

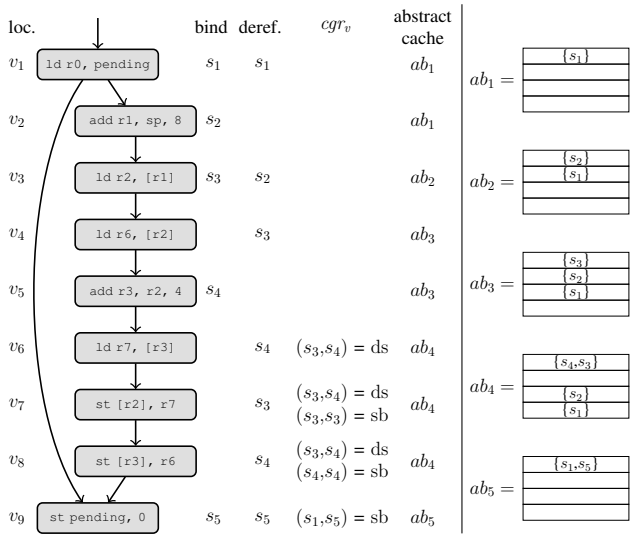


Figure 8. Results of the relational cache analysis.

E. Join Function

Finally, we define how to safely combine analysis information. The join function, illustrated in Figure 7, is structurally similar to the join function of Ferdinand's must-cache analysis. To be sound, one has to take the maximum of the two age bounds for each cache element. We therefore define the join function as

$$ab_1 \sqcup^{\leq}_{rel} ab_2 = \lambda s \in \mathcal{N}. \max(ab_1(s), ab_2(s)).$$

F. Example

Consider again the function `swap` and the cache configuration in Figure 1 on page 3. Figure 8 shows the control-flow graph of `swap`; the column on the left enumerates the program locations; the *bind* column shows the symbolic names that are attached to the occurrences of address expressions; the *dereference* column shows the symbolic name that represents the accessed address; the cgr_v column lists relevant congruence information valid before program location v; and the *abstract cache* column shows the abstract cache after the corresponding program location. In the following we explain how those results were computed.

Congruence Analysis: In the first of the three phases, we attach unique symbolic names to each occurrence of an address expression. Relational information between pairs of symbolic names is computed in the second phase. Consider the relations $cgr_{v_6}(s_3, s_3)$ and $cgr_{v_6}(s_3, s_4)$, valid before v_6. s_3 is the symbolic name for `ld r2, [r1]`, which computes the address `a` in the original program. Thus, after

this address expression, $cgr_{v_6}(s_3, s_3) = \mathrm{sb}$ holds. Using the results of GVN and information about the offsets of address expressions, we can find out that the addresses represented by s_3 and s_4 differ by four. Since we assumed that the size of a cache line equals the size of an `int` and we have two cache sets, we get $cgr_{v_6}(s_3, s_4) = \mathrm{ds}$ as described in Section V-D.

Relational Cache Analysis: We only explain the most interesting updates. First, we consider v_6, the load via s_4 and the corresponding update of the abstract cache. As $cgr_{v_6}(s_4, s_1) = cgr_{v_6}(s_4, s_2) = \top$ and an access via s_4 might bring a new element to the cache, we have to pessimistically assume that s_1 and s_2 both age by one. s_3 does not need to be aged because the access via s_4 affects a different cache set, $cgr_{v_6}(s_3, s_4) = \mathrm{ds}$. Now, consider v_7, the store via s_3. As s_3 is already guaranteed to have age zero, no other cache elements have to be aged.

Interpreting the Results: Remember that before v_7, s_3 is guaranteed to be in the cache. Thus the memory reference at v_7 is classified as always hit. In a similar way, the references at v_8 and v_9 are classified as always hit. All three references could not be classified using Ferdinand's analysis (see Section III): the first two due to imprecise address information and the latter due to pessimistic updates in the presence of accesses to imprecisely determined addresses. One can show that relational cache analysis generally is at least as precise as Ferdinand's analysis, given that one employs the address analyses of the block-based approach as congruence analyses in the relational framework.

