Abstract Interpretation with Applications to Timing Validation*

(Invited Tutorial)

Reinhard Wilhelm and Björn Wachter

Universität des Saarlandes, Saarbrücken, Germany
{wilhelm,bwachter}@cs.uni-sb.de

Abstract. Abstract interpretation is one of the main verification technologies besides model checking and deductive verification.
Abstract interpretation has a rich theory of abstraction and strong support for the construction of abstract domains. It allows to express a precise relation to the (concrete) semantics of the programming language inducing a clear relation between the results of an abstract interpretation and the properties of the analyzed program. It permits trading efficiency against precision and offers means to enforce termination where this is not guaranteed.
We explain abstract interpretation using examples from a particular application domain: the determination of bounds on the execution times of programs. These bounds are used to show reliably that hard real-time systems satisfy their timing constraints.
The application domain requires a number of static analyses and domains with different characteristics. Most domains exhibit Galois connections, a few do not. Some analyses require widening to leap infinite ascending chains and ensure termination.

1 Introduction

Abstract interpretation, the theory behind static program analysis, has its roots in the compiler domain. From early on, compilers used static analysis to compute invariants at program points, which would imply the applicability conditions of optimizing transformations. First strong theoretical results about static program analysis were obtained in the 70s [1, 2]. Exactly 30 years ago, Patrick Cousot submitted his PhD thesis [3], which contained the very rich theory of abstract interpretation and new static program analyses. He showed that all static analyses were abstractions of a suitable concrete semantics and hereby opened the way to analyses that could be proved correct or were even correct by construction.

* Work reported herein was partially supported by the European IST Project DAEDALUS, Validation of Critical Software by Static Analysis and Abstract Testing, the German Transregional Collaborative Research Centre AVACS (Automatic Verification and Analysis of Complex Systems) of the Deutsche Forschungsgemeinschaft, the European Networks of Excellence ARTIST2 and ARTIST DESIGN.

Since then, static program analysis has left the compiler domain and has become a verification method in its own right providing means to automatically prove safety properties of programs. Among the most spectacular applications of static program analysis to real systems probably are the analysis of the reasons for the failure of the Ariane5 rocket [4], the proof of the absence of run-time errors in safety-critical avionics code in the ASTRÉE project [5], and our method to determine reliable and precise execution-time bounds for hard real-time systems [6, 7]. Today, static analysis tools based on abstract interpretation [8–12] are widely used in industry and abstract interpretation continues to make inroads into new application domains [13–15].

2 Timing Analysis - the Application Domain

Hard real-time systems are subject to stringent timing constraints which are dictated by the surrounding physical environment. We are concerned with the problem of guaranteeing that all the timing constraints of tasks when executed on a given processor architecture will be met ("timing validation").

Systems show a variability of execution times depending on

the input data: this has always been so and will remain so as it is a property of the algorithm,

the initial execution state: this is caused by modern architectural features such as caches, pipelines, and speculation, and

interference from the environment: preemptions and interrupts.

The unit-time (executing an instruction always takes exactly one time unit) or constant-time abstraction used in many approaches to timing validation is thus rendered obsolete by the advent of modern processors.

In general, the state space of input data and initial states is too large to exhaustively explore all possible executions and so determine the *exact* worst-case and best-case execution times. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions inevitably lose information, and yet must guarantee upper bounds for worst-case and lower bounds for best-case execution time, respectively.

The alternative to exhaustive end-to-end measurement, which was just argued to be infeasible, and non-exhaustive measurement, which is unsound in general because it may underestimate, will be explained in this paper. It consists in the computation of upper (and possibly lower) bounds on the execution times of instructions or basic blocks and the determination of a worst-case path through the program. However, the variability of execution times also appears on the instruction level, even on the level of individual memory accesses and arithmetic operations, but the problem to bound execution times is easier to solve for instructions than for whole programs.

We can look at the execution times of an instruction as an interval, from the best case, e.g. all memory accesses hit the cache, all needed pipeline units are

free, no branch misprediction occurs, etc., to the worst case, where memory accesses miss the cache, pipeline units are occupied, buffers to fetch from are empty and buffers to write to are full, etc. Relative to the best case, we will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases the execution time as compared to the fastest time the *timing penalty* for this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters a timing accident depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history. Excluding timing accidents means decreasing the upper bounds.

