Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor Thesis

Towards Relational Cache Analysis

submitted by
Sebastian Hahn

submitted
November 2011

Supervisor
Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm

Advisor
Daniel Grund

Reviewers
Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm
Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Programs in hard real-time systems have to satisfy non-functional constraints, such as timing constraints, beyond their functional requirements. The static analysis that derives bounds on the execution times has to take into account the components of the underlying hardware platform, such as caches and pipelines, in order to derive tight bounds on the execution times.

Static cache analysis classifies memory accesses as cache hits, cache misses or unknown. State-of-the-art cache analyses cannot precisely model accesses whose addresses are imprecisely determined: no hits can be predicted for those accesses and they deteriorate analysis information excessively.

We present a relational cache analysis that can represent cache elements with imprecisely determined addresses and uses relations between cache elements — obtained by congruence analyses — in order to do precise updates and classifications. This paves the way for cache analysis to profit from analyses other than value analyses and enables the prediction of hits for new classes of accesses, e.g. those with input-dependent addresses.

Acknowledgments

First of all, thanks to my supervisor Professor Reinhard Wilhelm who has given me the opportunity to do research on this nice topic in his group.

My special thanks go to Daniel Grund who inspired the idea of this thesis. Thank you for the thousands of valuable discussions and the support throughout all phases of researching and writing: I learned a lot from you about static program analysis, abstract interpretation, cache analysis, and, last but not least, many new LaTeX commands.

I also want to thank Professor Sebastian Hack for his feedback and ideas as well as his willingness to review this thesis. Further thanks go to

(a) Thomasz Dudziak for interesting discussions as well as inspirations on further applications of the relational cache analysis;

(b) the people of the compiler design lab for their other interesting thesis proposals and their feedback, especially to Jörg Herter and Sebastian Altmeyer;

(c) Christoph Mallon for his support during the implementation and especially for his invaluable knowledge about libFirm; and also to

(d) Florian Haupenthal and Barbara Dörr for reading and correcting parts of the thesis, lively conversations and their support throughout my studies.

Last but not least, I would like to express my special gratitude to my family: my parents Thomas Hahn and Ursula Klemann-Hahn as well as my grandparents Werner and Anni Klemann for their love and support throughout my life and thanks to my sister Bernadette Hahn, especially for her continuous help with my studies and with mathematics.

Contents

1 Introduction

In hard real-time systems, the set of tasks that are supposed to run on a microprocessor has to undergo a schedulability analysis. A task works correct if it computes correct results and satisfies timing constraints. The schedulability analysis checks statically whether a given set of tasks is schedulable in a way that each task always satisfies its timing constraints. As input, the schedulability analysis needs the running times of the given programs. The running time of a program can vary due to different inputs and different initial hardware states. Thus, the schedulability analysis has to account for all possible running times and especially the running time in the worst case. Measured running times might dramatically differ from the actual running time in the worst case. In this case, the schedulability analysis could classify a set of tasks as schedulable although it is not. Computing the worst-case execution time statically is not always possible, e.g. as the running time might depend on inputs not known before runtime. Therefore, it is only safe to use upper bounds on the execution times.

The analysis that computes an upper bound on the execution times of a program is called Worst Case Execution Time (WCET) analysis. The analysis is divided in several phases that are shown in Figure 1. We will give a short overview and then focus on cache analysis.

The first phase reconstructs the control-flow graph (CFG) of the given program binary. The CFG is used as the intermediate representation subsequent analyses are performed on.

Value analysis computes over-approximations on the set of values a register or a memory location can hold. An interval analysis can be used for this purpose.

Loop bound analysis tries to determine an upper bound on the number of iterations of each loop. If the loop bound analysis fails to do so, the WCET analysis requires additional information on the loop bounds. This information has then to be provided by the user.

Control-flow analysis tries to detect infeasible paths through the program that can later be excluded from the final WCET computation.

Micro-architectural analysis determines worst-case execution times of basic blocks. Modern microprocessors have several components that speed up computation, e.g. pipelines and caches. Although these components are important for acceptable execution times, they complicate WCET analysis. Not taking the underlying hardware platform into account would lead to very loose approximations of execution times that are virtually useless for schedulability analysis.

Finally, global bound analysis determines the longest path through the program with the help of the results of the micro-architectural analysis and the control-flow analysis.

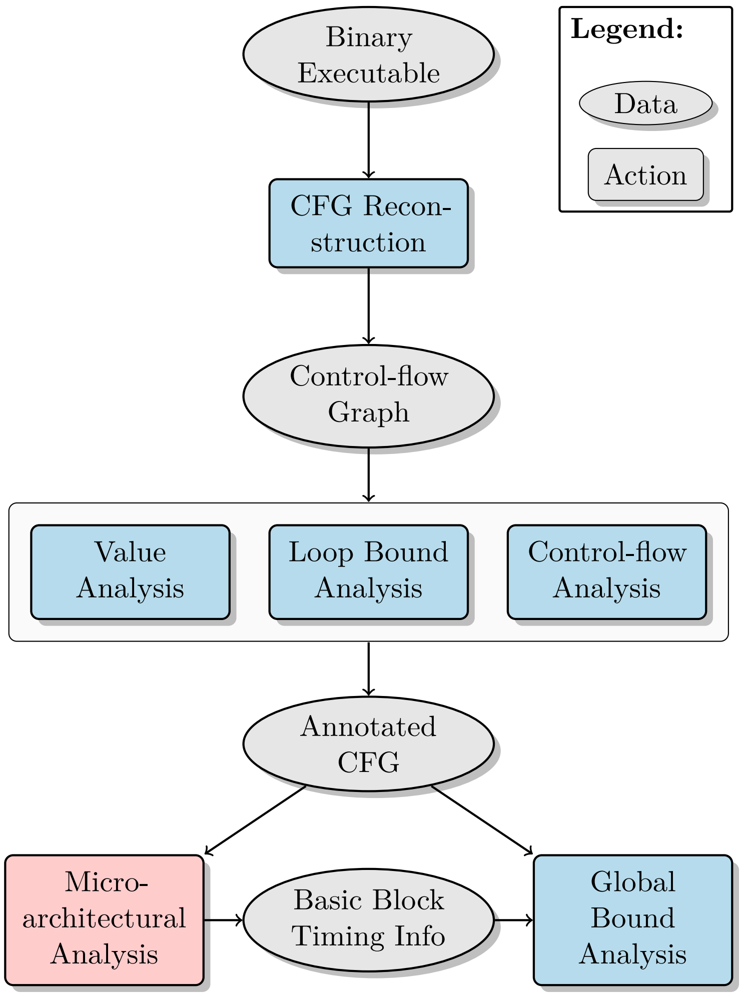


Figure 1: Tool architecture for WCET analysis

Cache analysis In the following, we focus on cache analysis, which is a crucial part of the micro-architectural analysis. A cache is a small and fast memory that aims at bridging the latency gap between the processor and main memory. The cache therefor stores memory blocks, consisting of several adjacent memory words, that are likely to be accessed in the near future. If a memory block is accessed that resides in the cache, the access is called a cache hit, otherwise a cache miss. Ignoring the influence of the cache on the execution time leads to high overestimations: whereas a memory access can be serviced in one cycle in case of a cache hit, loading a cache line from main memory in case of a cache miss takes up to several 100 cycles. Cache analysis tries to classify memory accesses that occur in the program either as hit, miss or unknown. If the analysis classifies an access as hit (miss), the memory access is guaranteed to hit (miss) the cache. Otherwise, the cache analysis could neither guarantee that the access hits the cache nor that it misses the caches.

In cache analysis, there are several sources of uncertainty [15] (also see Figure 2) that influence the precision of the analysis. Usually, one cannot make assumptions on the initial cache contents as they depend on previously executed tasks. Different initial cache contents can result in different execution times. When control flow joins, one has to safely combine information that comes from different paths, which typically leads to imprecision in cache analysis. Holding information from different paths apart can yield preciser results but is inefficient.

In order to produce precise results, currently employed cache analyses depend on precisely computed addresses: The cache analysis makes use of the previous value analysis that has approximated the addresses of memory blocks that might be accessed. However, it is not always possible to obtain the precise address that is accessed, e.g. if the program involves input-dependent addresses. If a function involves stack-relative accesses and is called several times during program execution, the value analysis obtains imprecisely determined addresses as the stack-pointer changes. Similarly, addresses of array accesses with loop-dependent indices can only be determined imprecisely. In the latter cases, current approaches increase context sensitivity to be able to compute precise addresses within each context.

Our Contributions We present a novel cache analysis, that alleviates the dependency on address analysis. Basically, it does away with the popular misconception that precise address information is necessary for cache analysis. To predict cache hits for instance, it is sufficient to show that the currently accessed block coincides with a cached one, possibly without knowing its actual address.

As our first contribution, we introduce the concept of symbolic names as abstract cache elements. A symbolic name is a unique identifier for a static memory reference and can thus be seen as a representative for all memory blocks that might be

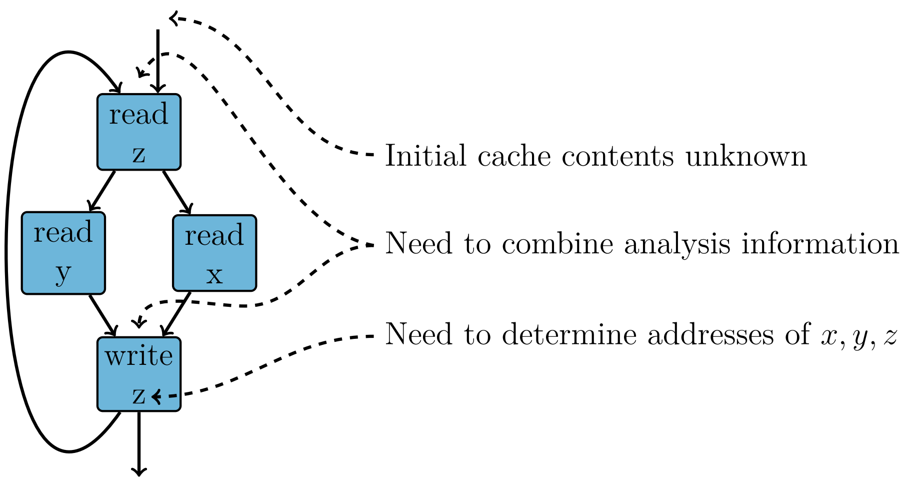

read
z
read
y
read
x
write
z
Initial cache contents unknown
Need to combine analysis information
Need to determine addresses of x, y, z

Figure 2: Sources of uncertainty in cache analysis

accessed by that reference. It abstracts from specific addresses and therefore allows the modeling of cache elements whose addresses are imprecisely determined.

To retain information about a symbolic name in an abstract cache, we consider relations between cache sets. We do not analyse cache sets independently anymore but in combination: a cached symbolic name is guaranteed to be cached while it might map to any cache set and the cache set it actually maps to is unknown.

To do precise updates and classifications, we use relations between symbolic names. Therefore, we have a module called congruence analysis, that computes relations between symbolic names, e.g. that the currently accessed symbolic name coincides with the one accessed two instructions ago. This paves the way for cache analysis to use information from analyses other than address analysis.

There is a wide range of application examples, e.g. programs that involve input-dependent accesses, or stack-relative accesses with an imprecisely determined stack pointer.

We implemented a prototype of the relational cache analysis using the intermediate representation Firm. The first results obtained by this implementation are promising. When analyzing stack-relative accesses without distinguishing call sites, the relational cache analysis can still classify up to 56% of the static memory references as hits — or up to 83% of the hits that can be predicted by the analysis with full context sensitivity. If increasing the context sensitivity is not feasible, e.g. in deeply nested loops, the relational cache analysis can nevertheless find out a reasonable amount of hits whereas the traditional cache analysis cannot predict any hits. Up to 14% more hits can be predicted by the relational cache analysis compared to the traditional one, if accesses with input-dependent addresses are involved. These results are obtained by using simple congruence analyses like an interval analysis or global value numbering — employing more powerful congruence analyses will lead to an improvement on the relational cache analysis.

Structure of the Thesis

Section 2 presents foundations like program representation and abstract interpretation. Section 2.4 especially treats the cache analysis of Ferdinand et al. [7]. We are particularly interested in how accesses to sets of memory blocks instead of only one memory block are handled. This is necessary for accesses whose addresses cannot be precisely computed.

In Section 3 we first motivate our work in more detail on the basis of an example. We show how to improve the traditional cache analysis in the presence of accesses with imprecisely determined addresses. We formally define the concept of symbolic names and the congruence analysis that computes relations between symbolic names. Subsequently, we present a relational cache analysis that makes use of symbolic names and congruence analysis results. The underlying concrete semantics is also defined and the analysis is proven correct with respect to this concrete semantics. Furthermore, we revisit the motivating example to demonstrate the results of our relational cache analysis.

In Section 4 we discuss work related to (data) cache analysis and compare these approaches to our relational cache analysis.

Section 5 deals with the implementation details, i.e. we present the employed congruence analyses and the framework we used as an intermediate program representation. In Section 6 we identify classes of programs for which our analysis is profitable and compare its results to the ones of the traditional cache analysis.

We conclude in Section 7 with a short summary and outlook on future work.

2 Foundations

2.1 Program Representation

Definition 2.1 (Rooted Graph)**.** A *rooted graph* G is a triple (V, E, r), where V is the vertex set and $E \subseteq V \times V$ is the set of arcs. The distinguished root vertex $r \in V$ has no incoming edges, i.e. $\forall v \in V : (v, r) \notin E$.

Definition 2.2 (Path)**.** Let $G = (V, E, r)$ be a rooted graph and $u, v, w \in V$. A *path* π from v to u is a non-empty sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $v_0 = u, v_k = v$, and $\forall i \in \{1, \ldots, k\} : (v_{i-1}, v_i) \in E$. For short, we write $\pi = u \rightarrow^* v$.

Definition 2.3 (Program)**.** Let A denote the set of *variables* and be $\mathtt{r_1}, \ldots, \mathtt{r_i}$, $\mathtt{o_1}, \ldots, \mathtt{o_j} \in A$. An *instruction* is an assignment of the form $(\mathtt{r_1}, \ldots, \mathtt{r_i}) := \rho(\mathtt{o_1}, \ldots, \mathtt{o_j})$ with operation ρ, operands $\mathtt{o_1}, \ldots \mathtt{o_j}$, and results $\mathtt{r_1}, \ldots, \mathtt{r_i}$. A *program* P is a control-flow graph (CFG), $G = (V, E, r, I)$, where each vertex $v \in V$ has associated exactly one instruction, $I(v)$. E models the potential control flow, r represents the unique program entry point.

2.2 Abstract Interpretation

Many problems in static analyses are undecidable or the results of analyses are not efficiently computable. In order to statically obtain information about a program (in an efficient manner), one computes *approximations*. The theory of *Abstract Interpretation* [4] formalises this idea of approximation and allows for the design of semantics-based program analyses as well as for the proofs of their correctness. Different types of program semantics and abstractions are presented in [3] as well as their use for static program analysis.

Section 2.2.1 covers basic program semantics that occur throughout the thesis such as trace semantics, collecting semantics and abstract semantics. In Section 2.2.2 we show how soundness of abstractions can be established.

2.2.1 Program semantics

A program semantics describes the meaning of a specific program. The set of program semantics is a partially ordered set $(P, \sqsubseteq)$. The partial order models the notion of precision. Let p_1 and p_2 be two program semantics. If $p_1 \sqsubseteq p_2$, we say p_1 is at least as precise as p_2, i.e. p_1 describes the meaning of the program at least as precise as p_2. We will present some kinds of semantics that occur similarly in this thesis.

Trace semantics Let Σ be a set of system states and $IC \subseteq \Sigma$ the set of initial system states. The underlying partially ordered set for the trace semantics is $(2^{(V\times\Sigma)^*}, \subseteq)$. Besides the sequence of system states during execution, we also explicitly model the corresponding program point during execution, that can also be part of Σ. Consider a transition relation $\sim : (V \times \Sigma) \times (V \times \Sigma)$ that models an elementary step of the whole system. The trace semantics is defined in terms of a fixpoint semantics.