VII. Related Work

Most of the prior work on cache analysis, e.g. [2], [3], [4], [5], takes the two-step approach already explained in Section II: First, an address analysis, e.g. an interval analysis, computes an approximation to the set of possibly accessed addresses for each memory reference. Then the actual cache analysis uses this address information to infer information about cache contents. If addresses cannot be determined precisely, the problems explained in Section III arise.

The analyses of Ferdinand et al. [4] are based on abstract interpretation and have already been explained in Section II. White et al. [5] propose static cache simulation for direct-mapped data caches and set-associative instruction caches. They compute may-cache, dominator and post-dominator information in order to classify the memory references. Sen et al. [3] propose Circular Linear Progressions (CLPs) for address analysis. Those are more expressive than intervals and allow to precisely model memory references that depend on loop induction variables. Their approach extends Ferdinand's must-cache analysis such that CLPs can be used as address information.

All the above approaches alleviate their problems with imprecisely determined addresses by high context sensitivity: By distinguishing between different analysis contexts, the address analysis can compute sharper information within each context. For instance, when distinguishing *all* call strings, the value of the stack pointer can be precisely determined—within each context. The downside of higher context sensitivity is, of course, higher analysis time.

Our relational cache analysis has weaker requirements: Instead of addresses, only relations between addresses are required. Thus any analysis whose results can be used to characterise relations between addresses can be employed. This includes all value analyses [6], e.g. interval analysis, CLPs, and octagons; as well as congruence analyses [12], e.g. linear congruence relations; and others such as global value numbering [8].

Blieberger et al. [13] present an approach to analytically derive a cache hit function at compile-time that bounds the number of cache hits. In a first step, they generate a symbolic tracefile: Symbolic expressions and conditional recurrences are employed to give a constructive description for all possible memory accesses in chronological order. In a second step, symbolic cache evaluation derives an analytical cache hit function from the generated symbolic tracefile by transforming the conditional recurrences into unconditional ones. Their *symbolic expressions* are similar to our notion of *symbolic names*. However, we employ *relations between symbolic names* in order to describe interferences instead of performing an expensive recurrence transformation.

The cache analysis proposed in [14] can be seen as a special case of our framework. Using our terminology, one can say that the relational view on accesses is restricted to the sb relation whereas we also take other relations, e.g. ds and $\overline{\mathrm{ss}\,\mathrm{db}}$, into account. The cache analysis itself is specialised to the use of alignment information and difference bound matrices (DBM). In our approach the derivation of relational information is encapsulated in the congruence analysis module, which could also include alignment information and DBMs.

Besides analyses that classify individual memory references, there are other approaches that argue about sets of references, e.g. persistence analysis. For a discussion of their pros and cons please refer to [7].

VIII. Evaluation

We implemented the relational cache analysis framework in the program representation FIRM (http://www.libfirm.org), which is originally used in compilers. The cache analysis is performed on the x86-assembler low-level representation of the programs to be compiled. As congruence analyses we employ an interval analysis and global value numbering. With these simple congruence analyses, we already obtain good results.

We identified three classes of programs whose worst-case execution time analysis can take advantage of our relational cache analysis. We now discuss the results of one representative program for each of these classes. We benchmark our relational cache analysis and Ferdinand's

```
int triple(int a1, int a2, int a3,
           int b1, int b2, int b3,
           int c1, int c2, int c3) {
   // s1 = (a × b) · c
   int p1 = a2 * b3 + a3 * b2;
   int p2 = a3 * b1 + a1 * b3;
   int p3 = a1 * b2 + a2 * b1;
   int s1 = p1 * c1 + p2 * c2 + p3 * c3;
   // s2 = (a × c) · b
   ...
   // s3 = (b × c) · a
   ...
   return s1 + s2 + s3;
}
```

Configuration			SP = 0xc000		SP ∈ [0xc000, 0xc010]	
ls	k	n	Ferdinand's	Relational	Ferdinand's	Relational
4	4	4	18	18	0	14
4	8	4	18	18	0	15
8	4	4	25	25	0	15
8	8	4	25	25	0	18
16	4	4	28	28	0	18
16	8	4	28	28	0	18

Figure 9. Function that computes the sum of three triple products, with 32 references in total. Table that shows the number of references predicted as always hit.

must-cache analysis [4] for different cache configurations (line size ls, associativity k, and number of sets n) and stack pointer configurations (precisely and imprecisely determined stack pointer).