The computation of worst-case bounds for a program is realized by first employing an abstract processor model to compute a cycle-level abstract semantics of the program and, in a second phase, mapping resulting time bounds for program portions to an Integer Linear Program (ILP) whose optimal solution yields the final bound. This tool architecture has been successfully used to determine precise upper bounds on the execution times of real-time programs running on processors used in embedded systems [16, 17, 6, 7, 18]. A commercially available tool, `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, was implemented and is used in the aeronautics and automotive industries.

In this tutorial, we deal with the more compute-intensive first phase. It has the following three constituents, which we will treat in more detail later:

1. Value analysis attempts to compute information about data accesses and control flow, in particular it tries to identify infeasible paths, syntactically possible paths that will never be taken because of contradictory conditions.
2. Cache-behavior prediction determines a safe and concise approximation of the contents of caches in order to classify memory accesses as definite cache hits or misses.
3. Pipeline-behavior prediction analyzes how instructions pass through the pipeline taking cache-hit or miss information into account. The cache-miss penalty is assumed for all cases where a cache hit cannot be guaranteed.
   At the end of simulating one instruction, a certain set of final states has been reached. The pipeline analysis starts the analysis of the next instruction in all those states.

Most powerful microprocessors have so-called *timing anomalies*. These are counter-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole program. The interaction of several processor features can cause a locally faster execution of an instruction to lead to a globally longer execution time of the whole program. For example, a cache miss contributes the cache-miss penalty to the execution time of a program. It was, however, observed for the MCF 5307 [18] that a cache miss may actually speed up program execution if it prevents a costly branch misprediction. The existence of timing anomalies forces the analysis to consider a rather large search space since it has to follow not only the local worst-case transitions in the architecture.

3 Abstract Interpretation

This section describes crucial program analyses in the context of timing validation. In Subsection 3.1, we introduce the theoretical foundations before we describe constant propagation in 3.2, interval analysis in 3.3, cache analysis in 3.4 and pipeline analysis in 3.5.

3.1 The Theory

Program. A program is represented by a control flow graph which consists of a set of program points V, an initial location v_{in} (models program entry), and a set of labeled control flow edges $E \subseteq V \times Op \times V$ (the elements of Op model the operation that is executed when the edge is taken).

A program semantics consists of a (possibly infinite) set S of program states, a set of initial states $S_0 \subseteq S$ and a semantics function $[\![.]\!] : Op \to (S \to S)$ that assigns to each operation and thus to each control flow edge, a transfer function modeling its effect on the current program state.

Concerning the operations and the semantics, we observe that the execution-time bounds of a program cannot be determined from the source code of a high-level language like C: executable code has to be analyzed.

For readability we employ an imperative toy language rather than an assembly language to explain the first two example analyses. We will only regard assignment statements, $x \leftarrow e$ and the labels of the two outgoing edges of conditionals, $\mathsf{true(e)}$ and $\mathsf{false(e)}$, for a condition e. As a semantic domain, it uses states assigning integer values to variables, $\rho : \mathit{Vars} \to \mathbb{Z}$. A statement op transforms the state ρ. The semantics of the operations of the toy language is defined by:

$$
\begin{array}{ll}
[\![\mathsf{true}\,(e)]\!]\,\rho = \rho & \text{if } [\![e]\!]\,\rho = \mathtt{tt} \\
[\![\mathsf{false}\,(e)]\!]\,\rho = \rho & \text{if } [\![e]\!]\,\rho = \mathtt{ff} \\
[\![x \leftarrow e]\!]\,\rho = \rho \oplus \{x \mapsto [\![e]\!]\,\rho\} &
\end{array}
$$

Compared to our toy language, executables have a different concept of "variables" as they employ registers defined in the instruction set architecture (ISA). Note that the ISA is still somewhat machine independent, e.g. the PowerPC architecture has many implementations for which the same value analysis can be used. Machine-dependent semantics for cache and pipeline behavior prediction are discussed in Subsection 3.4 and Subsection 3.5, respectively.