Definition 2.4 (Trace Semantics)**.** Let P be a program given by the CFG $G = (V, E, r, I)$. The trace semantics $\mathit{Trace}_P \in 2^{(V\times\Sigma)^*}$ of P on the initial system configurations IC is defined as the least fixpoint of the following equation.

$$\mathit{Trace}_P = \{(r, s) \mid s \in IC\} \cup \{\pi \circ x \circ y \mid \pi \circ x \in \mathit{Trace}_P \wedge x \sim y\}$$

Sticky Collecting Semantics Let S be a set of possible descriptions of program properties we are interested in and $f : V \to S \to S$ the corresponding transformer function. The underlying partially ordered set is $(V \to 2^S, \sqsubseteq)$. $\sqsubseteq$ is defined component-wise:

$$s_1 \sqsubseteq s_2 := \forall v \in V. s_1(v) \subseteq s_2(v)$$

The collecting transformer $f_{coll} : V \to 2^S \to 2^S$ then looks like $f_{coll}(v)(S) = \{f(v)(s) \mid s \in S\}$. The corresponding path effect $[\![\pi]\!]_{f_{coll}}$ is defined as

$$[\![\pi]\!]_{f_{coll}} = \begin{cases} id & : \pi = \epsilon \\ f_{coll}(v_n) \circ [\![(v_1, \dots, v_{n-1})]\!]_{f_{coll}} & : \pi = (v_1, \dots, v_n) \end{cases}$$

Definition 2.5 (Sticky Collecting Semantics)**.** Let P be a program given by the CFG $G = (V, E, r, I)$. The sticky collecting semantics $\mathit{Coll}_P : V \to 2^S$ of P on the set of initial state IS is

$$\mathit{Coll}_P := \lambda v \in V. \{[\![\pi]\!]_{f_{coll}} IS \mid \pi : r \to^* v\}$$

Abstract Semantics Let $(A, \sqsubseteq_A)$ be a partially ordered set of abstract descriptions of interesting program properties and $f_{abs} : V \to A \to A$ the corresponding abstract transformer. The underlying partially ordered set is $(V \to A, \sqsubseteq)$ where $\sqsubseteq$ is defined component-wise using $\sqsubseteq_A$. The corresponding path effect $[\![\pi]\!]_{f_{abs}}$ is again defined as

$$[\![\pi]\!]_{f_{abs}} = \begin{cases} id & : \pi = \epsilon \\ f_{abs}(v_n) \circ [\![(v_1, \dots, v_{n-1})]\!]_{f_{abs}} & : \pi = (v_1, \dots, v_n) \end{cases}$$

Definition 2.6 (Sticky Abstract Semantics)**.** Let P be a program given by the CFG $G = (V, E, r, I)$. The sticky abstract semantics $Abs_P : V \to A$ of P on the abstract initial state $AI \subseteq A$ is

$$Abs_P := \lambda v \in V. \bigsqcup \{ [\![\pi]\!]_{f_{abs}} AI \mid \pi : r \to^* v \}$$

2.2.2 Establishing soundness

We will provide some basic information on abstract interpretation and how we use it to finally prove the correctness of our analysis. In the following, we consider a *concrete domain* $(D_C, \leqslant)$ as well as an *abstract domain* $(D_A, \sqsubseteq)$.

Relating the abstract to the concrete domain We need to give meaning to elements of our abstract domain. We therefor define a monotonic function $\gamma : D_A \to D_C$, the so called *concretization function.* $\gamma(a)$ describes the concrete element that is represented by the abstract element a.

The requirement that γ has to be monotonic is quite natural. If $a_1 \in D_A$ is more precise than $a_2 \in D_A$ $(a_1 \sqsubseteq a_2)$, the concrete element $\gamma(a_1)$ should also be more precise than the concrete element $\gamma(a_2)$ $(\gamma(a_1) \leqslant \gamma(a_2))$.

In abstract interpretation there is usually also an *abstraction function* $\alpha : D_C \to D_A$ that maps a concrete element to its abstract description. Since we will not make further use of it, we will not go into detail about it. For further reading, please refer to [4].

Definition 2.7 (Soundness)**.** Let P be a program and $Conc_P \in D_C$ the concrete semantics of P and $Abs_P \in D_A$ the abstract semantics of P. Furthermore let $\gamma : D_A \to D_C$ be the concretization function. Abs_P is a sound approximation of $Conc_P$ if and only if

$$Conc_P \leqslant \gamma(Abs_P)$$

Establishing soundness between sticky semantics Let $D_C = V \to C$, $D_A = V \to A$ and $\leqslant_C$, $\sqsubseteq_A$ the respective partial orders on C and A. The sticky semantics are given by the domain $(C, \leqslant_C)$ and the transformer $f^\natural : V \to C \to C$, respectively $(A, \sqsubseteq_A)$ and $f^\sharp : V \to A \to A$. The partial order can be extended by

$$\begin{aligned} c_1 \leqslant c_2 &= \forall v \in V. c_1(v) \leqslant_C c_2(v) \\ a_1 \sqsubseteq a_2 &= \forall v \in V. a_1(v) \sqsubseteq_A a_2(v) \end{aligned}$$

The concretization function γ between D_A and D_C can be defined program point-wise by the concretization function γ_{AC} between A and C.

$$\gamma(d_A) = \lambda v \in V.\gamma_{AC}(d_A(v))$$

Besides the monotonicity of the concretization function, it is also natural to demand monotonicity for the concrete and the abstract transformer. Inputs that are more precise than others should lead to outputs that are also more precise than others. For soundness, it is sufficient to show that the abstract transformer $f^\sharp$ approximates the concrete one $f^\natural$. This leads us to the definition of local consistency.

Definition 2.8 (Local consistency)**.** Let $a \in A$. The abstract transformer $f^\sharp$ is called *locally consistent* with respect to the concrete transformer $f^\natural$ if and only if

$$\forall v \in V.f^\natural(v)(\gamma_{AC}(a)) \leqslant_C \gamma_{AC}(f^\sharp(v)(a))$$

Finally, we come to the main theorem that establishes soundness.

Theorem 2.9 (Soundness)**.** *Let P be an arbitrary program. If $f^\sharp$ is locally consistent with respect to $f^\natural$ and the abstract initial state $I^\sharp$ approximates the concrete initial state $I^\natural$, the sticky abstract semantics Abs_P (described by $(A, \sqsubseteq_A)$ and $f^\sharp$) is a sound approximation of the sticky concrete semantics $Conc_P$ (described by $(C, \leqslant_C)$ and $f^\natural$).*

$$\forall a \in A\ \forall v \in V.f^\natural(v)(\gamma_{AC}(a)) \leqslant_C \gamma_{AC}(f^\sharp(v)(a)) \wedge I^\natural \leqslant_C \gamma_{AC}(I^\sharp) \\ \Rightarrow Conc_P \leqslant \gamma(Abs_P)$$

2.3 Caches

Modern computer architectures feature processors with few fast registers and large but slow memory. To minimise the overall latency of memory accesses, small and fast memories — so called caches — are employed. Even several levels of caches can be employed, e.g. a small and fast cache at the first level and a larger but slower one at the second level. A general memory hierarchy is shown in Figure 3. Since caches have a major impact on the system performance and thus the execution time of a program on the given hardware, they have to be taken into account during the micro-architectural analysis.

Why do they work? Although caches are smaller compared to the main memory, they perform well in practice due to the *principle of locality.* Temporal locality means that a memory word that has just been accessed is very likely to be accessed

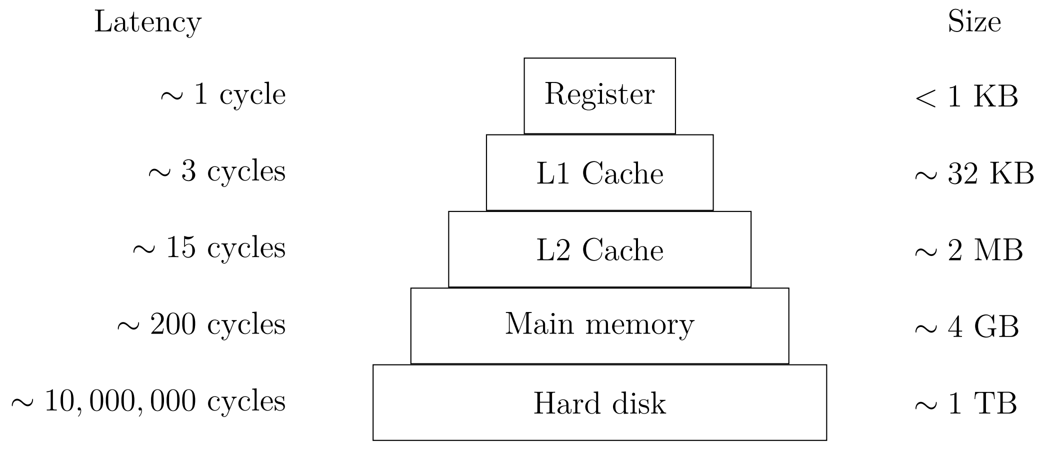


Figure 3: Memory hierarchy in modern computer architectures

in the near future, e.g. instruction fetches for a loop. Spatial locality means that a memory word whose adjacent word has just been accessed is also very likely to be accessed in the near future, e.g. instruction fetches for straight-line code.

The cache now tries to exploit locality. First, it stores memory words that are accessed in order to provide them very fast if they are accessed again. Second, if a word from memory is loaded into the cache, some adjacent memory words are also loaded into the cache within one operation in the hope that they are accessed afterwards.

How do they work? In the following we refer to Figure 4.

The memory is partitioned in equally sized *memory blocks* that exactly fit into a *cache line*. Adjacent memory words form such a memory block. The number of words per block depends on the size of the memory block and thus the size of the cache line. The cache is organised in cache lines.

During a memory access, the cache controller has to look up whether the requested memory block resides in the cache. In case the memory block does, it is called a *cache hit* — otherwise a *cache miss*. In order to do fast look-ups, a memory block can only be stored in a small number of cache lines that are grouped in so-called *cache sets* of size k. The cache is then called k-way associative.

The cache set to which a memory block maps is determined by the *index* bits of the address. The *offset* bits of the address are used to extract the requested memory word from the memory block. The remaining bits form the *tag* of the address. The tag is used to test whether the requested memory block matches a memory block in the cache set.

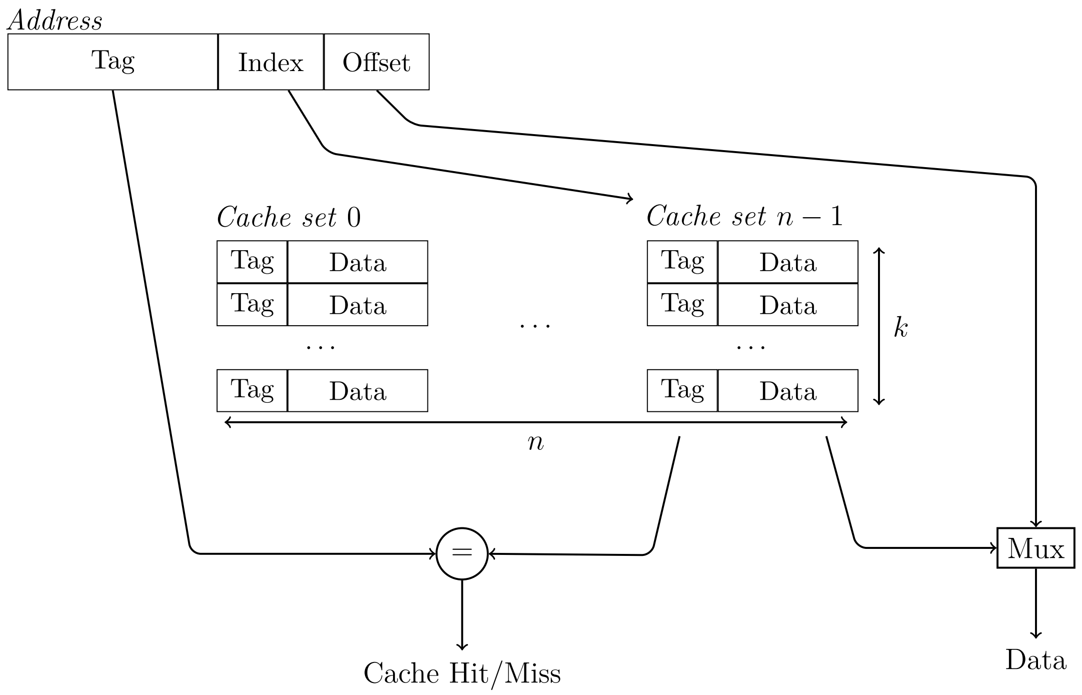


Figure 4: Scheme of a k-way associative cache

In case a cache miss occurs, the memory block has to be loaded from the main memory and to be stored in the cache. As not all memory blocks fit into the cache at the same time, another memory block has to be evicted. The policy that determines which memory block to evict is called *replacement policy*. We will focus on the LRU — the least recently used — replacement policy that has turned out to be the best in terms of predictability [16]. The LRU policy always replaces the memory block that has not been accessed for the longest time.

2.4 Traditional Cache Analysis

In the following, we assume a k-way associative LRU cache with n cache sets. In this section, we describe the abstract cache domain for the must analysis first presented in [7]. Of particular interest are the classification and update formulas which handle cache accesses with imprecisely determined addresses. The analysis is based on the following observation.

Observation 2.10. *A memory block b in cache set i that has just been accessed is evicted after accesses to k other, distinct blocks that map to cache set i. It does not matter whether these accesses are hits or misses.*

There is a notion of age of a memory block. The *age* of a memory block b denotes the number of accesses to pairwise different memory blocks, mapping to the same cache set as b, since the last access to b. We can reformulate the above observation as follows. A memory block b resides in the cache as long as its age is less than the associativity of the cache.

Since we do not always know the exact age of a memory block statically, we bound the age; once from above and once from below. The analysis that tracks upper (lower) bounds on ages of memory blocks is called *must (may) analysis*. If the must analysis can guarantee that a memory block has a maximal age less than k and this block is accessed, the access will result in a cache hit. An access to a memory block will result in a cache miss if the may analysis guarantees that its age is at least k. In the following we will focus on the must analysis presented in [7].

Lattice Let $\mathcal{B}$ denote the set of all memory blocks and $\mathcal{B}_i$ the set of all memory blocks mapping to cache set i. Furthermore, let $\mathcal{AB}^{\leq}$ denote the set of bounds on the age an element in a k-way associative LRU cache can have. It is defined as

$$\mathcal{AB}^{\leq} := \{0, \dots, k-1, \infty\}$$

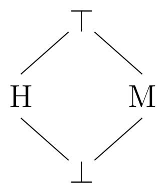
We now define our basic lattice.

$$\begin{aligned}\mathcal{S}_{Fer}^{\leq,i} &:= \mathcal{B}_i \to \mathcal{AB}^{\leq}\\ \mathcal{C}_{Fer}^{\leq} &:= \prod_{i=0}^{n-1} \mathcal{S}_{Fer}^{\leq,i}\end{aligned}$$

$\mathcal{S}_{Fer}^{\leq,i}$ is the space of all functions that bound the age of all memory blocks mapping to cache set i from above. $\mathcal{C}_{Fer}^{\leq}$ is the Cartesian product of n-times $\mathcal{S}_{Fer}^{\leq}$, the n cache sets are treated independently of each other. In the following let $ab = (ab_0, \dots, ab_{n-1})$ and $ab' = (ab'_0, \dots, ab'_{n-1}) \in \mathcal{C}_{Fer}^{\leq}$. We give the partial order and join operation.

$$\begin{aligned}ab \sqsubseteq_c ab' &\Leftrightarrow \forall 0 \leq i < n. ab_i \sqsubseteq_s ab'_i\\ ab_i \sqsubseteq_s ab'_i &\Leftrightarrow \forall b \in \mathcal{B}_i. ab_i(b) \leq ab'_i(b)\\ ab \sqcup_c ab' &= \left(ab_0 \sqcup_s ab'_0, \dots, ab_{n-1} \sqcup_s ab'_{n-1}\right)\\ ab_i \sqcup_s ab'_i &= \lambda b \in \mathcal{B}_i. \max(ab_i(b), ab'_i(b))\\ \top &= (\lambda b \in \mathcal{B}_0.\infty, \dots, \lambda b \in \mathcal{B}_{n-1}.\infty)\end{aligned}$$

For a given memory block b, $ab_{cs(b)}(b)$ bounds the age of b from above, i.e. b has at most this age. The function $cs : \mathcal{B} \to \{0, \dots, n-1\}$ computes the cache set the



Classification	Meaning
H	cache hit
M	cache miss
⊤	unclassified
⊥	undefined

Figure 5: Hasse diagram of the classification lattice *Cl*.

given memory block maps to. Thus, it extracts the index bits from the address of the given memory block and is formally defined as

$$cs(b) = addr(b)/linesize \mod nsets$$

As long as the age bound of a block is less than k, the block is guaranteed to be in the cache. An age bound ∞ means that the block may be in the cache but is not guaranteed to be.

