Computing *Same Block* Relations for Relational Cache Analysis

Simon Wegener

AbsInt Angewandte Informatik GmbH
Science Park 1, 66123 Saarbrücken, Germany
wegener@absint.com

Abstract

In contrast to the classical cache analysis of Ferdinand, the relational cache analysis does not rely on precise address information. Instead, it uses *same block* relations between memory accesses to predict cache hits. The relational data cache analysis can thus also predict cache hits if fully unrolling a loop is not feasible during analysis, for example due to high memory consumption or long computation time. This paper proposes a static analysis based on abstract interpretation which is able to compute *same block* relations for relational cache analysis.

1998 ACM Subject Classification C.4 Performance of Systems, D.2.4 Software/Program Verification

Keywords and phrases Cache Analysis, WCET Analysis, Real-time Systems, Static Program Analysis, Abstract Interpretation

Digital Object Identifier 10.4230/OASIcs.WCET.2012.25

1 Introduction

In his doctoral thesis [4], Ferdinand proposed an analysis based on abstract interpretation to predict the contents of set-associative caches with LRU replacement policy.

The analysis is split in two parts. One part, the so-called *must analysis* is used to predict definite cache hits ("always hit") by computing an under-approximation of the possible cache contents at any program point. The other part, called *may analysis* is used to predict definite cache misses ("always miss") by computing an over-approximation of the possible cache contents. For those memory accesses where neither the must analysis nor the may analysis are able to predict a definite result, "not classified" is returned.

One requirement of the cache analysis described above is the knowledge of precise address information for the targets of memory accesses. This information is typically available if instruction caches are analyzed since the control flow and thus the addresses of instructions have been computed beforehand. For data caches or unified instruction / data caches, this requirement cannot always be fulfilled.

Consider for example an array access depending on a loop counter as in listing 1. As a data-flow analysis computes invariants which hold for each and every execution of a program, it can only compute a set or an interval of possible addresses. If for example a is the array's base address and w the width of an array element, then the address interval would be

Listing 1 Array summation in C.

```
for (i = 0; i < 128; i++)
    sum += arr[i];
```

12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012).
Editor: Tullio Vardanega; pp. 25–37

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$[a, a + 127 \cdot w]$. In this case, all abstract cache sets which possibly would handle one of these addresses must be updated. For the may sets, this means that all the addresses included in the interval must be added. For the must sets, this means that none of these addresses are added, but all entries age by one. Ferdinand's cache analysis thus reacts very sensitively to imprecise address information.

One way to overcome this obstacle is to fully unroll the loop. Then, the different memory accesses can be distinguished by the analysis. However, fully unrolling may not be feasible, for example if the loop iterates several thousand or even million times. No information would reside in the abstract data cache sets inside such a loop. The memory accesses could then not be classified as either cache hits or misses.

Recent research [15, 9] studied how the dependency of the cache analysis on precise address information can be reduced. The result is the so-called relational cache analysis. There, symbolic names are used to identify memory accesses. Cache hits are predicted with the help of *same block* relations. These relations describe sufficient conditions whether two memory accesses target the same cache line. Ferdinand's cache analysis can be seen as an instance of the relational cache analysis framework, with cache address equality as *same block* relation.

The goal of this paper is to define additional analyses which can be used to compute *same block* relations. These can then be used in the relational cache analysis framework described in [9] to predict cache hits.

2 Cache Configuration

In the following (and if not stated otherwise), the cache configuration is assumed to be a write-through, write-allocate 2-way set associative LRU cache with a cache line size of 32 bytes. The whole memory is assumed to be cached. No cache locking takes place. Upon a miss, a whole line is loaded into the cache.

3 Predicting Cache Hits: Globally Precise Address Information vs. Locally Precise Address Information

A shortcoming of Ferdinand's cache analysis is its dependency on globally precise address information. This dependency on the exact position of a memory access in the whole address space is quite natural if we look at the definition of a cache hit in the concrete domain:

$$\text{hit}(a) \stackrel{\text{def}}{=} \exists i \in \{1, 2\} : \text{cacheset}(a)[i].\text{valid} \wedge \text{cacheset}(a)[i].\text{tag} = \text{tag}(a)$$

A memory access is a cache hit if and only if there is an entry in the cache set which is valid and which stores the tag of the access address a (see also Figure 1). To compute the tag and the cache address, the exact memory address is needed. If this information is not available, we cannot evaluate the formula above.