Collecting Semantics. The collecting semantics of a program assigns to each program point the set of states that may occur at it during some execution. The collecting semantics is expressible as the fixed point of a set of recursive equations and is, in general, not computable and, even in the finite-state case, not efficiently computable. To this end, the program analyses presented here compute a safe over-approximation of the collecting semantics of a program by computing a fixed point in a simpler domain.

The collecting semantics $\mathbb{S} : V \to 2^S$ is defined by the least fixpoint $lfp(F) = F^*(\lambda v.\emptyset)$ of the functional $F : (V \to 2^S) \to (V \to 2^S)$:

$$F(f)(v') = \begin{cases} S_0 & \text{if } v' = v_{in} \\ \bigcup\limits_{(v,op,v') \in E} [\![op]\!](f(v)) & \text{otherwise} \end{cases} .$$

Program Analysis. A program analysis $\mathbb{A} = (D, [\![.]\!]^\sharp)$ consists of an abstract domain D and an abstract semantics $[\![.]\!]^\sharp$.

An *abstract domain* $D = (S, A, \beta, \gamma)$ is defined by a complete semi-lattice $A = (A, \sqsubseteq, \bigsqcup, \bot, \top)$, a representation function $\beta : S \to A$, mapping concrete to abstract states, and a concretization $\gamma : A \to 2^S$, mapping abstract states to the set of concrete states they represent. Concretization and representation function are required to be montone functions with respect to set inclusion and $\sqsubseteq$, respectively, and must be consistent to each other, i.e. the representation of a concrete state s must concretize to a set of states containing that concrete state, i.e. $s \in \gamma(\beta(s))$.

We define an abstraction function $\alpha : 2^S \to A$ by $\alpha(S') = \bigsqcup\{\beta(s) \mid s \in S'\}$. Given a lattice and a concretization, there may be a plethora of admissible representation functions with varying precision that lead to a domain, e.g. one could map some or all values to $\top$. To formalize the notion of optimal precision at the level of the domain, the concept of a *Galois connection* was introduced. If the concretization and the abstraction fulfill the condition $\alpha(X) \sqsubseteq a \Leftrightarrow X \subseteq \gamma(a)$, we shall call the pair (α, γ) a *Galois connection*.

The *abstract semantics* $[\![.]\!]^\sharp : Op \to (A \to A)$ assigns abstract transfer functions $[\![op]\!]^\sharp : A \to A$ to statements. We impose two requirements, first, the transfer functions are monotone with respect to $\sqsubseteq$ and, second, they approximate (or even equal) the best abstract transfer function $[\![op]\!]^\sharp_{\text{best}}(a) = \alpha([\![op]\!](\gamma(a)))$, i.e. $[\![op]\!]^\sharp_{\text{best}} \sqsubseteq [\![op]\!]^\sharp$ (or $[\![op]\!]^\sharp = [\![op]\!]^\sharp_{\text{best}}$).

The *program analysis problem* is to compute invariants $\mathbb{S}^\sharp : V \to A$ (in terms of the abstract domain) for all program points v such that $\mathbb{S}(v) \subseteq \gamma(\mathbb{S}^\sharp(v))$. This is solved by computing the fixpoint $lfp(F^\sharp) = F^{\sharp *}(\lambda v.\bot)$ of the functional $F^\sharp : (V \to A) \to (V \to A)$:

$$F^\sharp(f)(v') = \begin{cases} l_0 & \text{if } v' = v_{in} \\ \bigsqcup\limits_{(v,op,v') \in E} [\![op]\!]^\sharp(f(v)) & \text{otherwise} \end{cases}$$

where the initial abstract state is chosen such that $\alpha(S_0) \sqsubseteq l_0$.