Update First, we define the abstract transformer, also called update function, for cache sets $U^{\leq}_{s,Fer} : \mathcal{S}^{\leq,i}_{Fer} \times \mathcal{B}_i \to \mathcal{S}^{\leq,i}_{Fer}$. For a given abstract cache set and a memory block, it returns an abstract cache set that results from the access to that block.

$$U^{\leq}_{s,Fer}(ab_i, a) := \lambda b \in \mathcal{B}_i. \begin{cases} 0 & : b = a \\ ab_i(b) & : b \neq a \wedge ab_i(a) \leq ab_i(b) \\ ab_i(b) + 1 & : b \neq a \wedge ab_i(a) > ab_i(b) < k - 1 \\ \infty & : b \neq a \wedge ab_i(a) > ab_i(b) \geq k - 1 \end{cases}$$

Finally, we define the update function for whole caches $U^{\leq}_{c,Fer} : \mathcal{C}^{\leq}_{Fer} \times 2^{\mathcal{B}} \to \mathcal{C}^{\leq}_{Fer}$. For a given abstract cache and a set of possibly accessed memory blocks, it returns an abstract cache that results from an arbitrary access to one of the given memory blocks.

$$U^{\leq}_{c,Fer}([ab_0, \ldots, ab_{n-1}], B) := \bigsqcup_{b \in B} \left[ab_0, \ldots, U^{\leq}_{s,Fer}(ab_{cs(b)}, b), \ldots, ab_{n-1}\right]$$

Classification The classification function derives for an access to a set of memory blocks and an abstract cache whether the access always results in a hit, miss or whether nothing can be said about this access. The classification domain *Cl* is given by the Hasse diagram in Figure 5.

Again, we first define the classification function for abstract must cache sets $Class^{\leq}_{s,Fer} : \mathcal{S}^{\leq,i}_{Fer} \times \mathcal{B}_i \rightarrow Cl$.

$$Class^{\leq}_{s,Fer}(ab_i, a) := \begin{cases} \text{H} & : ab_i(a) < \infty \\ \text{M} & : a \notin B := \{b \in \mathcal{B}_i \mid ab_i(b) < \infty\} \wedge |B| = k \\ \top & : \text{otherwise} \end{cases}$$

Finally, we define the classification function for whole abstract must caches $Class^{\leq}_{c,Fer} : \mathcal{C}^{\leq}_{Fer} \times 2^{\mathcal{B}} \rightarrow Cl$.

$$Class^{\leq}_{c,Fer}([ab_0, \ldots, ab_{n-1}], B) := \bigsqcup_{b \in B} Class^{\leq}_{s,Fer}(ab_{cs(b)}, b)$$

3 Relational Cache Analysis

First, we give an overview on the structure of this part of the thesis. In Section 3.1 we examine the behaviour of Ferdinand's must cache analysis in the presence of accesses with imprecisely determined addresses and point out what can be improved. We also outline further key motivations for us to work on the relational cache analysis. An overview on the overall framework — the congruence analysis, the relational cache analysis and the interaction between them — is given in Section 3.2. In Section 3.3.1 we define cache trace semantics. All analysis domains that are subsequently presented can be related to this cache trace semantics. Section 3.3 formalises the notion of congruence analysis. We specify what the congruence analysis is supposed to do and relate it to the underlying (cache) trace semantics. A collecting name-instrumented cache semantics that is our underlying concrete semantics is explained in Section 3.4.1. Section 3.4.2 describes an approach to the relational cache analysis. An example is examined in Section 3.5. Finally, Section 3.6 outlines some open questions on how to further improve the precision of the relational cache analysis as well as the congruence analysis.

3.1 Motivation

Ferdinand's Cache Analysis — An example We apply Ferdinand's must cache analysis to an example program with memory accesses to addresses that are imprecisely determined, e.g. input-dependent or stack-relative accesses.

Consider the example program in Figure 6. Generating its control-flow graph and annotating the results of Ferdinand's must cache analysis yields Figure 7. The generated control-flow graph is represented on the left hand side. Next to each instruction, the abstract cache state after the execution of the respective

```
int pending = 1;

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

Setting:

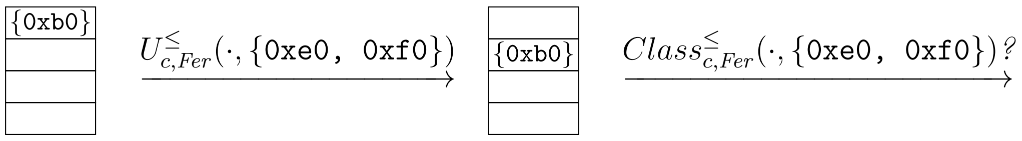
- write-allocate LRU cache
- 4-way associative
- 2 cache sets
- stack pointer unknown

Figure 6: Example program

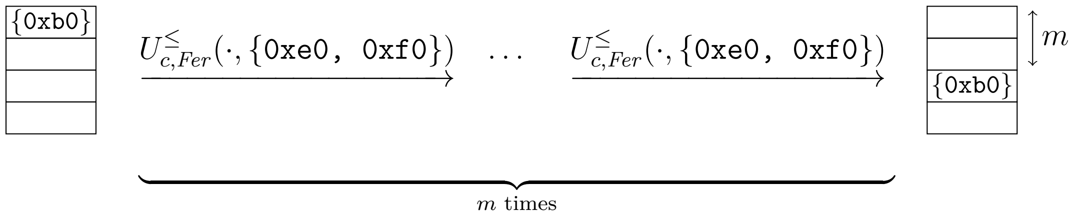
instruction is shown. The variable `pending` is located at address `0x601018`. Its address is statically known as it is a global variable in a statically linked binary. The addresses of all other accesses are completely unknown because they are either input dependent or stack relative with an unknown stack pointer.

We make several interesting observations that reveal room for improvement.

Observation 3.1 (No representation). *For memory accesses with imprecisely determined addresses, the memory block which is accessed during runtime is not known at analysis time. Cache analyses that employ memory blocks as abstract cache elements are unable to model such accesses precisely and thus cannot predict hits for these.*


{0xb0}
$U^{\leq}_{c,Fer}(\cdot, \{\texttt{0xe0, 0xf0}\})$
{0xb0}
$Class^{\leq}_{c,Fer}(\cdot, \{\texttt{0xe0, 0xf0}\})$?

Observation 3.2 (Excessive aging). *In an LRU must cache, memory accesses with imprecisely determined addresses age cached elements overly pessimistically. m accesses to the same but unknown memory block age cached elements by m.*


{0xb0}
$U^{\leq}_{c,Fer}(\cdot, \{\texttt{0xe0, 0xf0}\})$
...
$U^{\leq}_{c,Fer}(\cdot, \{\texttt{0xe0, 0xf0}\})$
{0xb0}
m
m times

The need by current cache analyses for exactly determined addresses for precise cache predictions motivates this work as we want to overcome this requirement. This enables us, e.g., to predict hits for accesses whose addresses depend on the possibly unknown input. The example program is revisited in Figure 12 on page 30 and we demonstrate the effectiveness of our relational cache analysis in the presence of accesses with imprecise address information with the help of this example.

Reducing context sensitivity Analysing stack-relative accesses using, e.g., the method of Kim et al. [11] requires to distinguish all call sites to obtain precise stack pointer values within each context. This results in a huge number of different call strings and many analysis contexts, namely one for each possible value the stack pointer can have.

Consider the example program in Figure 8. Analysing function `comp` using state-of-the-art cache analyses requires to distinguish four call strings — namely

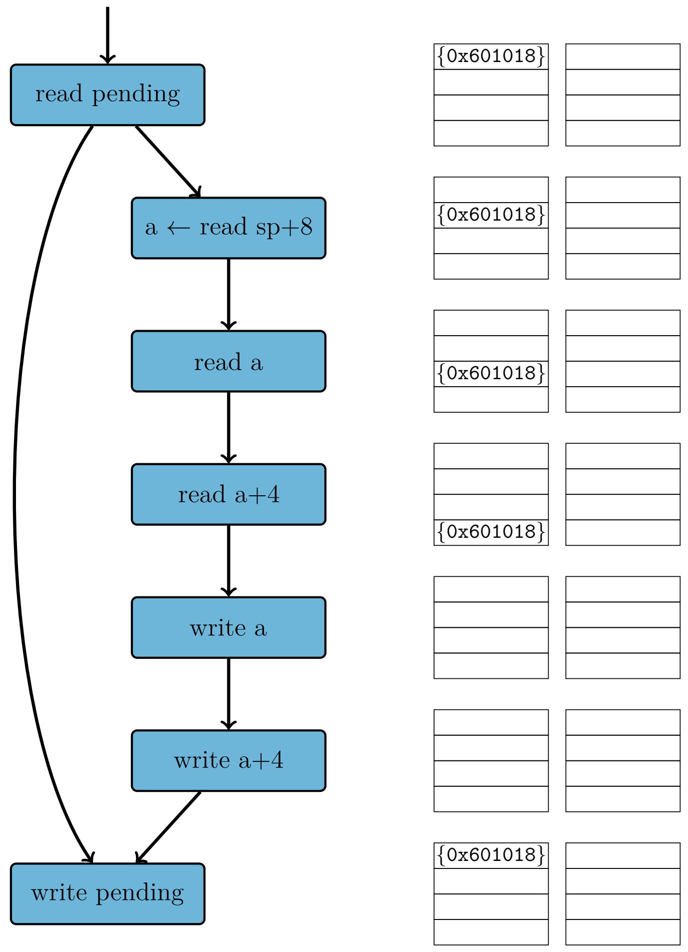


Figure 7: Analysis results using the LRU must cache analysis by Ferdinand. The generated control-flow graph of the analysed program is shown on the left hand side. Next to an instruction, the abstract must cache state after the execution of this instruction is represented.

```
int main() {
  ... = A();
  ...
  ... = B();
  ...
  ... = comp();
}

int B() {
  ... = comp();
}

int A() {
  ... = comp();
  ...
  ... = B();
}

int comp() {
  ...
}
```

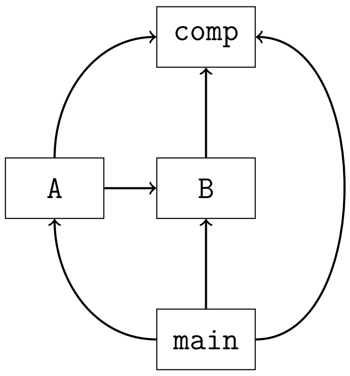


Figure 8: Program and call graph demonstrating the need for highly context-sensitive traditional cache analysis.

`main.comp`, `main.A.comp`, `main.A.B.comp` and `main.B.comp` — in order to predict stack-relative accesses in `comp`. Instead of running the analysis in four contexts with different stack pointers, we only need one run of our relational cache analysis.

Similarly to the previous example, analysing array accesses within a loop often requires to unroll the loop (completely). The store operation in Figure 9 always results in a write hit since the preceding load operation just accessed the same array element. Since the index, and thus the accessed address, is changing from iteration to iteration, obtaining precise address information for the traditional cache analysis requires loop unrolling. The relational cache analysis classifies the store operation as hit — without unrolling the loop.

Decreasing such forms of context sensitivity without losing too much precision in (data) cache analysis is also a motivation for our work as it can potentially speed up the overall analysis. Detailed results for these kinds of programs are also discussed in Section 6.1.