Now recall the array summation example: There, some cache hits will occur after a cache miss, because a cache miss loads whole cache lines into the cache. The subsequent memory accesses will then lead to cache hits as long as they target the same cache line. How can we use this fact to predict cache hits if we do not have the exact address information?

Fortunately, there is a possibility to specify whether two memory accesses target the same cache line which does not depend on the exact addresses. Let a_{prev} be the address of the memory access previous to the ongoing access (with address a). Assume that the distance $x = a - a_{prev}$ between the two access addresses is known as well as the position

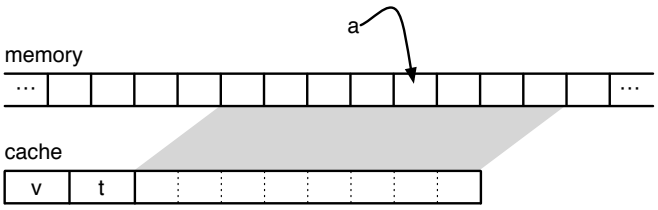


Figure 1 A cache consisting of only one cache line. The tag t maps the cache line to a specific interval of memory locations. The memory access to a is a hit if $\text{tag}(a) = t$ and the valid bit v is set. The value of t depends on the memory access before the one to a.

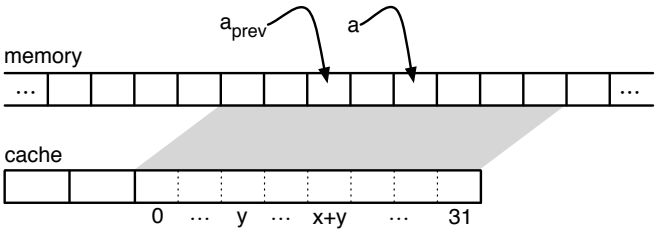


Figure 2 A cache consisting of only one cache line. a is the address of the ongoing memory access, a_{prev} is the address of the previous access, $x = a - a_{prev}$ is the distance between the two accesses and $y = a_{prev} \bmod 32$ is the position of the previous access inside the cache line. The memory access to a is a hit if the cache line offset of a_{prev} plus the distance between a and a_{prev} is smaller than the cache line size.

of the previous access inside the cache line $y = a_{prev} \bmod 32$. Then, the ongoing memory access is a cache hit if $0 \leq x + y < 32$ (see Figure 2).

The values of x and y can be computed without knowing the exact values of a and a_{prev} (see section 4). Locally precise address information is thus enough to predict cache hits.

4 Computing *Same Block* Relations

In this section, the findings from the previous section are formalized and used to build an analysis which is able to compute *same block* relations for the relational cache analysis.

4.1 Alignment Information

One information used to compute the *same block* relation is the relative position of a memory access inside a cache line. To compute the relative position, we use a static analysis [7] using arithmetical congruences as abstract domain. Values are identified by the congruence class to which they belong. The 32-bit address a for example is described by the pair $\langle a, 2^{32} \rangle$ because $a \equiv a \pmod{2^{32}}$ holds. The two addresses a and $a + 2$ are described either by the pair $\langle 0, 2 \rangle$ or by the pair $\langle 1, 2 \rangle$, depending whether a is even or odd.

Recall the array access example. If an element of the array has a size of four bytes, the analysis deduces that the address of each memory access inside the loop goes to the same congruence class (modulo 4). Formally speaking, the analysis computes the pair $\langle y, 4 \rangle$, that is, the equation $\exists y \in \{0, \ldots, 3\} : \forall i \in \{0, \ldots, 127\} : a_i \equiv y \pmod 4$. If the array is properly 32-bit aligned, the analysis can even deduce that $y = 0$.

However, this information is not precise enough for our needs, because not the whole cache line is covered by congruence classes, i.e. the modulus is to small. To improve the precision, some of the addresses must be kept apart (see section 4.2).

4.2 Loop Peeling and Loop Unrolling

The precision problems arise from the the fact that the computed invariants must hold for each and every loop iteration. Thus only the least common divisor survives as modulus.