Termination. The transfer functions are required to be monotone, so that in each fixpoint iteration the values at the program points do not decrease with respect to $\sqsubseteq$. Nontermination can only occur if the lattice exhibits infinite ascending chains, i.e. sequences $a_1, a_2, a_3, \ldots$ of distinct elements with increasing order $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \ldots$. Then widening is used [19, 20] to enforce termination of fixpoint iteration. A widening operator accumulates (monotonely) increasing or

decreasing values in such a way that each variable in the system of equations will only be changed finitely many times. This will guarantee termination albeit at the cost of a loss of precision. Widening for numerical domains has received a lot of attention, e.g. [21, 22].

In this context, we discuss the interval domain (see 3.3), a simple and yet very useful numerical domain with infinite ascending chains.

3.2 Constant Propagation

Constant propagation attempts to find out for each program point which variables have which constant values whenever execution reaches that point. The resulting information can be used to *fold* (sub-)expressions and conditions, i.e., compute their values at compile time.

The abstract domain of constant propagation is constructed in two steps; we first define a partial order for the potential values of variables, the domain:

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \qquad \text{and} \quad x \sqsubseteq_{\mathbb{Z}^\top} y \quad \text{iff } y = \top \text{ or } x = y$$

where $\top$ is an extension of the set of integer values used to denote that the value of a variable is unknown.

The representation function maps integers to the corresponding element in $\mathbb{Z}^\top$, i.e. $\beta_{\mathbb{Z}^\top}(z) = z$. The abstraction function takes subsets $M \subseteq \mathbb{Z}$ of the integers as arguments; it maps singleton sets to their element and all other sets to unknown:

$$\alpha_{\mathbb{Z}^\top}(M) = \begin{cases} z & \text{if } M = \{z\} \\ \top & \text{otherwise} \end{cases} .$$

The concretization is defined by

$$\gamma_{\mathbb{Z}^\top}(\top) = \mathbb{Z} \qquad \text{and} \quad \gamma_{\mathbb{Z}^\top}(z) = \{z\} \quad \text{iff } z \neq \top .$$

In a second step, we lift the abstraction for values to an abstraction of *variable bindings* (states) and consider the complete lattice:

$$A = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\bot\}$$

The new element $\bot$ denotes the fact that the analysis has not yet reached this program point. The partial order on this abstract domain, "$\sqsubseteq$", is defined as:

$$D_1 \sqsubseteq D_2 \qquad \text{iff} \qquad \bot = D_1 \quad \text{or} \quad D_1\, x \sqsubseteq_{\mathbb{Z}^\top} D_2\, x \quad \text{for all } x \in \mathit{Vars}$$

An abstract variable binding D_1 is more precise than a binding D_2 if D_1 binds all variables that D_2 also "knows" to the same values, but possibly knows some more values of variables. A together with this partial order is a complete lattice.

The concretization $\gamma(\bot)$ of the bottom element is the empty set of variable bindings, for all other abstract variable bindings D, it is the lifting of the concretization of $\mathbb{Z}^\top$:

$$\gamma(D) = \{s \mid \forall v \in \mathit{Vars} :\ s(v) \in \gamma_{\mathbb{Z}^\top}(D(v))\} .$$

The representation function is given by: $\beta(s)(v) = \beta_{\mathbb{Z}^\top}(s(v))$. The abstraction function maps the empty set to the bottom element, for non-empty sets we lift the abstraction function of $\mathbb{Z}^\top$:

$$\alpha(S')(v) = \alpha_{\mathbb{Z}^\top}(\{s(v) \mid s \in S'\})$$

The transfer functions of statements, $[\![op]\!]^\sharp : A \to A$, simulate the concrete evaluation function. They employ an abstract evaluation function for arithmetic expressions. For a binary operator $\square$, it is defined by:

$$a \,\square^\sharp\, b = \begin{cases} \top & \text{if} \quad a = \top \text{ or } b = \top \\ a \,\square\, b & \text{otherwise} \end{cases}$$

The evaluation function is able to deal with unknown values of variables. It propagates this information; the result is $\top$ if one of the operands is unknown, i.e., is $\top$. The result is the same as in the concrete case for two known operands. The transfer functions of the abstract semantics for the toy language are given in Figure 1.