```
for (int i = 0; i < 50; ++i) {
  ... = a[i]
  a[i] = ...
}
```

Figure 9: Another example necessitating context sensitivity for the traditional cache analysis: Array accesses within a loop.

3.2 Overview

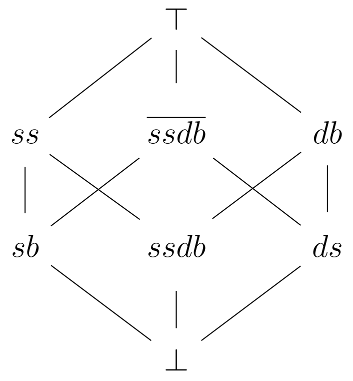
Our relational cache analysis is based on the same observation as Ferdinand's cache analysis (Observation 2.10 on page 11). He interprets the observation in the following way. If one statically knows the exact memory blocks that are accessed during program execution, one can easily test whether memory blocks are different or the same. Thus, an analysis can track the k memory blocks that were accessed most recently in order to predict cache hits or cache misses. We will interpret the observation in a more general way. Actually, it is not necessary to know the precise address of a memory block that is accessed. According to the observation, it is already sufficient to know whether blocks that are accessed coincide or differ.

Symbolic names First, we introduce the concept of a *symbolic name*.

Definition 3.3 (Symbolic name)**.** A symbolic name is a unique identifier for a static memory reference. Hence, it can also be seen as a representative for all memory blocks that might be accessed by this memory reference. The set of symbolic names is denoted by $\mathcal{N}$.

What makes an element of a set a symbolic name is that it is associated to a static memory reference. Each such static memory reference is associated with a uniquely determined element from this set. Thus two different static memory references also have different symbolic names. This is necessary for our analysis to work properly. If two different static memory references, accessing different memory blocks, would have the same symbolic name, one could not distinguish between them and one might classify accesses as hits (same symbolic name) although we access different memory blocks.

In theory, we can use $\mathcal{N} = \mathcal{L}$, where $\mathcal{L}$ denotes the set of program locations with associated instructions. We thereby assume that one instruction corresponds to one static memory reference and accesses exactly one memory block per execution. In practice, we have to extend our notion of symbolic names because there are instructions that might trigger more than one memory access. This is postponed to Section 5.



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

Figure 10: Hasse diagram of the lattice of relations.

A symbolic name potentially represents several memory accesses and each memory access itself can be seen as having the corresponding symbolic name attached. Symbolic names are now used as our new abstract cache elements. We thereby solve the problems arising from Observation 3.1 as accesses with imprecisely determined addresses can now be explicitly modeled in our abstract cache.

Relating symbolic names Although we are now able to model accesses with imprecisely computed addresses, we still face the question how to check whether symbolic names denote coinciding memory blocks or differing ones. This is important for classifying accesses and to update abstract cache states. As we are no longer interested in the addresses of memory accesses, we consider *relations* between symbolic names. The lattice of relations, $\mathcal{R}$, is shown and described in Figure 10. The purpose of each relation is further discussed in Section 3.3.3 and Section 3.4.2. We use relations, e.g., to

- *classify hits*: the relation between the currently accessed symbolic name s and another symbolic name t that is still in the cache, is sb
- *do precise updates*: accessing a symbolic name s does not age a symbolic name t if the relation between them is ds.

Framework Figure 11 shows the relational cache analysis framework. On the left, we have the modular *congruence analysis* with a clearly defined interface: given two symbolic names, it returns the relation between the two. Due to abstraction of the underlying analyses, it is not possible to always get the precise relation between two symbolic names, i.e. sb, ss db or ds, but it is guaranteed to give a

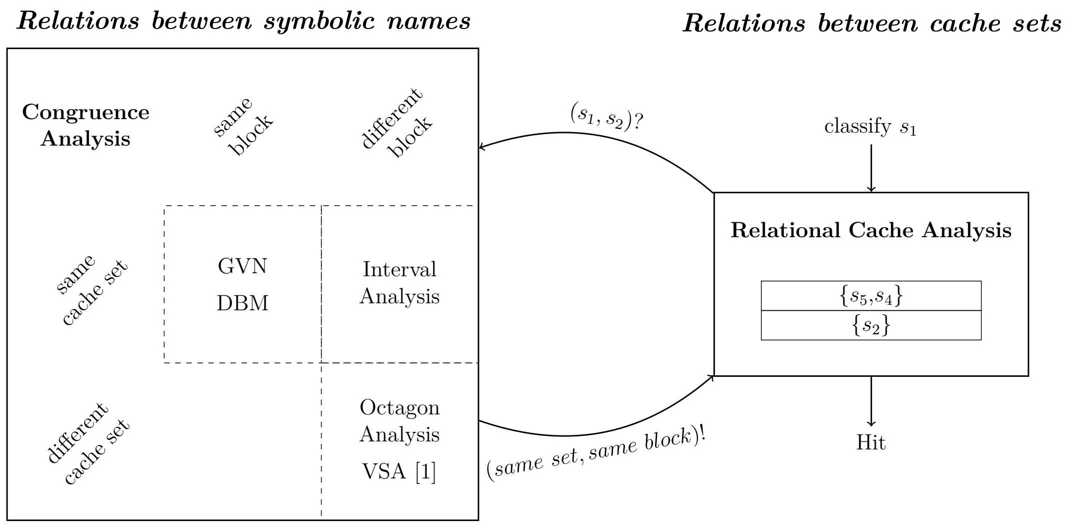


Figure 11: Interaction between congruence analysis and cache analysis

safe approximation, e.g. ss. The congruence analysis is *modular* because one could plug arbitrary analyses into it without changing the interface or the relational cache analysis. The left box also shows some examples of useful analyses that can compute relations between memory accesses. They are further discussed in Section 5.

On the right, we have the *relational cache analysis*. It uses symbolic names instead of concrete memory blocks and relations between symbolic names instead of comparing addresses of the concrete memory blocks. Note that it does not treat the cache sets independently like the traditional cache analysis does. Intuitively, as we do not distinguish cache sets, there is one abstract *relational cache* and no notion of abstract cache set. If a symbolic name s resides in the relational cache, this means that the memory block it represents could potentially map to any cache set and it is guaranteed to reside in the concrete cache, no matter which cache set it actually maps to.

Now, we take a closer look at the interaction between the relational cache analysis and the congruence analysis. Assume we want to classify the symbolic name s_1, so we ask the relational cache analysis to classify s_1. The relational cache analysis asks the congruence analysis for relations between s_1 and each of the symbolic names that reside in the abstract cache. For the symbolic names s_1 and s_2, the congruence analysis can guarantee that s_1 will access the same memory block as s_2 did before. The relational cache analysis can now predict a hit for the access to s_1 because it will access the same memory block as s_2 did before and this memory block has at most age one and thus is still in the cache.

Note that the information flow in the framework is unidirectional: the relational cache analysis depends on the results of the congruence analysis, but not the other way around. Therefore, the congruence analysis can be performed in isolation. As the relational cache analysis depends on the congruence analysis, we parametrise the abstract cache domain with the results of the congruence analysis.

3.3 Formalizing Congruence Analysis

Intuitively, the congruence analysis is supposed to give safe approximations on the relations between symbolic names. A relation $r \in \mathcal{R}$ between two symbolic names is safe if for each execution trace the relation r' between the two memory blocks that have been accessed most recently via these symbolic names is at least as precise as r ($r' \sqsubseteq_{\mathcal{R}} r$). A formal definition is given below.

3.3.1 Cache Trace Semantics

Instead of considering traces of complete system states, we focus on traces that only contain the information about the corresponding memory accesses during the execution. That is, we consider the underlying domain

$$\mathcal{T} := 2^{(V \times \mathcal{N} \times \mathcal{B})^*}$$

Let $Trace_P \subseteq (V \times \mathcal{N} \times \mathcal{B})^*$ be the semantics describing all memory access sequences through the program P. Such finite sequences are of the form

$$\tau = \langle v_1, s_1, b_1 \rangle \circ \ldots \circ \langle v_{l-1}, s_{l-1}, b_{l-1} \rangle \circ \langle v_l, s_l, b_l \rangle$$

Each triple consists of a program point, the symbolic name associated with the respective instruction and the memory block that is accessed by that instance of the instruction. We now have the most precise description of the history of memory accesses at any program point. Therefore, it is suitable to formally describe relations between symbolic names. Before addressing the issue how to obtain the relational information, we first define the collecting semantics.

3.3.2 Collecting Semantics

In the following, let $v \in V$ be a fixed program point. We now give the collecting semantics that keeps track of which memory blocks have been accessed most recently via symbolic names.

Lattice The underlying domain is given by

$$\mathcal{D}_{coll} := 2^{\mathcal{N} \to \mathcal{B}_\perp}$$

We can also directly define the partial order $\sqsubseteq = \subseteq$, the join function $\sqcup = \cup$ as well as the top element $\top = \mathcal{N} \to \mathcal{B}_\perp$ and the bottom element $\perp = \emptyset$. Let $blk \in \mathcal{D}_{coll}$. $blk(s) = \perp$ means that s was not accessed yet.

Concretization function We will now relate the collecting semantics with the underlying concrete semantics, our cache trace semantics. We therefor define a concretization function $\gamma_{\mathcal{D}_{coll}} : \mathcal{D}_{coll} \to \mathcal{T}$

$$\gamma(F) = \bigcup_{blk \in F} \{\tau \in (V \times \mathcal{N} \times \mathcal{B})^* \mid \tau = \tau' \circ \langle v, \_, \_ \rangle \wedge \forall s \in \mathcal{N}. last(\tau, s) = blk(s)\},$$

where F is the information at program point v, which we fixed before. The auxiliary function *last* is defined as follows

$$last(\tau, s) = \begin{cases} last(\tau') & : \tau = \tau' \circ \langle v_l, s_l, b_l \rangle \wedge s_l \neq s \\ b_l & : \tau = \tau' \circ \langle v_l, s_l, b_l \rangle \wedge s_l = s \\ \perp & : \tau = \epsilon \end{cases}$$

and computes the block that was most recently accessed via the given symbolic name.

Update function Finally, we define the update function of the collecting semantics $U_{\mathcal{D}_{coll}} : \mathcal{D}_{coll} \times (\mathcal{N} \times \mathcal{B}) \to \mathcal{D}_{coll}$.

$$U_{\mathcal{D}_{coll}}(F, (s, b)) = \bigcup_{blk \in F} \{blk[s \mapsto b]\}$$

where

$$blk[s \mapsto b] := \lambda t. \begin{cases} b & : t = s \\ blk(t) & : \text{otherwise} \end{cases}$$

3.3.3 Congruence Analysis

In the previous section, we have described the concrete semantics we base our congruence analysis on. This section formalises which kind of information a congruence analysis is supposed to compute. Later, we show how it can be used for cache analysis.

For each program point v, which we assume to be fixed again, a congruence analysis is supposed to compute a function

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

whose meaning is defined by the following predicate

$$consbcgr : (\mathcal{N} \to \mathcal{B}_\perp) \times (\mathcal{N} \times \mathcal{N} \to \mathcal{R}) \to \mathbb{B}$$

consbcgr checks whether a symbol-to-block mapping, *blk*, respects the results of our congruence analysis module, cgr_v. Formally, it is defined as

$$\begin{aligned} consbcgr(blk, cgr_v) = \forall s_1, s_2 \in \mathcal{N}.c = cgr_v(s_1, s_2), b_1 = blk(s_1), b_2 = blk(s_2) : \\ (c \neq \top \quad &\Rightarrow b_1 \neq \perp \wedge b_2 \neq \perp) \\ \wedge(c \sqsubseteq db \quad &\Rightarrow b_1 \neq b_2) \\ \wedge(c \sqsubseteq ss \quad &\Rightarrow cs(b_1) = cs(b_2)) \\ \wedge(c \sqsubseteq \overline{ssdb} &\Rightarrow b_1 = b_2 \vee cs(b_1) \neq cs(b_2)) \end{aligned}$$

Let s_1 and s_2 be two symbolic names, c the relation between them and b_1 and b_2 the memory blocks that have been accessed most recently via the respective symbolic name or $\perp$ according to *blk*. In case b_1 or b_2 is $\perp$, i.e. at least one of s_1 and s_2 has not been accessed so far, there cannot be a relation between these symbolic names, in which case the relation is $\top$. If the relation c between symbolic names s_1 and s_2 is not $\top$, both symbolic names have been accessed already and thus b_1 and b_2 cannot be $\perp$.

Depending on the actual relation, further constraints must hold. Let us consider two examples. First, if the relation between two symbolic names is *db*, b_1 and b_2 must denote different blocks. Second, consider the relation *sb*. Then, the constraints for ss and $\overline{\text{ss db}}$ apply. The constraints for ss guarantee that the cache set b_1 and b_2 map to are equal. This excludes the second case in the disjunction arising from the constraint for $\overline{ssdb}$. Thus $b_1 = b_2$ must hold, which expresses the meaning of *sb*. The constraints for the other kinds of relations are similar.

With the help of this predicate we can define the concretization function

$$\gamma : (\mathcal{N} \times \mathcal{N} \to \mathcal{R}) \to \mathcal{D}_{coll}$$

$$\gamma(cgr_v) = \{blk \in (\mathcal{N} \to \mathcal{B}_\perp) \mid consbcgr(blk, cgr_v)\}$$

After having specified the meaning of the congruence analysis, we now address the second module: the relational cache analysis.

3.4 Relational Cache Analysis

After defining what the congruence analysis is supposed to compute, namely relations between pairs of symbolic names, we will now define the relational cache analysis that makes use of this relational information. We start by defining the concrete name-instrumented cache domain — our underlying concrete collecting semantics. Furthermore, we formalise the abstract relational cache domain and explain how classifications of accesses as well as updates on abstract relational caches work using the calculated relational information.

3.4.1 Concrete Name-Instrumented Cache Domain

In the following, we will refer to concrete cache sets and concrete caches. These are defined here

$$\mathcal{S}_B := \mathcal{L}_B^k$$

$$\mathcal{C}_B := (\mathcal{S}_B)^n$$

$\mathcal{S}_B$ is the set of k-way associative concrete cache sets. $\mathcal{C}_B$ is the set of concrete caches, where n is the number of cache sets.

Lattice The domain we are about to define is an extended version of the above collecting semantics, additionally taking the cache into account. Our concrete relational cache domain is thus defined as the following powerset lattice

$$\mathcal{I} := 2^{(\mathcal{N} \to \mathcal{B}_\perp) \times \mathcal{C}_B}$$

Let $(blk, cc) \in \mathcal{I}$. Like before, blk keeps track of the memory block each symbolic name has accessed most recently. $blk(s) = \perp$ means that this symbolic name was not accessed yet. cc is a concrete cache that emerges from the initial cache and an access sequence respecting blk. It is an instrumented version of the standard concrete cache domain $2^{\mathcal{C}_B}$. This instrumentation is a technical requirement for later proving our framework correct. We have the usual partial order and join operation.

Classification Function Next, we define the concrete classification function $Class : \mathcal{I} \times \mathcal{N} \to Cl$.

$$Class(CC, (s, b)) := \bigsqcup_{(blk,cc) \in CC} \begin{cases} \mathrm{H} & : age(cc, b) < \infty \\ \mathrm{M} & : \text{otherwise} \end{cases}$$

where $age : \mathcal{C}_B \times \mathcal{B} \to \mathcal{AB}^{\leq}$ is defined as

$$age([ccs_0, \ldots, ccs_{n-1}], b) := age(ccs_{cs(b)}, b)$$

and $age : \mathcal{S}_B \times \mathcal{B} \to \mathcal{AB}^{\leq}$

$$age([b_0, \ldots, b_i, \ldots, b_{k-1}], b) := \begin{cases} i & : b_i = b \\ \infty & : \text{otherwise} \end{cases}$$

Given a concrete cache and a memory block, *age* returns the age of the memory block, i.e. the position of the memory block in the cache. If the age of the accessed memory block is less than ∞, the block is guaranteed to reside in the cache and thus is classified as a hit. Otherwise a miss is predicted.

Update Function Next, we define the transformer, the update function for sets of name-instrumented caches $U : \mathcal{I} \times \mathcal{N} \to \mathcal{I}$

$$U(CC, (s, b)) := \bigcup_{(blk,cc) \in CC} \{(blk[s \mapsto b], U_c(cc, b))\}$$

The transformer for a concrete cache $U_c : \mathcal{C}_B \times \mathcal{B} \to \mathcal{C}_B$ is defined as

$$U_c([ccs_0, \ldots, ccs_i, \ldots, ccs_{n-1}], b) := [ccs_0, \ldots, U_s(ccs_i, b), \ldots, ccs_{n-1}],$$

where $i = cs(b)$. For a concrete cache set $U_s : \mathcal{S}_B \times \mathcal{B} \to \mathcal{S}_B$ the update is

$$U_s([b_0, \ldots, b_{k-1}], b) := \begin{cases} [b_i, b_0, \ldots, b_{i-1}, b_{i+1}, \ldots, b_{k-1}] & : b_i = b \\ [b, b_0, \ldots, b_{k-2}] & : \text{otherwise} \end{cases}$$

3.4.2 Abstract Relational Cache Domain

After having defined the underlying concrete cache semantics, we consider the abstract domain to demonstrate how to operate on caches based on symbolic names and using relations between symbolic names.

Our abstract relational cache domain is parametrised by the results of the congruence analysis. Each domain operation depends on the results of the congruence analysis that are valid for the program point the operation is performed at.

Lattice The relational abstract domain is defined using the following function space.

$$\mathcal{C}^{\leq}_{rel} := \mathcal{N} \to \mathcal{AB}^{\leq}$$

The aim of the analysis is to bound the ages of symbolic names from above. Please note the structural similarities to the analysis described in Section 2.4.

First, we need an auxiliary function

$$eab^{\leq} : \left(\mathcal{N} \to \mathcal{AB}^{\leq}\right) \times (\mathcal{N} \times \mathcal{N} \to \mathcal{R}) \to \left(\mathcal{N} \to \mathcal{AB}^{\leq}\right)$$

that takes an element from the domain *ab* and the congruence analysis results that are valid at the current program point cgr_v and returns a function called *effective age bound*. For a symbolic name s, not only $ab(s)$ bounds the age of s from above, but each $ab(t)$ in case that s and t are in *sb* relation at the current program point. The effective age bound takes care of this fact:

$$eab^{\leq}(ab, cgr_v) = \lambda s \in \mathcal{N}. \min\{ab(t) \mid t \in \mathcal{N} \wedge cgr_v(s,t) = \mathrm{sb}\},$$

where the minimum over an empty set is defined as ∞. We use

$$eab^{\leq}(ab, cgr_v) = \lambda s \in \mathcal{N}. \min_{t \approx_{\mathrm{sb}} s}\{ab(t)\}$$

for short. The partial order and join operation are then defined as follows.

$$\begin{aligned} ab \sqsubseteq ab' &= \forall s \in \mathcal{N}. eab^{\leq}(ab, cgr_v)(s) \leq eab^{\leq}(ab', cgr_v)(s) \\ ab \sqcup ab' &= \lambda s \in \mathcal{N}. \max(eab^{\leq}(ab, cgr_v)(s), eab^{\leq}(ab', cgr_v)(s)) \\ \top &= \lambda s \in \mathcal{N}. \infty \end{aligned}$$

Concretization Before defining the abstract update and classification functions, we relate our abstract cache semantics to the name-instrumented concrete cache semantics. In the concretization function

$$\gamma^{\leq}_{rel} : \mathcal{C}^{\leq}_{rel} \to \mathcal{I}$$

we make use of the predicate *consbcgr* introduced in Section 3.3.3. *consbcgr* checks whether a symbol-to-block mapping respects cgr_v, the results of our congruence analysis module. Now, we can define the concretization function $\gamma^{\leq}_{rel} : \mathcal{C}^{\leq}_{rel} \to \mathcal{I}$.

$$\begin{aligned} \gamma^{\leq}_{rel}(ab) := \{(blk, cc) \in (\mathcal{N} \to \mathcal{B}_{\perp}) \times \mathcal{C}_B \mid {} & consbcgr(blk, cgr_v) \wedge \\ & \forall s \in \mathcal{N}. blk(s) \neq \perp \Rightarrow age(cc, blk(s)) \leq eab^{\leq}(ab, cgr_v)(s)\} \end{aligned}$$

The abstract cache represents a concrete cache if and only if its symbol-to-block mapping respects the results of the congruence analysis module and the age of the memory block a symbolic name accessed most recently respects the effective age bound of this symbolic name in the abstract cache.

Normalization function In our domain, there exist different elements $ab, ab' \in \mathcal{C}^{\leq}_{rel}$ that express the same: $ab \sqsubseteq ab' \wedge ab' \sqsubseteq ab$. Therefore we will employ a normalization function in the sense of [5]. This enables easier classification and update functions as well as a tighter abstract domain with the same expressive power. Indeed, the normalization function is defined already: $eab^{\leq}$ (Definition 3.4.2).

The tighter domain can be obtained by putting symbolic names denoting the same memory block together into one equivalence class. The lexicographically smallest symbolic name is chosen as the representative of the equivalence class. Only the age bounds for the representative elements are stored.

Note that the normalization depends on the current program point as the results of the congruence analysis also depend on the current program point. Thus, two symbolic names s and t may belong to the same equivalence class at one program point and belong to different equivalence classes at another program point.

Classification Function Next, we define the classification function that operates on the normalised domain.

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

The analysis classifies an access to a symbolic name s as hit if and only if the corresponding (effective) age bound is less than ∞ and thus s is guaranteed to be in the cache.

Update function We define the update function $U^{\leq}_{rel} : \mathcal{C}^{\leq}_{rel} \times \mathcal{N} \to \mathcal{C}^{\leq}_{rel}$ by

$$U^{\leq}_{rel}(ab, s) := \lambda t. \begin{cases} 0 & : srel = sb \\ ab(t) & : srel \in \{ds, \overline{ssdb}\} \\ ab(t) & : srel \sqsupseteq_{\mathcal{R}} ssdb \wedge ab(s) \leq ab(t) \\ ab(t) + 1 & : srel \sqsupseteq_{\mathcal{R}} ssdb \wedge ab(s) > ab(t) \wedge ab(t) < k - 1 \\ \infty & : srel \sqsupseteq_{\mathcal{R}} ssdb \wedge ab(s) > ab(t) \wedge ab(t) \geq k - 1 \end{cases}$$

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

The age bounds of symbolic names that most recently have accessed the same memory block as s does now, are set to 0 as the memory block just gets accessed again. The age bounds of symbolic names that have a higher age than s before the access are not changed. The symbolic names that might map to the same cache set as s and have a lower age are aged by one or evicted according to their age. The most interesting case is the second one. Note that we do not have to age elements that are guaranteed to map into a different cache set or are guaranteed to either have accessed the same memory block as s or to be in a different cache set. The latter case can occur when we miss alignment information. We cannot predict hits in that cases but we still do not treat these elements overly pessimistically.

Theorems

All theorems about the correctness of the analysis and their respective proofs can be found in Appendix A.

3.5 Example

Consider the function swap and the cache parameters in Figure 6 on page 15. We will examine this example once more, but this time using our relational cache analysis. The overall analysis and the following explanation always refer to Figure 12. For the sake of simplicity, we do not normalise explicitly.

Congruence analysis In the first of the three phases, we associate unique symbolic names with each instruction accessing memory. Relational information between pairs of symbolic names are computed in the second phase and are shown in the rightmost column. We only list relevant congruence information. Take a closer look at the relations $cgr_v(s_5, s_3)$ and $cgr_v(s_5, s_4)$. Global value numbering might find out that s_5 and s_3 access the same but unknown address, thus $cgr_v(s_5, s_3) = \text{sb}$. Similarly, another analysis using the results of the GVN might find out that the addresses of s_5 and s_4 differ by four. Depending on the alignment of the memory blocks, that are actually accessed, within the cache line and the cache line size, they either access the same memory block or the accesses affect different cache sets. As no alignment information is currently taken into account, we get $cgr_v(s_5, s_4) = \overline{\text{ss}\,\text{db}}$.

Cache analysis We do not examine the whole example, but point out the important updates and the classifications. First consider the access to s_4 and the corresponding update of the abstract cache. As $cgr_v(s_4, s_1) = cgr_v(s_4, s_2) = \top$ and the access might bring a new element to the cache, we have to pessimistically assume that s_1 and s_2 both age by one. On the other hand, s_3 needs not be aged as $cgr_v(s_4, s_3) = \overline{\text{ss}\,\text{db}}$. Either s_3 accessed the same memory block as s_4 or the two accesses affect different cache sets. In the first case, the age of s_3 would be reset to 0 and in the second case, the age would not change as the access to s_4 affects another cache set.

Now, consider the access to s_5. The congruence analysis tells us that s_5 will now access the same memory block as s_3 did ($cgr_v(s_5, s_3) = \text{sb}$) and as s_3 is guaranteed

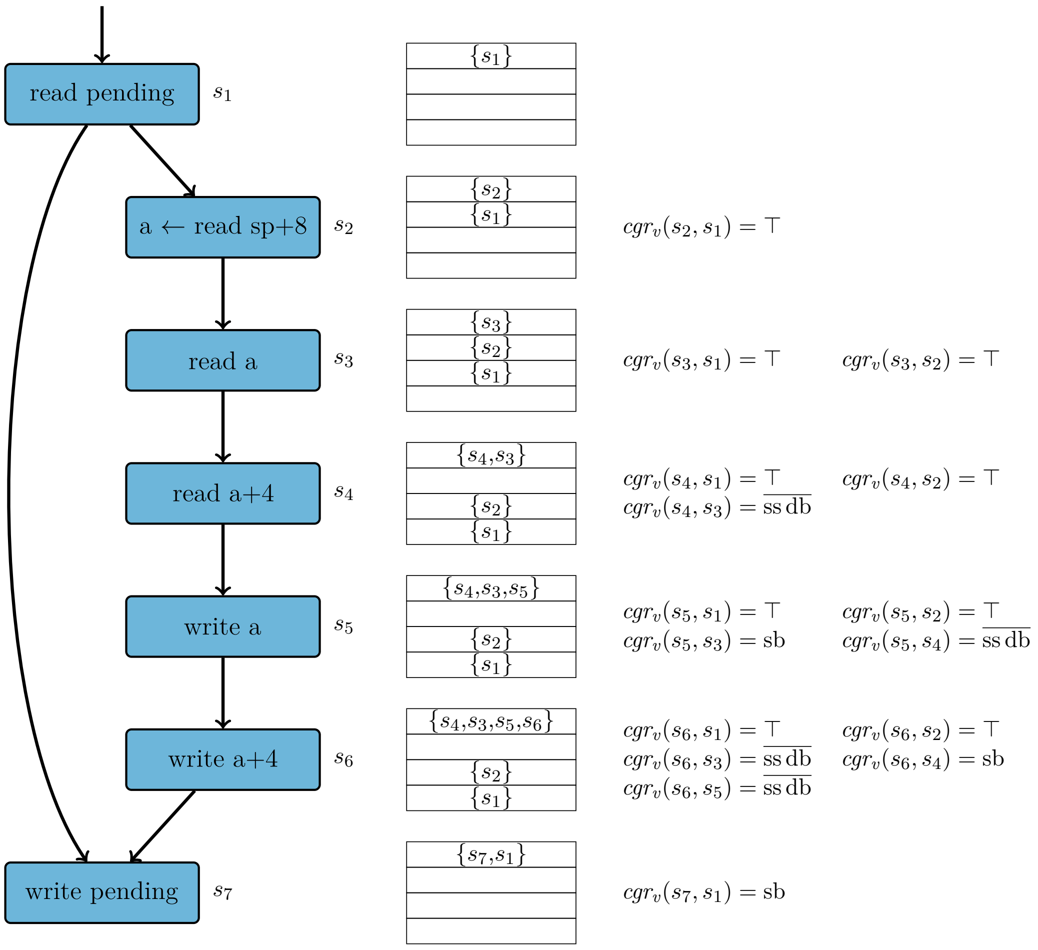

read pending
s_1
$a \leftarrow$ read sp+8
s_2
read a
s_3
read a+4
s_4
write a
s_5
write a+4
s_6
write pending
s_7
$\{s_1\}$
$\{s_2\}$ $\{s_1\}$
$cgr_v(s_2, s_1) = \top$
$\{s_3\}$ $\{s_2\}$ $\{s_1\}$
$cgr_v(s_3, s_1) = \top$ $cgr_v(s_3, s_2) = \top$
$\{s_4,s_3\}$ $\{s_2\}$ $\{s_1\}$
$cgr_v(s_4, s_1) = \top$ $cgr_v(s_4, s_2) = \top$
$cgr_v(s_4, s_3) = \overline{\text{ss db}}$
$\{s_4,s_3,s_5\}$ $\{s_2\}$ $\{s_1\}$
$cgr_v(s_5, s_1) = \top$ $cgr_v(s_5, s_2) = \top$
$cgr_v(s_5, s_3) = \text{sb}$ $cgr_v(s_5, s_4) = \overline{\text{ss db}}$
$\{s_4,s_3,s_5,s_6\}$ $\{s_2\}$ $\{s_1\}$
$cgr_v(s_6, s_1) = \top$ $cgr_v(s_6, s_2) = \top$
$cgr_v(s_6, s_3) = \overline{\text{ss db}}$ $cgr_v(s_6, s_4) = \text{sb}$
$cgr_v(s_6, s_5) = \overline{\text{ss db}}$
$\{s_7,s_1\}$
$cgr_v(s_7, s_1) = \text{sb}$

Figure 12: Analysis results using the relational must cache analysis

to be in the cache, the access is classified as cache hit. As s_3 is guaranteed to have age 0, no other cache elements have to be aged.

Results Using our relational cache analysis, we can predict the accesses to s_5, s_6 and s_7 as hits. All three could not be classified using Ferdinand's analysis (see Figure 7 on page 17): the first two due to imprecise address information and the latter one because of too pessimistic updates in the presence of accesses to imprecisely determined addresses.