Skillful application of loop peeling (listing 2) and loop unrolling (listing 3) can be used to keep some of the address apart. 4-fold unrolling of the loop in the running example can be used to improve the alignment information such that the four pairs $\langle 0, 16\rangle$, $\langle 4, 16\rangle$, $\langle 8, 16\rangle$ and $\langle 12, 16\rangle$ are computed for the four array accesses (assuming a properly aligned array). Loop peeling is used to control which alignment information pair is computed for what unrolled array access (see section 6 for more details).

The loop unrolling and loop peeling can be done either directly by using the corresponding loop transformations or virtually by using a specially tailored context mapping (e.g. VIVUM [15]).

Listing 2 Loop peeling.

```
sum += arr[0];
sum += arr[1];
sum += arr[2];
for (i = 3; i < 128; i++) {
    sum += arr[i];
}
```

Listing 3 Loop unrolling.

```
for (i = 0; i < 128; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
}
```

4.3 Distance Relations

Often, a variable is not invariant inside a loop, but changes its value. Thus, static analyses can only compute an abstraction of the possible values, for example an interval. This abstraction might be very imprecise. One possibility to improve the results is to check how the value evolves over time.

In the running example, the address of the memory access is increased by the size of one element in each iteration. More formally, the equation $\forall i \in \{1, \ldots, 127\} : a_i - a_{i-1} = x$ holds, where x is the size of one array element.

Such linear relations can be expressed with difference bound matrices (or short DBMs). DBMs have been introduced by Dill [3] to express clock zones for the model-checking of timed-automata. Miné [13] used them to build an abstract domain which efficiently handles a restricted form of linear relations, namely those of the form $x_i - x_j \leq c$.

In a DBM, the row / column positions identifies which variables are used in the relation and the value inside the DBM is the constraining factor (i.e. the c above). The first row / column is used for the special variable x_0 which is used to express the constant zero. Infinity is used as value if the difference cannot be bounded.

If again 4-fold unrolling is applied to the loop in the example, one gets the four DBM variables $x_1, \ldots, x_4$. The analysis computes the DBM in Figure 3, assuming each array element has a size of 4 bytes.

The DBM analysis is applied to the whole program. However, not every register is covered in the DBMs but only those that are used to compute the addresses of memory accesses. This set is identified for each loop independently.

$$\begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & 4 & 8 & 12 \\ \infty & -4 & 0 & 4 & 8 \\ \infty & -8 & -4 & 0 & 4 \\ \infty & -12 & -8 & -4 & 0 \end{pmatrix}$$

Figure 3 DBM for the array accesses in the loop in listing 3. The address of each access is exactly four bytes greater than the address of the previous access.

4.4 Hit Classification

We have now all information available to compute the *same block* relation $\overset{\text{sb}}{\sim}$, which in turn is used to predict cache hits.

Three data structures are needed for the classification: the abstract cache set $\hat{\mathcal{S}}$, the map $\hat{\mathcal{V}}$ containing the relative positions of memory accesses inside a cache line and the map $\hat{\mathcal{R}}$ containing the distances between memory accesses.

The abstract cache set $\hat{\mathcal{S}}$ is build as an array containing sets of reference memory accesses. A reference memory access is an access which forces a particular line to be loaded into the cache.[1] To identify such a reference memory access, symbolic names are used, e.g. the instruction which induced the access.

The map $\hat{\mathcal{V}}$ contains for each memory access the alignment information pairs from section 4.1.

The map $\hat{\mathcal{R}}$ contains for each memory access a set of tuples, where each tuple consists of another memory access and the minimal and maximal distance of this memory access to the one used as index. If the DBM analysis could not derive such information, the set is empty.

Additionally, the width of a memory access is given as w_i, as it depends only on the instruction inducing the memory access.