Fig. 1. The abstract semantics for constant propagation:

$$\begin{aligned} [\![x \leftarrow e]\!]^\sharp D &= D \oplus \{x \mapsto [\![e]\!]^\sharp D\} \\ [\![\mathsf{true}\,(e)]\!]^\sharp D &= \begin{cases} \bot \text{ if } \quad [\![e]\!]^\sharp D = \mathtt{ff} \\ D \text{ otherwise} \end{cases} \\ [\![\mathsf{false}\,(e)]\!]^\sharp D &= \begin{cases} \bot \text{ if } \quad [\![e]\!]^\sharp D = \mathtt{tt} \\ D \text{ otherwise} \end{cases} \end{aligned}$$

If the condition can be definitely evaluated to ff, then the true branch is unreachable, and if it can be definitely evaluated to tt, then the false branch is unreachable. An assignment is analyzed by evaluating the right side in the abstract variable binding and over-writing the binding of the left side with the resulting value, which may be $\top$.

Constant propagation is useful for timing analysis since it transports statically available information to relevant places. The computed information is used by value analysis and control-flow analysis. Though of infinite size the domain is only of finite height, i.e. there are no infinite ascending chains. Further, the presented abstraction and concretization function form a Galois connection.

3.3 Interval Analysis

A static method for data-cache behavior prediction needs to know effective memory addresses of data, in order to determine where a memory access goes. However, effective addresses are only available at run time. Here interval analysis as described by Cousot and Cousot [19] comes into play. It can compute intervals

for address-valued objects like registers and variables. An interval computed for such an object at some program point bounds the set of potential values the object may have when program execution reaches this program point. Such an analysis, as part of aiT's *value analysis*, has been shown to be very effective on disciplined code [7].

Interval analysis generalizes constant propagation by replacing the domain $\mathbb{Z}^\top$ for variables by that of intervals. The interval domain is given by

$$\mathbb{I} = \{[l,u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

Note that this definition admits only intervals that represent non-empty sets of integers. The set of intervals is ordered by "$\sqsubseteq$", defined by

$$[l_1,u_1] \sqsubseteq_{\mathbb{I}} [l_2,u_2] \qquad \text{iff} \qquad l_2 \leq l_1 \wedge u_1 \leq u_2$$

Least upper bound and greatest lower bound of two intervals are defined by:

$$[l_1,u_1] \sqcup [l_2,u_2] = [\mathsf{min}\{l_1,l_2\}, \mathsf{max}\{u_1,u_2\}]$$
$$[l_1,u_1] \sqcap [l_2,u_2] = [\mathsf{max}\{l_1,l_2\}, \mathsf{min}\{u_1,u_2\}], \text{ if } \mathsf{max}\{l_1,l_2\} \leq \mathsf{min}\{u_1,u_2\}$$

The representation function maps an integer to a singleton interval: $\beta_{\mathbb{I}}(z) = [z,z]$ and the abstraction function maps a subset $M \subseteq \mathbb{Z}$ of the integers to an interval with the infimum and supremum of M as endpoints: $\alpha_{\mathbb{I}}(M) = [\inf_{z \in M} z, \sup_{z \in M} z]$. The concretization function relates concrete values and intervals:

$$\gamma_{\mathbb{I}}([l,u]) = \{z \in \mathbb{Z} \mid l \leq z \leq u\} \ .$$

Analogous to constant propagation, the numerical abstraction for variable values is lifted to an abstraction of variable bindings (states), i.e. we consider the complete lattice with elements $(Vars \to \mathbb{I})_\perp$.

To obtain an abstract semantics, some arithmetic on intervals is defined, first the sum of two intervals: $[l_1,u_1] +^\sharp [l_2,u_2] = [l_1+l_2, u_1+u_2]$ where $-\infty + \_ = -\infty$ and $+\infty + \_ = +\infty$ (the underscore $\_$ stands for "any value"). For unary minus, we define: $-^\sharp [l,u] = [-u,-l]$. Multiplication on intervals is more involved and division yet more difficult. For this and comparisons on intervals, we refer to [23]. The abstract semantics is the same as the one for constant propagation (cf. Figure 1) *except* that it uses the expression evaluation function for intervals.