```
...
if (i < 5) {
  x = ...; //symbolic name: s
} else {
  x = ...; //symbolic name: t
}
... = x; //symbolic name: u
```

Figure 13: Example pointing out room for improvement of the congruence analysis

3.6 Open Issues

In the following we list some issues on how to further improve the precision of the analysis.

Congruence analysis Consider the code snippet in Figure 13. Obviously the access to `x` via symbolic name u results in a cache hit during program execution. Either the `if` branch is executed in which case `x` is loaded into the cache via s or the `else` branch is executed in which case `x` is also loaded into the cache but this time via t.

Our analysis will not be able to predict the access via u as a hit due to the following problems with the congruence analysis. According to the semantics of our congruence analysis, $cgr_v(s, t) = \top$ as s and t can never occur together in a trace. Taking the join of the abstract cache as it is, neither s nor t are afterwards guaranteed to still be in the cache. The problem is not with the cache analysis but with the semantics of the congruence analysis that is not yet sophisticated enough. Refining the semantics of the congruence analysis might solve this issue.

Relational cache analysis The relational cache domain can also be further improved in terms of precision. Consider the access sequence $\langle s, t, u\rangle$ of symbolic names with relations $cgr_v(s, t) = cgr_v(s, u) = \top$ and $cgr_v(t, u) = \text{ds}$. After the first access, s has guaranteed age 0. The consecutive two accesses each age s pessimistically by one, although t and u access different cache sets. Thus the analysis can only guarantee age 2 although the actual age is 1. This is due to the lack of history information in the abstract cache domain. In case one knows the actual cache sets t and u map into, one possibility is to maintain age bounds on symbolic names for each cache set. This way, one only updates the age bounds of a symbolic name that are associated with cache sets that may be affected by an access.

4 Related Work

Work closest to our contribution is related with data cache analysis because in data cache analysis one has to care about accesses with imprecisely determined addresses. There are several approaches to data cache analysis and we discuss them in the following and compare them with our approach.

Li et al. [12] present an approach based on Integer Linear Programming (ILP). First, they employ a data flow analysis to compute a range of possible addresses or a list of addresses that are accessed sequentially. Second, a cache conflict graph is constructed per cache set which corresponds to load/store instructions that may access addresses mapping to the same cache set. Finally, they generate integer linear equations from the conflict graph and solve them.

In [10], Hur et al. compute precise addresses of *static* accesses, i.e. accesses relative to a global base pointer or relative to the stack pointer. To have precise address information about accesses relative to the stack pointer, they distinguish all call sites, thus needing a huge number of call strings. Since each possible value of the stack pointer is considered separately, it is possible to obtain precise address information for stack-relative accesses for each call site. So-called *dynamic* accesses are handled by adding the miss penalty twice, once for the access itself and once for a possibly evicted cache block. Note, that this approach only applies to direct-mapped data caches.

In [11] and [19], slight extensions to the previous approach are proposed. Misclassification of accesses as static and dynamic is reduced by tracking which accesses (indirectly) depend on the global base pointer or the stack pointer. The number of misses caused by dynamic accesses is bounded using the pigeonhole principle. An additional miss penalty is only added if a *useful cache block* might be replaced.

Mueller et al. [14] introduce a technique for direct-mapped instruction cache analysis, the so called static cache simulation. White et al. [21] extend the static cache simulation to data caches in the first part of their paper. In a first step, they calculate addresses of accesses. In order to gain precise addresses of accesses to the stack, they distinguish different call sites. The algorithm for calculating data cache states for direct-mapped caches adds all possibly accessed blocks to the abstract cache state during an access to imprecisely determined addresses. Therefore, their cache simulation is similar to a May-analysis. It remains unclear to us how hits can be predicted for such accesses in general. In the second part, this static cache simulation is extended towards set-associative instruction caches. Analysis of set-associative data caches is not discussed explicitly.

In [13], Lundqvist presents a method to analyse data cache behaviour when data structures are accessed in a predictable manner but with unknown placement in memory. In a first phase, a simulator classifies memory accesses once using an unknown base pointer and once using a fixed one. Data structures are called predictable with unknown placement, if an access to the data structure is classified as unpredictable by the first classification, but classified as predictable by the second classification. In a second phase, the accesses to each of these data structures are analysed separately using fixed base pointers. The final conflict analysis detects possible interferences of these access sequences by an exhaustive algorithm. This basic analysis can be made arbitrarily accurate, but in general it causes an explosion of the state space. There are neither restrictions on the programs nor on the cache configurations that can be analysed. Nevertheless, for programs that extensively use many data structures only the basic algorithm with low accuracy might be applicable for efficiency reasons. The basic algorithm leads to significant overestimation considering the presented benchmarks.

Ferdinand et al. [7] propose a must and a may analysis for predicting cache behaviour of set-associative LRU caches. The must (may) analysis gives upper (lower) bounds on the age of cache elements at a specific program point. A memory access can then be classified as a hit (miss). In [8], they propose methods on predicting the data cache behaviour. For scalar variables whose addresses can be statically computed, must and may analyses are employed. For a restricted class of programs, local memory hit ratios can be calculated using a special data dependence analysis of arrays. Additional restructuring methods are used to improve data locality in nested loops. In contrast to these methods, a persistence analysis is proposed that bounds the number of possible cache misses — also in presence of accesses with imprecisely determined addresses.

In [18], Sen et al. present an abstract–interpretation based method to analyse data caches. First of all, an address analysis is proposed (Circular Linear Progressions) that allows the precise modeling of induction variables and linear computations. The programs that can be analysed are therefore restricted to special forms of loops. The actual cache analysis is based on the approach of Ferdinand et al. [7]. The classification and update functions of the must analysis are extended to handle circular linear progressions (CLP) and thus also accesses with imprecisely determined addresses to some extent. Virtual and physical loop unrolling is employed to increase precision.

Blieberger et al. [2] propose a method for analytically deriving a cache hit function at compile-time. In a first step, based on *symbolic evaluation*, they generate a symbolic tracefile. Symbolic expressions and recurrences are employed to give a constructive description for all possible memory accesses in chronological order. In the second step, the symbolic cache evaluation derives an analytical function from the generated symbolic tracefile. The function returns the number of hits for a given program input. The approach is illustrated using array-manipulating programs. Programs with many different data accesses with imprecisely determined addresses cause many distinctions of cases during the transformation from conditional recurrences into unconditional ones. It is unclear to us how programs with different paths are analysed. It would have been interesting to see how the approach scales for these programs and how precise the approximations for solving the corresponding recurrences is.

In [20], Wegener presents an approach to improve the precision of static analysis of programs using loops while keeping the effort low. Instead of virtually unrolling a loop completely, it is only unrolled partially. In order to obtain precise analysis results, multiple accumulative contexts are used instead of one. Within this setting, he proposes a so-called *relational cache analysis.* With the help of alignment information, which can be obtained by having multiple accumulative contexts and difference bound matrices (DBM) he can conclude whether two accesses go to the same cache line. Each of the accesses whose memory block is guaranteed to be still in the cache is compared with the current access to determine whether they hit the same cache line.

The *symbolic expressions* [2] are similar to our notion of *symbolic names.* Note that we employ *relations between symbolic names* in order to detect interferences instead of doing a state space exploration as in [2] or [13].

Ferdinand's analysis [8] only gives precise results if the accessed addresses can be exactly predicted whereas our relational analysis does not require precise address information. This is discussed in detail in Section 3.1.

The cache analysis proposed in [20] can be seen as a special case of our framework. The relational view on accesses is restricted to the sb relation whereas we also take other relations, e.g. ds and $\overline{ssdb}$ into account. The cache analysis itself is specialised to the use of alignment information and DBMs. Our relational cache analysis is independent of the analyses that obtain relational information, thus we could make use of the alignment information and DBMs without any adjustments. Due to the restrictions on sb relations, his analysis can guarantee at most k memory blocks to be in the cache. Due to ds relations, our relational cache analysis can potentially guarantee this for as many elements as fit in the cache, namely $n \cdot k$.

5 Implementation

5.1 libFirm

libFirm[1] is the C-library implementing the intermediate representation Firm. Firm is a low-level intermediate representation that is based on SSA-form, described by Rosen et al. [17], and that allows easy implementation of program analyses and transformations. We also make use of the implementation of a C-compiler that transforms C-programs into Firm's intermediate representation. We will not go into detail about compilers and SSA-form in this thesis. Using libFirm has several advantages for us.

Low-level representation We do not have to care about parsing given input programs or translating them into our intermediate representation. Although our tool takes C-programs instead of binaries, the internal assembler (ASM) graph representation our cache analysis works on, is very close to a binary. The instructions, the nodes in the ASM graph, are already specific to the final target platform. All kinds of compiler optimizations are finished, instructions as well as the basic blocks are completely scheduled and resources like registers are also already allocated. We can also compute the final addresses an instruction will finally have in the binary. Thus, we get a perfectly realistic intermediate representation of our programs for free.

x86 Address Computation As we use the x86 backend of libFirm, we will sketch how an address is computed in x86. In general, an address computation can be expressed via the following formula

$$B + I \cdot S + O$$

where B denotes base, I index, S scale and O offset. B and I are thereby machine registers. They are both optional and I cannot be the *esp* register. S is a constant $\in \{1, 2, 4, 8\}$ and O a 32-bit number.

5.2 Symbolic Names

In Section 3.2, we have introduced the concept of symbolic names. For the sake of simplicity, each program location has been associated with one unique symbolic name. In practice, a single instruction can trigger several memory accesses. First the instruction itself is referenced during the fetch phase in the pipeline. Then,

[1] `http://pp.info.uni-karlsruhe.de/firm/Main_Page`

the execution of the instruction can, depending on the type, trigger up to two data memory accesses. Thus we associate up to three symbolic names with one instruction — one for each potential memory access. More formally, we choose $\mathcal{N} = \mathcal{L} \times \{I, D1, D2\}$ for implementation.

5.3 Congruence Analyses

In this section, we briefly describe the congruence analyses we employed and how to obtain information about the relation between symbolic names from their respective results. Note that although we describe the derivation of relational information for each analysis independently, one gains more precision if the results of all analyses are used simultaneously.

Several program analyses and transformations are performed during the optimization phase of the compiler, e.g. global value numbering. We get the results of these analyses and with their help we can already determine useful relations between symbolic names. Additionally, we implemented a simple interval analysis.

5.3.1 Global Value Numbering

Synopsis Global value numbering (GVN) [17] is a technique that assigns each variable and each expression a certain value number. In case two expressions have the same value number, they are provably equivalent and one could for example remove redundant code. As opposed to local value numbering, these global value numbers are valid across basic blocks.

Use We can use GVN to detect that a symbolic name always accesses the same memory block as another symbolic name did before. Formally, given two symbolic names s and t, t provably access the same memory block as s did before if

- the expressions representing the respective address computations have the same global value number and
- none of the addresses is recomputed on any path from s to t.

We can also find other relations using the results of GVN. Let s and t be two symbolic names and $B_1 + I_1 \cdot S_1 + O_1$, $B_2 + I_2 \cdot S_2 + O_2$ the respective address computations described in Section 5.1. Assume GVN finds out that $B_1 + I_1 \cdot S_1$ and $B_2 + I_2 \cdot S_2$ always evaluate to the same value and the offsets O_1 and O_2 are known.

- If we have information about the alignment of the accessed memory blocks and know that $|O_2 - O_1| + \mathit{alignment} \leq \mathit{linesize} - 4$, we can conclude that they access the same block.

- If we have no alignment information but know that $|O_2 - O_1| \leq \mathit{linesize} - 4$ and we have at least 2 cache sets, we can derive the relation $\overline{ssdb}$.
- If we know that $\mathit{linesize} \leq |O_2 - O_1| \leq \mathit{linesize} \cdot (\mathit{nsets} - 1)$, we can derive the relation ds.

5.3.2 Interval Analysis

Synopsis Interval analysis computes for each register and each memory location an interval $[l, u]$ including all possible values the expression can evaluate to. It is a well-known analysis that is already used in similar tools. We implemented an interval analysis, that basically propagates the range of the stack pointer to the stack-relative accesses and performs basic arithmetic operations. Although it is a simple analysis we can derive several useful relations.

Use In the best case we can derive the precise addresses for given symbolic names s (b_1) and t (b_2).

- $b_1 = b_2$ gives us the sb relation,
- $b_1 \neq b_2 \wedge cs(b_1) = cs(b_2)$ relation $ssdb$ and
- $cs(b_1) \neq cs(b_2)$ relation ds.

All other cases only allow to obtain less precise relations. As there are many possibilities to derive such relational information, we only give one more example. Let $[l_s, u_s]$ and $[l_t, u_t]$ be the interval approximating the addresses for the symbolic name s and t, respectively. If $u_s < l_t$ and $l_t - u_s \geq \mathit{linesize}$, we can derive the relation db between s and t.

5.3.3 Other useful analyses

There are other program analyses that compute useful relational information but are currently not implemented in our framework. We will mention three of them and give examples that would profit from them.

Difference-Bound Matrices and Octagon Analysis The Difference-Bound Matrix (DBM) is a datastructure that stores constraints of the form $x - y \leq c$ and $\pm x \leq c$, where x and y are variables (or registers) and c a constant. The octagon analysis computes constraints of form $\pm x \pm y \leq c$ and $\pm x \leq c$, where x and y are again variables or registers and c is a constant. This kind of relational information can be used for deriving relations between symbolic names.