1) Stack-relative Memory Accesses: Figure 9 shows the function `triple`, which makes heavy use of local variables. Since there are many local variables but only few registers on the target machine, the compiler needs to spill some locals on the stack. The addresses of the spill slots depend on the stack pointer. To predict hits for such references, prior analyses need to distinguish the call sites of `triple` to obtain precise address information. This results in higher analysis time, because one analysis instance per call site is required. Using our relational analysis, we are also able to predict hits in case of an unknown stack pointer value.

The table in Figure 9 shows the analysis results for the function `triple`. In case of a precisely determined stack pointer, the relational cache analysis produces the same number of cache hit predictions as Ferdinand's analysis.

In case of an imprecisely determined stack pointer, Ferdinand's cache analysis is not able to predict any hits—as expected. The relational cache analysis still classifies up to 83% of the references as hits, that it has classified as hits in the case of the precise stack pointer. The analysis results are naturally less tight in this case because the congruence information is more precise in case of a known stack pointer value.

2) Array Accesses within one Loop Iteration: Figure 10 shows a modified function taken from `ludcmp.c` of the Mälardalen benchmark suite [15]. The references to array `a[50][50]` are interesting. Again, there is the possibility to increase context sensitivity, i.e. virtually unrolling the loops in order to obtain the exact address for each reference. However,

```
int a[50][50], b[50];

int main(void) {
   int i, j, n = 50, w;
   for (i = 0; i <= n; i++) {
      w = 0;
      for (j = 0; j <= n; j++) {
         a[i][j] = (i + 1) + (j + 1); //reference (a)
         if (i == j)
            a[i][j] *= 10;            //reference (b)
         else
            a[i][j] *= 2;             //reference (c)
         w += a[i][j];                //reference (d)
      }
      b[i] = w;
   }
   return 0;
}
```

Configuration			SP = 0xc000		SP ∈ [0xc000, 0xc010]	
ls	k	n	Ferdinand's	Relational	Ferdinand's	Relational
4	4	4	0	3	0	3
4	8	4	0	3	0	3
8	4	4	3	6	0	3
8	8	4	3	6	0	3
16	4	4	5	8	0	3
16	8	4	5	8	0	3

Figure 10. Array accesses within one loop iteration, with 13 references in total. Table that shows the number of references predicted as always hit.

```
void fdct(int *blk, int lx) {
   ...
   block=blk;
   for (i = 0; i<8; i++) {
      ... = block[0]    + block[7*lx];
      ... = block[2*lx] - block[5*lx];
      ... = block[3*lx] - block[4*lx];
      ...
      block[0]    = ...
      block[4*lx] = ...
      block[2*lx] = ...
      ...
      block++;
   }
}
```

Configuration			SP = 0xc000		SP ∈ [0xc000, 0xc010]	
ls	k	n	Ferdinand's	Relational	Ferdinand's	Relational
4	4	4	11	15	0	7
4	8	4	26	37	0	17
8	4	4	36	42	0	11
8	8	4	54	67	0	21
16	4	4	56	67	0	14
16	8	4	73	84	0	32

Figure 11. Accesses with input-dependent addresses, with 123 references in total. Table that shows the number of references predicted as always hit.

this is not feasible for programs with deeply nested loops due to exponential increase in analysis time. Our relational cache analysis is able to predict hits for the last three references to `a[i][j]` (namely references `(b)`, `(c)` and `(d)`) in the inner loop—without any loop transformation or any context sensitivity.

The results are illustrated in Figure 10. The relational analysis always classifies three more references as hits than the one of Ferdinand, namely references `(b)`, `(c)` and `(d)`.