A reference memory access i_{ref} and an ongoing memory access i are *same block*-related if one can show that the addresses of the ongoing and the reference access target the same cache line. For this, adding the access width w_i and the maximum distance x_{max} to the relative position of the reference access inside the cache line must not exceed the length of the cache line. The same check has to be done for the lower cache line boundary, too. Formally speaking:

$$\begin{aligned} \overset{\text{sb}}{\sim} \overset{\text{def}}{=} \{\langle i, i_{ref}\rangle \mid\ & \exists\, x_{min}, x_{max}, y, z : \langle i_{ref}, x_{min}, x_{max}\rangle \in \hat{\mathcal{R}}[i] \wedge \langle y, z\rangle = \hat{\mathcal{V}}[i_{ref}] \\ & \wedge (y \bmod 32) + x_{max} + w_{instr} \leq \min(z, 32) \wedge (y \bmod 32) + x_{min} \geq 0\} \end{aligned}$$

The information $\langle y, z\rangle$ means that the address a_{ref} of the reference access satisfies $a_{ref} \equiv y \pmod z$. The remainder is then taken modulo the cache line width as we are only interested in the relative position inside the cache line. The modulus z is taken as basis of comparison if it is smaller than the cache line width.

[1] In the assumed cache setting, each memory access is also a reference memory access, as both reads and writes load a line when a cache miss happens. When a cache hit happens, the line is already loaded, thus the access is again a reference memory access.

Listing 4 Array summation in PowerPC assembler.

```
.INIT:
    lis r9, 0x18880000@h          // load base address of arr ...
    addi r9, r9, -736             // ... into r9
    lis r8, 0x18880000@h          // load first address after arr ...
    addi r8, r8, -224             // ... into r8
.LOOP:
    lwzx r0, +0(r9)               // load arr[i] into r0
    add r3, r3, r0                // add arr[i] to sum (r3)
    addi r9, r9, +4               // compute address of arr[i+1]

    lwzx r0, +0(r9)               // same for arr[i+1] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    lwzx r0, +0(r9)               // same for arr[i+2] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    lwzx r0, +0(r9)               // same for arr[i+3] ...
    add r3, r3, r0                // ...
    addi r9, r9, +4               // ... due to loop unrolling

    cmp cr0, 0, r9, r8            // check whether we are still ...
    blt cr0, 0x18002c8.t <LOOP> // ... in the bounds of arr
```

If one of the entries of the abstract cache set $\hat{\mathcal{S}}$ and the ongoing memory access are *same block*-related, then the ongoing access is classified as a hit. Otherwise, the access is not classified.

$$\text{hit}(i, \hat{\mathcal{S}}) \stackrel{\text{def}}{=} \exists j \in \{1, 2\} : \exists\, i_{ref} \in \hat{\mathcal{S}}[j] : i \overset{\text{sb}}{\sim} i_{ref}$$

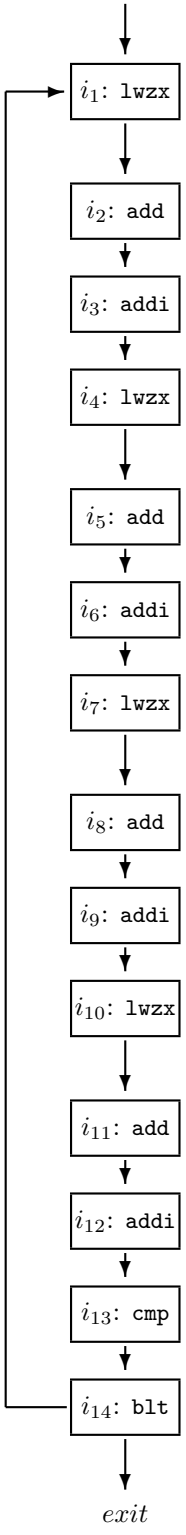
5 Example

To show the relational must analysis in action, we recall the little example used to show the shortcomings of the classical must analysis. The little snippet of C code in listing 3 is compiled to the PowerPC assembler code in listing 4. Note that loop unrolling has been applied.

The control-flow graph of the loop body together with the alignment information and the distance relations is given in Figure 4. The `lwzx` instructions are the only ones which accesses the data memory. The access width of them is four bytes. To enhance readability, only the ingoing and outgoing abstract cache sets of the `lwzx` instructions are shown, because all other instructions do not change the abstract cache sets.

The analysis starts with an empty abstract cache set $\hat{\mathcal{S}}$. Since no distance relations exist for i_1, no cache hit can be predicted. Thus every entry in $\hat{\mathcal{S}}$ ages by one and i_1 is added to the youngest one.