```
...
... = a[0];
for (int i = 1; i <= 100; i += 4) {
  a[i]++;
}
a[0] = ...;
...
```

Figure 14: Code snippet demonstrating the use of the Value-set Analysis

Let s_1 and s_2 be symbolic names that load from addresses x and y, respectively. If $|x - y| \leq \mathit{linesize} - 4$ and we have information about the alignment within the cache line, we can derive the sb relation. If $\mathit{linesize} \cdot (\mathit{nsets} - 1) \geq |x - y| \geq \mathit{linesize}$, we can derive the ds relation. Note that this formula has to be rewritten in order to fit the given kind of constraints.

Value-set Analysis The value-set analysis of Balakrishnan et al. [1] computes an over-approximation on the addresses that might be accessed via a specific symbolic name. They use reduced interval congruences (RIC) within their value-set analysis to represent possibly accessed addresses. A RIC can be represented by a 4-tuple (a, b, c, d) (written as $a \times [b, c] + d$) that denotes the values described by $\{a \cdot N + d \mid N \in [b, c]\}$. The domain is more expressive than the interval domain we currently employ.

To demonstrate the use of these RICs for the relational cache analysis, consider the example program in Figure 14. Assume that the addresses of the array `a` range from 1000 to 1400. Furthermore, let the cache be 4-way associative with 4 sets and a line size of 4 bytes.

Both accesses to `a[0]` go to address 1000. This can be represented in both domains, the interval and the reduced interval congruence domain.

Within the loop, every forth array element `a[i]` is accessed starting from 1004 up to 1400. Using an interval domain, the possibly accessed addresses are represented by an interval from 1004 up to 1400. The interval domain is not able to take linear congruences into account, such as only every forth array element is accessed. Using the reduced interval congruence domain, the addresses that might be accessed can be expressed as $16 \times [0, 24] + 1004$. Using interval information, we have to pessimistically assume that we access cache set 0 — the one that holds `a[0]` — 25 times during the execution of the loop. At the end, the cache analysis cannot guarantee that memory block `a[0]` still resides in the cache. With the help of the reduced interval congruence, we are able to conclude that all accesses within the loop map to cache set 1 and thus do not affect cache set 0. Thus, the relational cache analysis could predict the second access to `a[0]` as a hit.

6 Evaluation

6.1 Classes of Examples

There are several 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 will also outline other examples that require some extensions to our congruence analysis that are currently beyond the scope of this work.

6.1.1 Stack-relative memory accesses

Consider the function `comp` in Figure 15 in the corresponding context presented in the motivating example Figure 8 on page 18.

The function makes heavy use of local variables. Since there are that many local variables but only few registers available on the target machine, the compiler needs to spill some locals on the stack. The addresses of these spill slots depend on the stack pointer. To predict hits for such accesses, prior analyses need to distinguish

```
int comp(int a1, int a2, int a3,
            int b1, int b2, int b3,
            int c1, int c2, int c3) {
  int p1 = a2 * b3 + a3 * b2;
  int p2 = a3 * b1 + a1 * b3;
  int p3 = a1 * b2 + a2 * b1;

  int p4 = a2 * c3 + a3 * c2;
  int p5 = a3 * c1 + a1 * c3;
  int p6 = a1 * c2 + a2 * c1;

  int p7 = b2 * c3 + b3 * c2;
  int p8 = b3 * c1 + b1 * c3;
  int p9 = b1 * c2 + b2 * c1;

  return p1 * c1 + p2 * c2 + p3 * c3 +
         p4 * b1 + p5 * b2 + p6 * b3 +
         p7 * a1 + p8 * a2 + p9 * a3;
}
```

Figure 15: Function `comp`, which computes the sum of three triple products

Table 1: Number of 32 static memory references predicted as hits

Configuration			Precise SP `0xc000`		Imprecise SP `0xc000 - 0xc008`	
ls	*k*	*n*	traditional	relational	traditional	relational
4	4	4	18	18	0	14
4	8	4	18	18	0	15
8	4	4	25	23	0	15
8	8	4	25	25	0	18
16	4	4	28	28	0	18
16	8	4	28	28	0	18

the call sites of `comp` to obtain precise address information. This approach results in a higher running time of the analysis, because this requires one analysis instance per call site. Using our relational view, we are able to predict hits in the presence of an unknown stack pointer value.

Regard Table 1 that lists the number of cache hit predictions for function `comp`. We benchmark different cache configurations, different stack pointer configurations — once precisely and once imprecisely determined stack pointer — as well as different analysis types.

First, let us look at the analysis results using a precisely determined stack pointer. We recognise that the relational cache analysis produces almost the same number of cache hit predictions as the traditional one. For configuration linesize 8, associativity 4 and number of sets 4, the relational cache analysis performs a little bit worse. This is caused by the situation already mentioned in Paragraph 3.6.

Second, let us consider the results for an imprecisely determined stack pointer. The traditional cache analysis is not able to predict any hits — as expected. The relational cache analysis still predicts up to 83% of the hits it has predicted in the case of the precise stack pointer. The analysis results are naturally less tight than the ones before because the relational information is preciser in case of the precise stack pointer, but the analysis time is lower.

6.1.2 Array reuse within one loop iteration

Consider Figure 16. This is a slightly modified function taken from the Mälardalen benchmark suite[2]. The accesses to the array `int a[50][50]` are of major interest to us. Again, there is the possibility to increase context sensitivity, i.e. completely unroll the loops in order to gain the exact address for each access. This is of course not feasible for programs with deeply nested loops due to exponential increase in

[2]`http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

```
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);  // access (a)
      if (i == j)
        a[i][j] *= 10;              // access (b)
      else
        a[i][j] *= 2;               // access (c)
      w += a[i][j];                 // access (d)
    }
    b[i] = w;
  }
  return 0;
}
```

Figure 16: Function `main` taken from `ludcmp.c`

Table 2: Number of 13 static memory references predicted as hits

Configuration			Precise SP `0xc000`		Imprecise SP `0xc000 - 0xc008`	
ls	*k*	*n*	traditional	relational	traditional	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

running time. Our relational cache analysis is able to predict hits for the last three accesses to `a[i][j]` (namely accesses `(b)`, `(c)` and `(d)`) during each loop iteration — without any loop transformation or any rise of context sensitivity.

We benchmark again different cache configurations, different analysis types as well as different stack pointer configurations. The results are shown in Table 2. No matter which configuration we look at, the relational cache analysis always predicts three more hits than the traditional one. These three predicted hits correspond to the accesses `(b)`, `(c)` and `(d)` — as expected. Each time one of these accesses is performed, it will result in a cache hit. Although the analyses have been performed without increasing context sensitivity or transforming the loop, the relational cache analysis predicts 23% more static memory references as hits compared to the traditional cache analysis.

6.1.3 Input-dependent addresses

Finally, consider Figure 17 also taken from the Mälardalen benchmark suite. This time the base address of the array accesses is unknown as it is an argument. Additionally the indices may depend on the argument `lx`. State-of-the-art cache analyses are not able to classify any of these accesses as hits since their addresses are unknown. Increasing context sensitivity is not useful in this case and distinguishing all possible addresses of the argument is infeasible. Whereas the previous examples motivated our relational analysis by decreasing context sensitivity and thus running time of the analyses, this time our analysis generates a new class of programs that can be precisely analysed.

Consider the results of benchmarking different system configurations in Table 3. First, we focus on the case with a precisely determined stack pointer. The relational cache analysis is able to predict up to 11 hits — or 14% — more than the traditional cache analysis is able to. These hits are predicted for the stores in pass one and pass