Achieving termination of interval analysis requires some extra work because the partially ordered domain $\mathbb{I}$, as opposed to $\mathbb{Z}^\top$, exhibits infinitely ascending chains, e.g.

$$[0,0] \sqsubset [0,1] \sqsubset [0,2] \sqsubset [-1,2] \sqsubset \ldots$$

and so does the lifted lattice $(Vars \to \mathbb{I})_\perp$. In order to enforce termination of fixpoint iteration, widening is used [19, 20]. In our present interval analysis, any increasing upper bound of an interval will immediately be set to ∞, any decreasing lower bound of an interval to $-\infty$.

Interval analysis produces relevant information for cache analysis. The smaller the intervals bounding potential data addresses, the more precise are the results of cache analysis. The presented abstraction and concretization function form a Galois connection.

3.4 Cache Analysis

Abstract interpretation is also used to compute invariants about cache contents at all program points.

For brevity, we restrict our description to the semantics of fully associative caches with LRU replacement strategy. We refer to [24, 17] for descriptions of how to deal with direct-mapped and A-way set associative caches.

In the following, we consider a (fully associative) cache as a set of cache lines $L = \{l_1, \ldots, l_n\}$ and the store as a set of memory blocks $M = \{m_1, \ldots, m_k\}$. To indicate the absence of any memory block in a cache line, we introduce a new element I; $M' = M \cup \{I\}$. A *(concrete) cache state* is a function $c : L \to M'$. C denotes the set of all concrete cache states. The initial cache state c_I maps all cache lines to I. If $c(l) = m_i$ for a concrete cache state c, then i is the relative age of the memory block.

The cache *update* function $\mathcal{U} : C \times M \to C$ determines the new cache state for a given cache state and a referenced memory block. The LRU (Least-Recently Used) strategy always makes the referenced memory block the youngest, i.e. the referenced memory block moves into l_1 if it was in the cache already (cache hit). All memory blocks in the cache that had been used more recently than the referenced block increase their relative age by one, i.e., they are shifted by one position to the next cache line. If the referenced memory block was not yet in the cache (cache miss), it is loaded into l_1 after all memory blocks in the cache have been shifted and the 'oldest', i.e., least recently used memory block, has been removed from the cache if the cache was full. This is depicted in Figure 2.

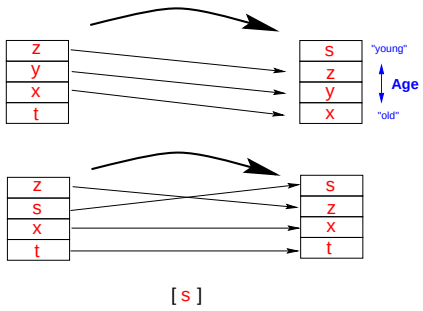

Fig. 2. Update of a concrete cache.

In our present exposition, we assume that for each basic block, the sequence of references to memory is known, i.e., there exists a mapping from operations to sequences of memory blocks: $\mathcal{L} : Op \to M^*$. This is realistic for instruction caches. For data caches, only intervals may be available. The techniques described here are also routinely applied to data caches. The slight adaptions necessary to handle adress intervals can be found in [16].

We can describe the effect of such a sequence on a cache with the help of the update function $\mathcal{U}$. Therefore, we extend $\mathcal{U}$ to sequences of memory references by sequential composition: $\mathcal{U}(c, \langle m_{x_1}, \ldots, m_{x_y} \rangle) = \mathcal{U}(\ldots(\mathcal{U}(c, m_{x_1}))\ldots, m_{x_y})$. The cache semantics of an operation op at a control-flow edge is then $[\![op]\!] = \mathcal{U}(\cdot, \mathcal{L}_{op})$.

The collecting semantics would be computable, although often of enormous size. Therefore, another step abstracts it into a compact representation, so called abstract cache states. Note that every information drawn from the abstract cache

states allows to safely deduce information about sets of concrete cache states, i.e., only precision may be reduced in this two step process. Correctness is guaranteed.