For i_4, there exists a distance relation: the distance between the memory access of i_4 and i_1 is four bytes. To check whether the access of i_4 is a hit, we have to check whether i_4 and i_1 are *same block*-related. To do so, we evaluate the constraint of relation $\overset{\text{sb}}{\sim}$,



	$\hat{\mathcal{S}}$	$\hat{\mathcal{V}}$	$\hat{\mathcal{R}}$	hit?
	$[\{\},\{\}]$			
i_1		$\langle 8,16\rangle$	$\{\}$	no
	$[\{i_1\},\{\}]$			
i_4		$\langle 12,16\rangle$	$\{\langle i_1,4,4\rangle\}$	yes
	$[\{i_4,i_1\},\{\}]$			
i_7		$\langle 0,16\rangle$	$\{\langle i_4,4,4\rangle,\langle i_1,8,8\rangle\}$	no
	$[\{i_7\},\{i_4,i_1\}]$			
i_{10}		$\langle 4,16\rangle$	$\{\langle i_7,4,4\rangle,\langle i_4,8,8\rangle,\langle i_1,12,12\rangle\}$	yes
	$[\{i_{10},i_7\},\{i_4,i_1\}]$			
i_{14}				
	$[\{i_{10},i_7\},\{i_4,i_1\}]$			

Figure 4 Example for the relational must cache analysis.

namely $(y \bmod 32) + x_{max} + w \leq \min(z, 32) \wedge (y \bmod 32) + x_{min} \geq 0$. Instantiation gives $8 + 4 + 4 \leq 16 \wedge 8 + 4 \geq 0$. Thus the access of i_4 is a predicted cache hit. Therefore no entry ages, but i_4 is inserted into the youngest one.

For i_7 there exist also some distance relations. Instantiation of the constraint of relation $\overset{\text{sb}}{\sim}$ gives $12 + 4 + 4 \leq 16 \wedge 12 + 4 \geq 0$ and $8 + 8 + 4 \leq 16 \wedge 8 + 8 \geq 0$. Both evaluate to false. Therefore, no hit can be predicted for i_7. All entries age by one and i_7 is added to the youngest one.

At i_{10}, the memory access is *same block*-related to i_7. Thus the access is a predicted cache hit. Again, no entry ages. The youngest entry will consist of i_{10} and i_7 after the update. The other entry stays the same.

Then the second round of the fixed point iteration starts. This time, we must first join the two incoming abstract cache sets at i_1. Set intersection plus maximal age gives $\hat{\mathcal{S}} = [\{\}, \{\}]$. This is the same value as the input in the first round, thus the fixed point iteration stabilizes directly.

6 Precision

In the example above, two of four accesses are classified as cache hits. This is far below the theoretical maximum of $\frac{7}{8}$.

There are two reasons for that. On the one hand, no relations existed for i_1. This is due to the join of flow from the entry and the recursive loop edge. One possibility would be to add some kind of partitioning to keep the values apart.

The other possibility is to shift the first loop iteration with loop peeling such that the first access in the unrolled loop coincides with the one in which the first entry of a cache line is accessed. Then, the access in which naturally no cache hit is predictable and the one in which not enough information is available coincide.

The other reason is the length of the unrolled loop. Only half of a cache line is covered by one iteration of the loop. If the unroll factor is high enough, the congruence information would cover the whole cache line.

Thus, loop peeling and loop unrolling should be applied in such a way that (a) the peeled prefix is long enough to shift the start of the unrolled loop to match the cache line boundaries and (b) the loop is unrolled enough to cover whole cache lines. The former is usually hard to achieve without a preceding data-flow analysis to determine the right choice. The latter can easily be computed: the unroll factor must be a multiple of the cache line size divided by the access width.

For the example, this means a peeled prefix of three array accesses and eight array accesses inside the unrolled loop. Then, the number of predicted cache hits equals the theoretical maximum of $\frac{7}{8}$.

7 Correctness

The following section sketches the proof of correctness of the relational must analysis with *same block* relation $\overset{\text{sb}}{\sim}$. We assume that the used data-flow analyses and the relational cache analysis framework are correct.