```
void fdct(int *blk, int lx) {
  ...
  /* Pass 1: process rows. */
  ...
  /* Pass 2: process columns. */

  block=blk;
  for (i = 0; i<8; i++) {
    tmp0 = block[0]    + block[7*lx];
    tmp7 = block[0]    - block[7*lx];
    tmp1 = block[lx]   + block[6*lx];
    tmp6 = block[lx]   - block[6*lx];
    tmp2 = block[2*lx] + block[5*lx];
    tmp5 = block[2*lx] - block[5*lx];
    tmp3 = block[3*lx] + block[4*lx];
    tmp4 = block[3*lx] - block[4*lx];
    ...
    block[0]    = ...
    block[4*lx] = ...
    block[2*lx] = ...
    block[6*lx] = ...
    block[7*lx] = ...
    block[5*lx] = ...
    block[3*lx] = ...
    block[lx]   = ...
    /* advance to next column */
    block++;
  }
}
```

Figure 17: Function `fdct` taken from `fdct.c`

Table 3: Number of 123 static memory references predicted as hits

Configuration			Precise SP `0xc000`		Imprecise SP `0xc000 - 0xc008`	
ls	*k*	*n*	traditional	relational	traditional	relational
4	4	4	11	13	0	7
4	8	4	26	28	0	17
4	16	4	43	54	0	43
8	4	4	36	38	0	11
8	8	4	54	54	0	20
8	16	4	64	71	0	51
16	4	4	56	57	0	14
16	8	4	73	79	0	32
16	16	4	78	89	0	55

two of the program. The remaining hit predictions are for stack-relative accesses as the computation itself uses a lot of local variables, which are spilled.

In the second scenario with the imprecisely determined stack pointer, the traditional cache analysis cannot predict any hits — neither for the array accesses nor the stack-relative accesses. As the relational cache analysis can handle both kinds of accesses, the number of predicted hits of the relational cache analysis and the traditional one differ greatly. The relational cache analysis is able to classify up to 45% of all static memory references as hits.

6.1.4 Further examples

There are still many other classes of programs that benefit from our relational cache analysis. We will now outline two more of them. Analysis of these programs requires extensions to our, up to now simple, congruence analysis that are out of the scope of this thesis. Note, that the cache analysis itself does not need to be modified which emphasises the modular character of the overall analysis.

Position-independent instruction cache analysis The overall WCET-analysis takes a statically linked binary as argument. As the binary is statically linked, one knows the exact addresses of the specific instructions and the cache analysis can precisely classify accesses to the instruction cache. Having our relational cache analysis, we observe that the analysis does not require precise address information but relations between symbolic names. Analysing the instruction cache, we can obtain precise relations (*sb*, *ssdb*, *ds*) without relying on the addresses. This enables *position-independent instruction cache analysis*, that means we could analyse code of dynamically linked binaries or of dynamically loaded libraries. Concrete application areas will need further investigation.

Heap allocated data structures Up to now, analysing accesses to dynamically allocated data structures is beyond technical feasibility. First, function calls to `malloc(...)` influence the cache state and second, the dynamically allocated data structures do not have statically known base addresses. Recent work [9] tackles the first problem: CAMA — a cache aware memory allocator — allows constant response times and a predictable effect on the cache. Although the precise address of a dynamically allocated data structure is still not known, one at least knows the cache set it maps to because this is passed as argument to `cmalloc(..., set)`. In [6], Dudziak et al. present a method how to obtain alias information of accesses with the help of a shape analysis based on 3-valued logic. Combining these techniques with our relational cache analysis will be part of future work. It might allow the use of dynamically allocated data structures in programs for embedded systems, which is currently prohibited.

6.2 Results

The results we discussed in the previous section have shown that our relational cache analysis obtains classifications of memory accesses that are at least as precise as the ones of the traditional approach. In general, this is not the case because the precision of the relational cache analysis depends on the precision of the congruence analysis. There are rare examples for which our relational cache analysis yields worse results, e.g. the one mentioned in Paragraph 3.6. Remembering the key motivations for our work (Section 3.1), we meet all of these expectations:

Reducing context sensitivity We have shown that reducing the context sensitivity (distinguishing call sites, unrolling loops) has not that high effect on the relational cache analysis compared to the state-of-the-art approach. Although we also lose precision in these cases, we can predict a quite valuable number of memory accesses as cache hits. In the examples we discussed in detail, we could still predict up to 56% of the static memory references as hits — at most 31% less hits than possible when the context sensitivity is increased. The resulting WCET bound might be already tight enough for our purposes and the overall analysis time can be reduced.

Input-dependent addresses In case the address of a memory access depends on an unknown input, these accesses worsen the must cache information of the