The abstraction consists in two analyses one computes an under- and the other an overapproximation of the cache content as follows: To classify definite cache hits, the *must analysis* determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. To classify definite misses, a *may analysis*, not described in this paper, determines all memory blocks that may be in the cache at a given program point.

The domains for the must analysis (and also the may analysis) consist of *abstract cache states*: An *abstract cache state* $c^\sharp : L \to 2^M$ maps cache lines to sets of memory blocks. These sets are disjoint so that each memory block has unique position: it is either in one of the abstract cache lines or it is not in the cache. The position of a memory block in an abstract cache denotes, as in the case of concrete caches, the relative age of the corresponding memory blocks. As explained above, must analysis determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. The positions of the memory blocks in the abstract cache state are thus the upper bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

Good information, in the sense of being valuable for the prediction of cache hits, is the knowledge that a memory block is in the cache. The bigger the set the better. This is connected to the "age" of a memory block. Therefore, the partial order $\sqsubseteq$ is as define follows: Take an abstract cache state $c^\sharp$. Elements that are higher up with respect to $\sqsubseteq$ than $c^\sharp$ in the domain, i.e., less precise, are states where memory blocks from $c^\sharp$ are either missing or are older than in $c^\sharp$. Therefore, the $\bigsqcup$-operator applied to two abstract cache states $c_1^\sharp$ and $c_2^\sharp$ will produce a state $c^\sharp$ containing only those memory blocks contained in both, and will give them the maximum of their ages in $c_1^\sharp$ and $c_2^\sharp$ (see Figure 3). The *representation function* $\beta : C \to C^\sharp$ forms singleton sets from concrete cache states it is applied to, i.e., $\beta(c)(l_i) = \{m_x\}$ if $c(l_i) = c_x$. Concretization of an abstract cache state, $c^\sharp$, produces the set of all concrete cache states, which contain all the memory blocks contained in $c^\sharp$ with ages not older than in $c^\sharp$. Cache lines not filled by these are filled with other memory blocks. The *concretization function* $\gamma : C^\sharp \to 2^c$ is defined by $\gamma(c^\sharp) = \{c \mid \beta(c) \sqsubseteq c^\sharp\}$.

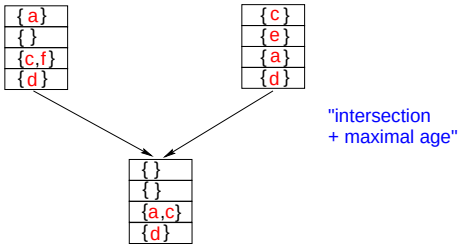

Fig. 3. Join.

The abstract semantics is defined by *abstract cache update* functions, denoted $\mathcal{U}^\sharp$, which describe the effects of a control flow edge on an element of the abstract

domain. An abstract cache update function (example depicted in Figure 4) is a lifted version of the corresponding concrete update function to sets, in that the referenced memory block goes to line l_1, all younger blocks age by one.

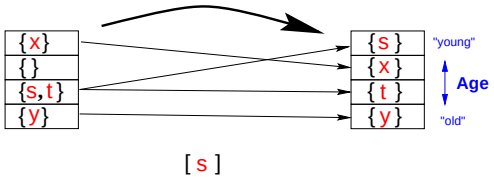

Fig. 4. Update of an abstract cache.

The solution of the must analysis problem is interpreted as follows: Let $c^\sharp$ be an abstract cache state at some program point. If $m_x \in c^\sharp(l_i)$ for a cache line l_i then m_x will definitely be in the cache whenever execution reaches this program point. A reference to m_x is categorized as *always hit* (ah).

Termination. There are only a finite number of cache lines and for each program a finite number of memory blocks. This means, that the domain of abstract cache states $c : L \to 2^M$ is finite. Hence, every ascending chain is finite. Additionally, the abstract cache update functions, $\mathcal{U}^\sharp$, are monotonic. This guarantees that all the analyses will terminate.

The abstract cache-state domain is essential for the efficiency and therefore for the feasibility of cache analysis. The results are precise enough although this type of cache analysis loses information at merge points. More precise analyses are possible. However, experiments have shown that the corresponding analyses are too slow.