▶ **Lemma 1** (Soundness of $\hat{\mathcal{V}}$, $\hat{\mathcal{R}}$). *$\hat{\mathcal{V}}$ contains only sound abstractions of the alignment of memory accesses. $\hat{\mathcal{R}}$ contains only sound abstractions of the linear relations between two memory accesses.*

Proof. Both statements follow from the correctness of the underlying data-flow analyses. ◀

▶ **Lemma 2** (Soundness of $\hat{\mathcal{S}}$). *For any instruction i in a given program, $\hat{\mathcal{S}}$ contains only reference memory accesses for cache lines that are in the cache at the point of execution of i.*

Proof. Follows from the correctness of the relational cache analysis framework. ◀

▶ **Lemma 3** (Soundness of $\overset{\text{sb}}{\sim}$). *Two instruction i, i_{ref} are* same block*-related only if the induced memory accesses of both instructions target the same cache line.*

Proof. The proof is carried out by showing that if two accesses target different cache lines, then they are not *same block*-related.

Let c be the size of the cache, b the size of one cache line. Let instruction i induce a memory access to address a with width $w > 0$. Let instruction i_{ref} induce a memory access to address a_{ref}. Let $a - a_{ref} = x$.

Assume furthermore that $\langle y, z\rangle \in \hat{\mathcal{V}}[i_{ref}]$ and $\langle i_{ref}, x_{min}, x_{max}\rangle \in \hat{\mathcal{R}}[i]$. The last two assumptions are safe as otherwise the constraint of $\overset{\text{sb}}{\sim}$ evaluates to false. From lemma 1, it follows that $y = a_{ref} \bmod z$ and $x_{min} \leq x \leq x_{max}$.

- Let a go into a cache line after a_{ref}, that is, $a \bmod c > a_{ref} \bmod c$. Then $(a_{ref} \bmod b)+x \geq b$ because the offsets of the original cache line lie in the interval $[0, b-1]$ and $(a_{ref} \bmod b)+x$ points to a later cache line.
  Two cases must be distinguished: (1) $b \leq z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{max} \geq b$. From $w > 0$, it follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$. Moreover, $k' \cdot z + y + x \geq k \cdot z$ holds with $k = \frac{b}{z} > 1$ and $0 \leq k' = \frac{(a_{ref} \bmod b)-y}{z} < k$. Thus $y + x \geq (k - k') \cdot z$ and $k - k' \geq 1$. Thus $(y \bmod b) + x_{max} \geq z$. From $w > 0$, it follows directly that the constraint evaluates to false.
- Let a go into the same cache line as a_{ref}, $a \bmod b > a_{ref} \bmod b$ and w such that the access crosses the cache line boundary. Then $(a_{ref} \bmod b) + x + w > b$.
  Two cases must be distinguished: (1) $b \leq z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{max} + w > b$. It follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$. Moreover, $k' \cdot z + y + x + w > k \cdot z$ holds with $k = \frac{b}{z} > 1$ and $0 \leq k' = \frac{(a_{ref} \bmod b)-y}{z} < k$. Thus $y + x + w > (k - k') \cdot z$ and $k - k' \geq 1$. Thus $(y \bmod b) + x_{max} + w > z$. It follows directly that the constraint evaluates to false.
- Let ma go into a cache line before a_{ref}, that is, $a \bmod c < a_{ref} \bmod c$. Then $(a_{ref} \bmod b)+ x < 0$ because the offsets of the original cache line lie in the interval $[0, b-1]$ and $(a_{ref} \bmod b) + x$ points to an earlier cache line.
  Two cases must be distinguished: (1) $b \leq z$ and (2) $b > z$.
  If (1) holds, then $y \bmod b = a_{ref} \bmod b$. Thus $(y \bmod b) + x_{min} < 0$. It follows directly that the constraint evaluates to false.
  If (2) holds, then $y \bmod b = y$ and $y \leq (a_{ref} \bmod b)$. Thus $(y \bmod b) + x_{min} < 0$. It follows directly that the constraint evaluates to false.