3) Input-dependent Memory Accesses: Figure 11 shows a function taken from `fdct.c` of the Mälardalen benchmark

suite [15]. The base address of the array as well as the indices of the array references are unknown as they depend on the function parameters. Therefore, state-of-the-art cache analyses are not able to classify any of these references as always hits. In this case, increasing context sensitivity is futile and distinguishing all parameter values is infeasible. Relational cache analysis therefore not only reduces analysis time (due to lower demands regarding context sensitivity); it also enables precise analysis of a new class of programs.

The results are illustrated in Figure 11. In case of a precisely determined stack pointer, the relational cache analysis is able to classify up to 13 additional references as always hits—24% more than Ferdinand's cache analysis. These hit classifications correspond to the write references at the end of the loop. The other hit classifications correspond to stack-relative references as the computation itself uses a lot of local variables.

In the second scenario with the imprecisely determined stack pointer, Ferdinand's cache analysis cannot classify any reference as always hit—neither the input-dependent array references nor the stack-relative references—whereas our relational cache analysis is able to classify them as hits.

IX. Summary and Conclusions

We have presented relational cache analysis that abstracts from concrete addresses using *symbolic names* as representatives for occurrences of address expressions. This abstraction is key as it enables the representation of information about references with unknown or imprecisely determined addresses. The *congruence analysis module*, for which we have pointed out four useful analyses, provides relations between symbolic names. These relations are subsequently used by the *cache analysis module* in order to precisely update abstract cache states and to classify memory references.

Relational cache analysis does away with the misconception that precise address analysis is necessary for cache analysis. Prior analyses required absolute address information; our approach only requires relations between addresses. As the latter can be deduced from the former, but not vice versa, the relational framework lowers the requirements for cache analysis. Future cache analyses should acquit themselves of this legacy and employ the relational framework instead.

Relational cache analysis is always at least as precise as Ferdinand's analysis. Additionally, relational cache analysis enables the analysis of new program classes: It can predict hits for memory references with input-dependent addresses and does not lose cache information upon such references. Furthermore, it can classify a large amount of stack-relative references and array references as hits, even if context sensitivity is limited or null.

X. Future Work

This work presented a must-analysis for LRU replacement caches. Analogously to this must-analysis, it is straightforward to define a may-analysis in the relational framework. Then, the evaluation should be supplemented by empirical, quantitative results for a broad range of benchmarks.

Furthermore, there are other replacement policies, e.g. first-in first-out. Is there a generic way to lift any cache analysis that uses memory blocks as abstract cache elements to a relational cache analysis?

Which other congruence analyses are particularly useful for which kind of language feature or program class?

Acknowledgements

We thank Jan Reineke and the anonymous reviewers for their comments on this paper. This work was supported by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

References

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

[2] D. Grund and J. Reineke, "Precise and efficient FIFO-replacement analysis based on static phase detection," in *ECRTS*. IEEE, 2010.

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

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

[5] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon, "Timing analysis for data caches and set-associative caches," in *RTAS*. IEEE, 1997.

[6] B. Jeannet and A. Miné, "APRON: A library of numerical abstract domains for static analysis," in *Computer Aided Verification*. Springer, 2009.

[7] D. Grund, "Static cache analysis for real-time systems – LRU, FIFO, PLRU," Ph.D. dissertation, Saarland University, 2012.

[8] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *POPL*, 1988.

[9] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*. Springer, 2004.

[10] S. Hahn, "Towards relational cache analysis," Bachelor's Thesis, Saarland University, 2011.

[11] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977.

[12] M. Müller-Olm and H. Seidl, "A generic framework for interprocedural analysis of numerical properties," in *Static Analysis Symposium*. Springer-Verlag, 2005.

[13] J. Blieberger, T. Fahringer, and B. Scholz, "Symbolic cache analysis for real-time systems," *Real-Time Syst.*, vol. 18, 2000.

[14] S. Wegener, "Improving static analysis of loops," Master's thesis, Saarland University, 2011.

[15] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *WCET*. OCG, 2010.