```
int a[50];
void foo() {
  for (int i = 1; i < 50; ++i) {
    ... = a[i];
    a[i-1] = ...;
  }
}
```

Figure 18: Reuse across loop iterations

traditional cache analysis overly pessimistically. Furthermore, no new memory blocks are loaded into the abstract must cache. Thus no hits can be predicted for such accesses. We have shown how to overcome this dependency on precise address information and the results of our relational cache analysis are already very promising. We are able to predict hits for many of those accesses. In the examples we discussed, the relational cache analysis could predict up to 14% more hits compared to the traditional cache analysis.

6.3 Limitations

The results for these first examples are already very promising. Nevertheless, the relational cache analysis can be further improved.

Reconsider Figure 16. We are able to detect the cache hits resulting from the temporal locality within the scope of the loop. Nevertheless, there is still more potential reuse due to spatial locality to exploit. Assume cache lines of size 16 bytes. The first access to `a[i][j]` will only result in a miss every fourth iteration. During the other three iterations, the access results in a hit as the complete cache line has been loaded at once. Up to now, we are not able to deal with this, but it will definitely be investigated in future work.

Similarly, we are not yet able to predict hits that result from inter iteration reuse. Consider one more example in Figure 18. Except the first access to `a[i-1]`, all others will result in cache hits because the accessed memory block is the same as the memory block `a[i]` accessed one iteration before. Again, we are not yet able to predict such accesses to result in cache hits. Similar problems arise from inter loop reuse.

7 Summary

7.1 Conclusion

In Section 3.1 we have discussed several problems and weaknesses of currently used cache analyses. The contribution of this work is to characterise and analyse these problems and to present a new cache analysis, called relational cache analysis, that is able to overcome most of these weaknesses. The approach is three-fold.

Symbolic names We have introduced the notion of a symbolic name. A symbolic name is a representative for all memory blocks that might be accessed at a specific instruction. We can use these symbolic names as our new abstract cache elements, thus accesses with imprecisely determined addresses can be represented in an abstract cache, in contrast to the traditional analysis.

Relations between symbolic names In order to do precise updates and classifications on the new abstract caches, we have to know, e.g., whether a symbolic name that is accessed, always accesses the same memory block as a symbolic name that resides in the abstract cache. In that case we can predict a hit.

In general, we employ a modular congruence analysis that computes relations between symbolic names and can answer questions like the one above. The congruence analysis is independent from the cache analysis and can thus run as a preprocessing step. It has a clearly shaped interface that is important as adjustments in the internal parts of the congruence analysis does not affect the cache analysis that makes use of it. The analysis is modular since we can plug in several analyses and compute the actual relations between symbolic names from their results. We have shown several possible analyses that can be used for that purpose and how relational information is extracted.

Relational cache analysis We have defined a new cache analysis that uses symbolic names as abstract cache elements and relations between them to do updates and classifications. The analysis is based on the approach of abstract interpretation, a widely used technique to relate concrete and abstract semantics and to prove the soundness of analyses. We have proven our analysis correct; it is a sound approximation of the underlying concrete collecting semantics. Several classes of programs have been examined and the results are throughout promising although we only used very simple congruence analyses in our implementation. More powerful congruence analyses lead to better relational information and thus more precise cache predictions.

When analyzing stack-relative accesses without distinguishing call sites, the relational cache analysis can still classify up to 56% of the static memory references as hits — or up to 83% of the hits that can be predicted by the analysis with full context sensitivity. Although we lose precision here, we can shorten the overall analysis time and the WCET bound might already be tight enough. If increasing the context sensitivity is not feasible, e.g. in deeply nested loops, the relational cache analysis can nevertheless find out a reasonable amount of hits whereas the traditional cache analysis cannot predict any hits.

In case of input-dependent addresses, the traditional cache analysis cannot predict any hits as the addresses can only be imprecisely determined. Increasing context sensitivity does not increase the precision of the traditional cache analysis and distinguishing all possible inputs is infeasible. Our relational cache analysis does away with the popular misconception that precise address information is required for cache analysis. Up to 14% more hits can therefore be predicted by the relational cache analysis compared to the traditional one, if accesses with input-dependent addresses are involved. Thus, our cache analysis generates a new class of programs that can be precisely analysed.

7.2 Future Work

We have presented fundamental work on relational cache analysis. Nevertheless, there is a lot more to do in order to improve the analysis.

Congruence analysis We want to examine more congruence analyses, e.g. the Value-Set analysis [1] and how their information can be used to derive relational information. Concrete effects of the different congruence analyses on the precision of the relational cache analysis is interesting future work.

Cache analysis We will try to further increase the precision of the analysis, e.g. by taking partial positioning information into account.

The effect of spatial locality on the cache predictions is not negligible although we do not care about this up to now. Doing more precise predictions due to spatial locality effects will be a further aim.

We presented only a relational must cache analysis so far. A may analysis that uses symbolic names as abstract cache elements can be defined analogously.

In this thesis, we have focused on the LRU replacement policy, but there are also other replacement policies like first-in first-out (FIFO). We will investigate whether existing cache analyses, that use memory blocks as abstract cache elements, can be canonically extended to a relational cache analysis.

Applications Last but not least, we want to investigate further application areas like e.g. programs using dynamically allocated data structures. Therefore we could combine the shape analysis from [6] with our congruence analysis in order to obtain precise aliasing relations.

A Proofs

Lemma A.1 (Consistency of $\sqsubseteq^{\leq}_{rel}$ and $\sqcup^{\leq}_{rel}$). *The partial order $\sqsubseteq$ and the join function $\sqcup$ are consistent :*

$$\forall x, y \in \mathcal{C}^{\leq}_{rel} \ . \ x \sqcup y = y \Leftrightarrow x \sqsubseteq y$$

Proof. Let x and $y \in \mathcal{C}^{\leq}_{rel}$ and cgr_v be the results of the congruence analysis at the current program point. For the sake of readability let $x' = eab^{\leq}(x, cgr_v)$ and $y' = eab^{\leq}(y, cgr_v)$. Using the transitivity property of the sb relation $(*)$, we conclude

$$\begin{aligned}
x \sqcup y = y &\overset{Def}{\iff} \forall s \in \mathcal{N}. eab^{\leq}(x \sqcup y, cgr_v)(s) = y'(s)\\
&\overset{Def}{\iff} \forall s \in \mathcal{N}. eab^{\leq}(\lambda t \in \mathcal{N}. \max(x'(t), y'(t)), cgr_v)(s) = y'(s)\\
&\overset{Def}{\iff} \forall s \in \mathcal{N}. \min_{u \approx_{\text{sb}} s} \max(\min_{v \approx_{\text{sb}} u} x(v), \min_{v \approx_{\text{sb}} u} y(v))(s) = y'(s)\\
&\overset{(*)}{\iff} \forall s \in \mathcal{N}. \min_{u \approx_{\text{sb}} s} \max(\min_{v \approx_{\text{sb}} s} x(v), \min_{v \approx_{\text{sb}} s} y(v))(s) = y'(s)\\
&\iff \forall s \in \mathcal{N}. \max(x'(s), y'(s)) = y'(s)\\
&\iff \forall s \in \mathcal{N}. x'(s) \leq y'(s)\\
&\overset{Def}{\iff} x \sqsubseteq y
\end{aligned}$$ □

Theorem A.2 (Join semilattice). *$(\mathcal{C}^{\leq}_{rel}, \sqsubseteq^{\leq}_{rel})$ is a (finite) join - semilattice that satisfies the ascending chain condition.*

Proof. Reflexivity, antisymmetry and transitivity of $\sqsubseteq^{\leq}_{rel}$ follow directly from the respective properties of $\leq$. The binary relation $\sqcup^{\leq}_{rel}$ is defined on all pairs of $\mathcal{C}^{\leq}_{rel}$ and is consistent with $\sqsubseteq^{\leq}_{rel}$ (see Lemma A.1). □

Theorem A.3 (Monotonicity of $\gamma^{\leq}_{rel}$). *The concretization function γ is monotonic:*

$$\forall x, y \in \mathcal{C}^{\leq}_{rel} : x \sqsubseteq y \Longrightarrow \gamma(x) \subseteq \gamma(y)$$

Proof. Let $x \sqsubseteq y \in \mathcal{C}^{\leq}_{rel}$, cgr_v be the results of the congruence analysis at the current program point and $(blk, cc) \in \gamma(x)$. We have to show $(blk, cc) \in \gamma(y)$, i.e.

$$consbcgr(blk, cgr_v) \wedge \forall s \in \mathcal{N}. blk(t) \neq \bot \Rightarrow age(cc, blk(s)) \leq eab^{\leq}(y, cgr_v)(s)$$

As $(blk, cc) \in \gamma(x)$, $consbcgr(blk, cgr_v)$ hold. To see the last part of the conjunction, let $s \in \mathcal{N}$. If $blk(t) = \bot$, the implication is trivially satisfied. Otherwise, as $(blk, cc) \in \gamma(x)$ we have

$$age(cc, blk(s)) \leq eab^{\leq}(x, cgr_v)(s).$$

Since $x \sqsubseteq y$, also

$$eab^{\leq}(x, cgr_v)(s) \leq eab^{\leq}(y, cgr_v)(s),$$

and thus together with the transitivity of $\leq$

$$age(cc, blk(s)) \leq eab^{\leq}(y, cgr_v)(s). \qquad \square$$

Lemma A.4 (Normalization function $eab^{\leq}$)**.** *The normalization function does not change the concretization of abstract elements*

$$\gamma_{rel}^{\leq} = \gamma_{rel}^{\leq} \circ eab^{\leq}$$

Proof. Let $ab \in \mathcal{C}_{rel}^{\leq}$ and cgr_v be the results of the congruence analysis at the current program point. The claim directly follows from the idempotence property of $eab^{\leq}$ proven in Lemma A.5.

$$\begin{aligned}
&\gamma_{rel}^{\leq}(eab^{\leq}(ab, cgr_v)) \\
&= \{(blk, cc) \in (\mathcal{N} \to \mathcal{B}_{\perp}) \times \mathcal{C}_B \mid consbcgr(blk, cgr_v) \wedge \\
&\qquad \forall s \in \mathcal{N}.blk(s) \neq \perp \Rightarrow age(cc, blk(s)) \leq eab^{\leq}(eab^{\leq}(ab, cgr_v), cgr_v)(s)\} \\
&\overset{A.5}{=} \{(blk, cc) \in (\mathcal{N} \to \mathcal{B}_{\perp}) \times \mathcal{C}_B \mid consbcgr(blk, cgr_v) \wedge \\
&\qquad \forall s \in \mathcal{N}.blk(s) \neq \perp \Rightarrow age(cc, blk(s)) \leq eab^{\leq}(ab, cgr_v)(s)\} \\
&= \gamma_{rel}^{\leq}(ab) \qquad \square
\end{aligned}$$

Lemma A.5 (Normalization function $eab^{\leq}$)**.** *The normalization function is idempotent, reductive, and monotonic. Let cgr_v be the valid results of the congruence analysis at the current program point.*
For all $ab, ab' \in \mathcal{C}_{rel}^{\leq}$

$$\begin{gathered}
eab^{\leq}(ab, cgr_v) = eab^{\leq}(eab^{\leq}(ab, cgr_v), cgr_v) \\
eab^{\leq}(ab, cgr_v) \sqsubseteq ab \\
ab \sqsubseteq ab' \Rightarrow eab^{\leq}(ab, cgr_v) \sqsubseteq eab^{\leq}(ab', cgr_v)
\end{gathered}$$

Proof. First, we will prove the idempotence property as the other properties follow directly from the idempotence.

Idempotent Let $s \in \mathcal{N}$ be a symbolic name. We can conclude using the transitivity of the sb relation $(*)$

$$\begin{aligned}
& eab^{\leq}(eab^{\leq}(ab, cgr_v), cgr_v)(s) \\
= & \min_{t \approx_{\mathrm{sb}} s} \min_{v \approx_{\mathrm{sb}} t} \{ab(v)\} \\
\overset{(*)}{=} & \min_{t \approx_{\mathrm{sb}} s} \min_{v \approx_{\mathrm{sb}} s} \{ab(v)\} \\
= & \min_{v \approx_{\mathrm{sb}} s} \{ab(v)\} \\
= & eab^{\leq}(ab, cgr_v)(s)
\end{aligned}$$

After having proven the idempotence of $eab^{\leq}$, the other properties are easy to prove.

Reductive

$$\begin{aligned}
& eab^{\leq}(ab, cgr_v) \sqsubseteq ab \\
\Longleftrightarrow & \forall s \in \mathcal{N}. eab^{\leq}(eab^{\leq}(ab, cgr_v), cgr_v)(s) \leq eab^{\leq}(ab, cgr_v)(s) \\
\Longleftrightarrow & \forall s \in \mathcal{N}. eab^{\leq}(ab, cgr_v)(s) \leq eab^{\leq}(ab, cgr_v)(s)
\end{aligned}$$

The claim follows by the reflexivity of $\leq$ on the natural numbers.

Monotonic

$$\begin{aligned}
& ab \sqsubseteq ab' \\
\Longleftrightarrow & \forall s \in \mathcal{N}. eab^{\leq}(ab, cgr_v)(s) \leq eab^{\leq}(ab', cgr_v)(s) \\
\Longleftrightarrow & \forall s \in \mathcal{N}. eab^{\leq}(eab^{\leq}(ab, cgr_v), cgr_v)(s) \leq eab^{\leq}(eab^{\leq}(ab', cgr_v), cgr_v)(s) \\
\Longleftrightarrow & eab^{\leq}(ab, cgr_v) \sqsubseteq eab^{\leq}(ab', cgr_v)
\end{aligned}$$

□

Theorem A.6 (Local consistency of $U^{\leq}_{rel}$)**.** *Let cgr_v be the results of the congruence analysis at the current program point. The abstract transformer $U^{\leq}_{rel}$ is locally consistent with the concrete transformer U :*

$$\begin{aligned}
\forall x \in \mathcal{C}^{\leq}_{rel}\ \forall s \in \mathcal{N}, b \in \mathcal{B}. & \\
& (blk, cc) \in \gamma(x) \wedge consbcgr(blk[s \mapsto b], cgr_v) \Rightarrow \\
& \qquad U((blk, cc), s, b) \in \gamma(U^{\leq}_{rel}(x, s))
\end{aligned}$$

Proof. Let $x \in \mathcal{C}^{\leq}_{rel}$, $s \in \mathcal{N}$ and $b \in \mathcal{B}$. Furthermore let $(blk, cc) \in \gamma^{\leq}_{rel}(x)$ where $consbcgr(blk[s \mapsto b], cgr_v)$ holds. The latter condition ensures, that the local consistency must only hold for valid combinations of a symbolic name and a memory block. We therefore assume that $\mathit{blk}[s \mapsto b]$ is consistent with cgr_v: blk is a valid symbol-to-block mapping before the access, $s \mapsto b$ models the address computation and cgr_v represents valid relational information after the address computation. Substituting the definitions of $\gamma^{\leq}_{\mathit{rel}}$ and parts of U our proof goal becomes

$$\begin{aligned}&\mathit{consbcgr}(\mathit{blk}[s \mapsto b], \mathit{cgr}_v) \wedge \\ &\qquad\qquad \forall t \in \mathcal{N}. \mathit{blk}(t) \neq \bot \Rightarrow \mathit{age}(U_c(cc, b), \mathit{blk}(t)) \leq \mathit{eab}^{\leq}(U^{\leq}_{\mathit{rel}}(x, s), \mathit{cgr}_v)(t)\end{aligned}$$

The first part of the conjunction holds, as it is one of our assumptions. Now, let $t \in \mathcal{N}$. If $\mathit{blk}(t) = \bot$, the implication is trivially satisfied. If $\mathit{blk}(t) \neq \bot$, the proof goal is reduced to

$$\mathit{age}(U_c(cc, b), \mathit{blk}(t)) \leq \mathit{eab}^{\leq}(U^{\leq}_{\mathit{rel}}(x, s), \mathit{cgr}_v)(t).$$

We split the proof into three subcases.

(a) $\mathit{cgr}_v(s, t) = \mathit{sb}$
Now $\mathit{consbcgr}$ guarantees that $\mathit{blk}(s) = \mathit{blk}(t)$. Thus

$$\begin{aligned}\mathit{age}(U_c(cc, b), \mathit{blk}(t)) &= \mathit{age}(U_c(cc, b), b) \\ &= 0 && \text{(Definition } U_c\text{)} \\ &= \mathit{eab}^{\leq}(U^{\leq}_{\mathit{rel}}(x, s), \mathit{cgr}_v)(t) && \text{(Definition } U^{\leq}_{\mathit{rel}}\text{)}\end{aligned}$$

(b) $\mathit{cgr}_v(s, t) = \mathit{ds}$
Now $\mathit{consbcgr}$ guarantees that $\mathit{cs}(\mathit{blk}(t)) \neq \mathit{cs}(\mathit{blk}(s))$. Thus, only the cache set $\mathit{cs}(\mathit{blk}(s))$ is updated and

$$\begin{aligned}\mathit{age}(U_c(cc, b), \mathit{blk}(t)) &= \mathit{age}(cc, \mathit{blk}(t)) && \text{(Definition } U_c\text{)} \\ &\leq \mathit{eab}^{\leq}(x, \mathit{cgr}_v)(t) && ((\_, cc) \in \gamma^{\leq}_{\mathit{rel}}(x)) \\ &= \mathit{eab}^{\leq}(U^{\leq}_{\mathit{rel}}(x, s), \mathit{cgr}_v)(t) && \text{(Definition } U^{\leq}_{\mathit{rel}}\text{)}\end{aligned}$$

(c) $\mathit{cgr}_v(s, t) = \mathit{ssdb}$
We have now to distinguish three further cases.
1) $x(s) \leq x(t)$ We get two further subcases.
i) $\mathit{age}(cc, \mathit{blk}(t)) > \mathit{age}(cc, b)$

$$\begin{aligned}\mathit{age}(U_c(cc, b), \mathit{blk}(t)) &= \mathit{age}(cc, \mathit{blk}(t)) && \text{(Definition } U_c\text{)} \\ &\leq \mathit{eab}^{\leq}(x, \mathit{cgr}_v)(t) && ((\_, cc) \in \gamma^{\leq}_{\mathit{rel}}(x)) \\ &= \mathit{eab}^{\leq}(U^{\leq}_{\mathit{rel}}(x, s), \mathit{cgr}_v)(t) && \text{(Definition } U^{\leq}_{\mathit{rel}}\text{)}\end{aligned}$$

ii) $age(cc, blk(t)) < age(cc, b)$

$$\begin{aligned} age(U_c(cc,b), blk(t)) &= age(cc, blk(t)) + 1 && \text{(Definition } U_c\text{)} \\ &\leq age(cc, b) \\ &\leq eab^{\leq}(x, cgr_v)(s) && ((\_, cc) \in \gamma^{\leq}_{rel}(x)) \\ &\leq eab^{\leq}(x, cgr_v)(t) \\ &= eab^{\leq}(U^{\leq}_{rel}(x,s), cgr_v)(t) && \text{(Definition } U^{\leq}_{rel}\text{)} \end{aligned}$$

2) $x(s) > x(t) \wedge x(t) < k-1$
From the fact that one access to an LRU cache can age any element by at most one, we conclude

$$\begin{aligned} age(U_c(cc,b), blk(t)) &\leq age(cc, blk(t)) + 1 \\ &\leq eab^{\leq}(x, cgr_v)(t) + 1 && ((\_, cc) \in \gamma^{\leq}_{rel}(x)) \\ &= eab^{\leq}(U^{\leq}_{rel}(x,s), cgr_v)(t) && \text{(Definition } U^{\leq}_{rel}\text{)} \end{aligned}$$

3) $x(s) > x(t) \wedge x(t) \geq k-1$

$$\begin{aligned} age(U_c(cc,b), blk(t)) &\leq \infty \\ &= eab^{\leq}(U^{\leq}_{rel}(x,s), cgr_v)(t) && \text{(Definition } U^{\leq}_{rel}\text{)} \end{aligned}$$

The cases $cgr_v(s,t) = \text{ss}$, $cgr_v(s,t) = db$ and $cgr_v(s,t) = \overline{ssdb}$ can be reduced to the above three cases by case distinction. For $cgr_v(s,t) = \text{ss}$, we have e.g. to consider the cases sb and $ssdb$ and we have to prove that the update for ss correctly accounts for both cases. □

Theorem A.7 (Monotonicity of $U^{\leq}_{rel}$)**.** *The abstract transformer $U^{\leq}_{rel}$ is monotonic.*

$$\forall s \in \mathcal{N}\ \forall x, y \in \mathcal{C}^{\leq}_{rel}.x \sqsubseteq y \Rightarrow U^{\leq}_{rel}(x,s) \sqsubseteq U^{\leq}_{rel}(y,s)$$

Proof. Let $x \sqsubseteq^{\leq}_{rel} y \in \mathcal{C}^{\leq}_{rel}$ be abstract relational caches and s be a symbolic name. We have to show that

$$\forall t \in \mathcal{N}.U^{\leq}_{rel}(x,s)(t) \leq U^{\leq}_{rel}(y,s)(t).$$

Let t be a further symbolic name. We distinguish five cases according to $cgr_v(s,t)$.
(a) $cgr_v(s,t) = sb$

$$\begin{aligned} U^{\leq}_{rel}(x,s)(t) &= 0 && \text{(Definition of } U^{\leq}_{rel}\text{)} \\ &= U^{\leq}_{rel}(y,s)(t) && \text{(Definition of } U^{\leq}_{rel}\text{)} \end{aligned}$$

(b) $cgr_v(s,t) \in \{ds, \overline{ssdb}\}$

$$\begin{aligned} U^{\leq}_{rel}(x,s)(t) &= x(t) && \text{(Definition of } U^{\leq}_{rel}\text{)} \\ &\leq y(t) && \text{(Precondition)} \\ &= U^{\leq}_{rel}(y,s)(t) && \text{(Definition of } U^{\leq}_{rel}\text{)} \end{aligned}$$

(c) $cgr_v(s,t) \sqsupseteq_{\mathcal{R}} ssdb \wedge x(s) \leq x(t)$
Then, $U^{\leq}_{rel}(x,s)(t) = x(t)$ and $U^{\leq}_{rel}(y,s)(t)$ can either be $y(t)$ or $y(t)+1$ or ∞. In all cases

$$U^{\leq}_{rel}(x,s)(t) \leq U^{\leq}_{rel}(y,s)(t)$$

(d) $cgr_v(s,t) \sqsupseteq_{\mathcal{R}} ssdb \wedge x(s) > x(t) \wedge x(t) < k-1$
Then, $U^{\leq}_{rel}(x,s)(t) = x(t)+1$ and $U^{\leq}_{rel}(y,s)(t)$ can again either be $y(t)$ or $y(t)+1$ or ∞. For the latter two cases $U^{\leq}_{rel}(x,s)(t) \leq U^{\leq}_{rel}(y,s)(t)$ holds. Let us consider the first case. From $U^{\leq}_{rel}(y,s)(t) = y(t)$, we can conclude by the definition of $U^{\leq}_{rel}$, that $y(s) \leq y(t)$. Thus

$$\begin{aligned} U^{\leq}_{rel}(x,s)(t) &= x(t)+1 && \text{(Definition of } U^{\leq}_{rel}\text{)} \\ &\leq x(s) && \text{(Assumption (d))} \\ &\leq y(s) && (x \sqsubseteq^{\leq}_{rel} y) \\ &\leq y(t) \\ &= U^{\leq}_{rel}(y,s)(t) && \text{(Definition of } U^{\leq}_{rel}\text{)} \end{aligned}$$

(e) $cgr_v(s,t) \sqsupseteq_{\mathcal{R}} ssdb \wedge x(s) > x(t) \wedge x(t) \geq k-1$
As $x \sqsubseteq^{\leq}_{rel} y$, $y(t) \geq x(t) \geq k-1$ and thus $U^{\leq}_{rel}(y,s)(t) = \infty$.
$U^{\leq}_{rel}(x,s)(t) \leq U^{\leq}_{rel}(y,s)(t)$ is now trivially satisfied.

□

Theorem A.8 (Validity of the classification function $Class^{\leq}_{rel}$)**.** *Let cgr_v be the results of the congruence analysis at the current program point. The abstract classification function $Class^{\leq}_{rel}$ is valid with respect to the concrete classification function Class.*

$$\begin{aligned} &\forall x \in \mathcal{C}^{\leq}_{rel}\ \forall s \in \mathcal{N}\ \forall b \in \mathcal{B}. \\ &\qquad (blk, cc) \in \gamma^{\leq}_{rel}(x) \wedge consbcgr(blk[s \mapsto b], cgr_v) \Rightarrow \\ &\qquad\qquad Class(\gamma^{\leq}_{rel}(x), s, b) \sqsubseteq_{Cl} Class^{\leq}_{rel}(x, s) \end{aligned}$$

Proof. Let $x \in \mathcal{C}^{\leq}_{rel}$ be a normalised abstract relational cache and $(blk, cc) \in \gamma^{\leq}_{rel}(x)$. Furthermore let $s \in \mathcal{N}$ be a symbolic name and $b \in \mathcal{B}$ a memory block with

$consbcgr(blk[s \mapsto b], cgr_v)$. The latter condition ensures, that the local consistency must only hold for valid combinations of a symbolic name and a memory block. We therefore assume that $blk[s \mapsto b]$ is consistent with cgr_v: blk is a valid symbol-to-block mapping before the access, $s \mapsto b$ models the address computation and cgr_v represents valid relational information after the address computation.

We distinguish two cases, corresponding to possible results of $Class^{\leq}_{rel}(x, s)$. The case $Class^{\leq}_{rel}(x, s) = \top_{Cl}$ is trivial. Now, let $Class^{\leq}_{rel}(x, s) = \mathrm{H}$. Therefore, the age bound of the symbolic name s must be smaller than ∞. We need to show

$$age(cc, b) < \infty$$

From the definition of $\gamma^{\leq}_{rel}$ follows, that

$$\begin{aligned} age(cc, b) &\leq eab^{\leq}(x, cgr_v)(s) \\ &= x(s) < \infty \end{aligned}$$

Thus, $Class(\gamma^{\leq}_{rel}(x), s, b) = \mathrm{H}$. $\square$

B Symbols

Symbol	Meaning
$\mathcal{B}$	Set of memory blocks
$\mathcal{B}_\perp$	Set of memory blocks including $\perp$
$\mathcal{B}_i$	Set of memory blocks mapping into cache set i
$\mathcal{L}$	Set of program locations
k	Cache associativity
n	Number of cache sets
$\mathcal{L}_B$	Set of cache lines contents
$\mathcal{S}_B$	LRU cache sets
$\mathcal{C}_B$	LRU caches
age	Function computing the age of a memory block
$\mathcal{AB}^\leq$	Set of possible ages
$\mathcal{S}^\leq_{Fer}$	Abstract LRU must cache sets
$\mathcal{C}^\leq_{Fer}$	Abstract LRU must caches
$\mathcal{N}$	Set of symbolic names
$\mathcal{R}$	Set of relations between symbolic names
cgr_v	Function providing relation information
$consbcgr$	Function checking the consistency of symbol-to-block mappings
$eab^\leq$	Normalization function computing effective age bounds
$\mathcal{I}$	Concrete Name-Instrumented LRU caches
$\mathcal{C}^\leq_{rel}$	Abstract relational LRU must cache

Table 4: Symbols used in this thesis

References

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 12th International Conference on Compiler Construction (CC '04)*, pages 5–23. Springer-Verlag, 2004. On pages 21, 38, and 49.

[2] Johann Blieberger, Thomas Fahringer, and Bernhard Scholz. Symbolic cache analysis for real-time systems. *Real-Time Syst.*, 18:181–215, May 2000. On pages 33 and 34.

[3] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001. On page 6.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. On pages 6 and 8.

[5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992. On page 28.

[6] Tomasz Dudziak and Jörg Herter. Cache analysis in presence of pointer-based data structures. In Enrico Bini, editor, *Proceedings Work-In-Progress Session of the 23rd Euromicro Conference on Real-Time Systems*, pages 7–10, July 2011. On pages 46 and 50.

[7] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35:163–189, November 1999. On pages 5, 11, 12, and 33.

[8] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, 1998. Springer-Verlag. On pages 33 and 34.

[9] Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011. On page 46.

[10] Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heonshik Shin, and Chong Sang Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 308–319, Washington, DC, USA, 1995. IEEE Computer Society. On page 32.

[11] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, RTAS '96, pages 230–240, Washington, DC, USA, 1996. IEEE Computer Society. On pages 16 and 32.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, pages 254–263, Washington, DC, USA, 1996. IEEE Computer Society. On page 32.

[13] Thomas Lundqvist. Data cache timing analysis with unknown data placement. Technical report, Dept. of Computer Engineering, Chalmers, 2002. On pages 32 and 34.

[14] Frank Mueller and David B. Whalley. Fast instruction cache analysis via static cache simulation. In *Proceedings of the 28th Annual Simulation Symposium*, pages 105–114, 1994. On page 32.

[15] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008. On page 3.

[16] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37:99–122, November 2007. On page 11.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. On pages 35 and 36.

[18] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 203–212, New York, NY, USA, 2007. ACM. On page 33.

[19] Jan Staschulat and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 227–236, Washington, DC, USA, 2006. IEEE Computer Society. On page 32.

[20] Simon Wegener. Improving static analysis of loops. Master's thesis, Universität des Saarlandes, Juni 2011. On page 34.

[21] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, RTAS '97, pages 192–202, Washington, DC, USA, 1997. IEEE Computer Society. On page 32.