◀

▶ **Theorem 4** (Soundness of the relational must cache analysis with *same block* relation $\overset{\text{sb}}{\sim}$). *For any instruction in a given program, a cache hit is only predicted if it would happen in every concrete run of the program.*

Proof. From lemma 2, it follows that at any instruction, $\hat{\mathcal{S}}$ contains only reference memory accesses for cache lines that are currently in the cache. A hit is only predicted, if the given instruction is *same block*-related to a reference memory access in $\hat{\mathcal{S}}$. From lemma 3 follows that two memory accesses are only *same block*-related if they target the same cache line. Thus a hit is only predicted if a memory access targets a cache line that is currently in the cache. ◀

8 Experimental Results

A prototype implementation of the relational cache analysis with *same block* relation $\overset{\text{sb}}{\sim}$ has been integrated into the `aiT` WCET analyzer [1]. With this prototype, the Mälardalen WCET Benchmark [8] has been analyzed. The results of both the classical cache analysis and the relational cache analysis are shown in table 1. For the classical cache analysis, the loops have been 1-fold peeled. For the relational cache analysis, the loops have been 1-fold peeled and 8-fold unrolled.

One interesting property is the data cache hit prediction ratio on the critical path. For most tests, the prediction ratio increased as expected.

A closer look has been taken for those tests where the relational cache analysis did not show the desired improvements. Three tests performed particularly bad: `insertsort`, `fir` and `cover`. They all have in common that no relations could be computed. This is because of the strange control flow generated by the compiler. In `insertsort` for example, two loops are merged into one. On one join point, the relation r7 = r0 comes from one edge and the relation r7 = r5 from another. As r0 and r5 have different values, no relation remains after the join. A similar thing happens in `fir`. In `cover`, some of the loops are split into two nested ones.

Another programming pattern that is bad for relational cache analysis is unveiled in `crc`. Here, the computation depends heavily on bit operations. These destroyed the relational information since the DBMs only handle linear relations.

One particular interesting test is `lcdnum`. The results of this test show a much higher data cache prediction ratio for the classical cache analysis than for the relational one. This seems counterintuitive at first glance, but there is no error. The reasons for this behaviour are the paths which could be proven infeasible due to the loop unrolling. On these paths are lots of loads which induce cache hits. Thus the data cache prediction ratio is higher than for the relational cache analysis.

Another point of interest is the performance of the relational cache analysis. For most tests, the additional runtime for performing the relational value analysis with difference bound matrices is only a few seconds. The biggest exception is `lms`, where the runtime increased from 3 to 127 seconds. However, the data cache prediction ratio increased from 56% to 78% and the WCET bound decreased by about a quarter. Thus the decreased performance is justified by the increased precision.

9 Related Work

Both Grewe [7] and Flexeder [5] propose to use alignment information to improve the prediction of cache hits. However, they do not work out the details of their ideas.

Hahn and Grund [9] use global value numbering [2] in combination with interval analysis to compute the *same block* relation. This technique, however, is not powerful enough to relate the changing addresses of the array accesses inside the loop.

Table 1 Results for the Mälardalen WCET benchmark. For both the relational cache analysis and the classical cache analysis, it is given the computed WCET bound in cycles, the predicted data cache hit ratio on the critical path and the analysis time in seconds.

	Relational			Classical		
program	WCET	ratio	time	WCET	ratio	time
adcpm	407360	.68	6	569195	.21	4
bs	1230	.08	1	1234	.08	1
bsort100	1827484	.86	14	3205665	.00	2
cnt	20007	.91	5	28668	.51	1
compress	4254976	.30	17	4281933	.30	2
cover	31196	.04	6	43102	.00	8
crc	175060	.17	3	228931	.01	2
duff	13039	.70	2	18425	.00	1
edn	541462	.42	12	836568	.04	3
expint	9999	.72	2	59291	.72	1
fac	824	.33	1	898	.33	1
fdct	11192	.93	2	22740	.15	1
fft1	57139	.93	14	66627	.87	5
fibcall	715	.00	1	715	.00	1
fir	32102	.04	6	36381	.01	1
insertsort	17887	.00	1	17887	.00	1
janne_complex[a]	12633	—	2	12633	—	1
jfdctint	17021	.91	2	32266	.08	1
lcdnum	2339	.57	1	5658	.78	1
lms	3575657	.78	127	4599643	.56	3
ludcmp	13129	.93	6	495695	.01	2
matmult	1570041	.43	44	2225727	.04	2
minver	13283	.70	2	36194	.10	3
ndes	644343	.57	17	759276	.54	3
ns	9744	.86	6	84442	.00	1
nsichneu	292027	.85	14	292027	.85	11
prime	27515	.76	2	27518	.76	1
qsort-exam	497534	.01	34	644161	.00	2
qurt	21101	.73	3	24002	.69	1
recursion	10972	.37	5	12308	.05	1
select	104532	.03	7	111912	.00	1
sqrt	9140	.50	2	20446	.50	1
st[b]	—	—	—	—	—	—
statemate	138338	.97	10	146960	.96	7
ud	9850	.93	3	104877	.01	2

[a] This program contains no loads.
[b] This program could not be compiled.

Lundqvist and Stenström [12] proposed a method to classify data structures as predictable or unpredictable. Unpredictable data structures should then be moved into some uncached memory areas to prevent the eviction of cache entries which buffer parts of the predictable data structures to reduce the overestimation caused by imprecise address information.

Vera and Xue [14] use cache miss equations [6]. They try to relate memory accesses, too, but instead of using an abstract interpretation based data-flow analysis, they compute linear equation systems to count the number of accesses between the reference and the ongoing access. If at least k such accesses exist, they assume a sure miss. Their analysis is restricted to precise address information.

Li, Malik and Wolfe [10, 11] expressed the cache behaviour through integer linear programs. In case of imprecise address information, they add a constraint to the ILP that only one of the addresses is taken. This allows the ILP solver to choose the worst case. This may lead to huge ILPs with unacceptable solving times. Moreover, they do not express mutually exclusive cache misses.

10 Conclusion and Future Work

This paper presented a method to compute *same block* relations for relational cache analysis. The relational cache analysis is a novel cache analysis which is able to predict cache hits even if precise address information is unavailable. It relates memory accesses and checks whether they go to the same cache line. Thus it forms a substantial improvement over Ferdinand's cache analysis which needs precise address information for its predictions.

Using the analyses presented in the work at hand, a *same block* relation $\overset{\text{sb}}{\sim}$ is computed which – in contrast to the one in [9] – is powerful enough to relate memory accesses with changing addresses, for example if array accesses happen inside a loop.

First experiments on the Mälardalen WCET Benchmark [8] show that the presented *same block* relation works well. In the mean, about 52% of the memory accesses have been predicted as cache hits by the relational cache analysis. The classical cache analysis predicts only about 27% cache hits if the loops are not unrolled, and about 64% cache hits if the loops have been fully unrolled to compute globally precise address information.

While loop peeling and loop unrolling are needed to increase the precision of $\overset{\text{sb}}{\sim}$, loop fusion and loop fission may hamper the precision as the experimental results show.

Further improvements are possible: At the moment, only cache hits are predicted by the relational cache analysis, i.e. it is a must analysis. Finding the right congruence analyses to support precise relational may analysis is still an open research question. Moreover, using only $\overset{\text{sb}}{\sim}$ as *same block* relation, the relational cache analysis cannot distinguish between the different cache sets, and all collapse into one, effectively decreasing the achievable precision. To overcome this obstacle, *same set*, *different set* relations need to be computed, too. Both will be targeted in future work.

Acknowledgements This work is based on the author's master's thesis [15], written under the supervision of Prof. Dr. Dr. h.c. Reinhard Wilhelm and Dr. Florian Martin. The author likes to thank Dr. Reinhold Heckmann, Dr. Daniel Grund and the anonymous reviewers for their valuable comments and suggestions.

References

1 AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. `http://www.absint.com/ait/`.

2 Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11. ACM, 1988.

3 David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, London, UK, 1990. Springer-Verlag.

4 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

5 Andrea Flexeder. *Interprocedural Analysis of Low-Level Code*. PhD thesis, Technische Universität München, 2011.

6 Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, July 1999.

7 Dominik Grewe. Static Congruence Analysis on Binaries. Bachelor's thesis, Saarland University, 2008.

8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

9 Sebastian Hahn and Daniel Grund. Relational Cache Analysis for Static Timing Analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, Los Alamitos, CA, USA, July 2012. IEEE.

10 Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, page 298, Washington, DC, USA, 1995. IEEE Computer Society.

11 Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, page 254, Washington, DC, USA, 1996. IEEE Computer Society.

12 Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, page 255, Washington, DC, USA, 1999. IEEE Computer Society.

13 Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 155–172, London, UK, 2001. Springer-Verlag.

14 Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA '02)*, pages 175–186, 2002.

15 Simon Wegener. Improving Static Analysis of Loops. Master's thesis, Saarland University, 2011.