The domain for LRU caches fulfills the ascending chain condition and forms a Galois connection. For other cache domains, such as for the popular Pseudo-LRU caches, there does not exist an optimal abstraction function, and hence no Galois connection [18].

3.5 Pipeline Analysis

Most state-of-the-art processors employed in embedded systems today have an instruction pipeline, i.e. the execution of several instructions is overlapped. Instructions simultaneously pass through different pipeline stages: An instruction is first loaded from memory (fetch stage). The duration of this stage is determined by the contents of the instruction cache. The instruction is then ready to be dispatched: it is decoded and operands are fetched. In the execution stage, instructions compete for resources, such as execution units, buses and memory, producing complex interdependences. Depending on the internal state of the processor the time from fetch to completion of an instruction can vary by several orders of magnitude.

Pipeline analysis works on executable programs and is based on an abstract timing model for the specific processor. A timing model is a state machine whose transitions correspond to clock cycles of the modeled processor. Technically, pipeline analysis is a program analysis on the basic block graph[1] that computes for each basic block an invariant on the machine states that can occur at it and an execution time bound for the number of cycles it takes to execute it whenever execution reaches that block. The abstract semantics of a basic block computes from the abstract processor states at entry to the block the set of processor states on exit of the block together with the bound. To this end, the analysis runs the abstract timing model of the processor cycle per cycle. Whenever abstraction produces uncertainty, e.g. inability to classify a cache access as a hit or a miss, the analysis follows all possibilities (both hit and miss case).

The structure of a timing model is determined by the different processor units and its memory system: the pipeline stages, a model of the processor chipset, the bus unit, the branch predictor, register files, and arithmetic units etc. Though structurally similar to the processor, the model concentrates on timing-relevant control components and data, e.g., it is not interested in what an arithmetic instruction computes, but in how many cycles the instruction takes.

Value analysis can be considered as factored-out arithemtic. The pipeline timing model imports the results of the value analsysis. The memory system is concisely abstracted by the chipset unit, bus unit and the cache domain described in the previous section and similarly factored out.

Cache and pipeline analysis are integrated to reflect the interdependences between the caches and the pipeline due to speculation and prefetching [25].

Abstract domain and transfer functions are determined by the timing model. The pipeline analysis for a state-of-the-art processor described in [25] uses the following domains:

- An abstract state of the timing model is a tuple $(p, c^\sharp)$ consisting of the pipeline state p and an abstract cache state $c^\sharp$. It represents a set of concrete states of the timing model.
- From the above domain of tuples, the disjunctive completion is taken. This results in a lattice whose elements are sets of states of the timing model. However, rather than taking set union as a join operator, a more sophisticated join is used that leverages the join operator of the cache domain. It takes a set of sets of processor states as input and produces a set of processor states that overapproximates the union of the input sets. Its number of elements equals the number of distinct pipeline states in the input sets. The join operator loses precision only on the cache side by joining abstract cache states where abstract pipeline state is identical. In the result set, each abstract pipeline state p is adjoined with the *join* of a set of abstract caches, namely the join of the abstract caches $c^\sharp$ such that $(p, c^\sharp)$ appears in one of the input sets:

$$\bigsqcup\{S_i \mid i = 1, ..., k\} = \{(p, c) \mid \exists i.(p, c') \in S_i \wedge c = \bigsqcup\{c'' \mid (p, c'') \in S_j\}\}$$

[1] A basic block is a maximal sequence of straight-line code in the program.

– The abstract domain results as the product of the semi-lattice of natural numbers with the maximum as join operator and the lattice of timing model states. The join operator is defined by:

$$\bigsqcup\{(n_i, S_i) \mid i = 1, ..., k\} = (\max_i n_i, \bigsqcup\{S_i \mid i = 1, ..., k\}$$

An abstract state is a tuple: the first component is a time bound (a number) and the second component a *set of states* of the timing model.

To evaluate the transfer function $[\![op]\!]^\sharp(a)$ for a control flow edge (b, op, b') and a tuple $(., S)$ (the first component is ignored), a finite transition system is computed. Its initial states are all those states $I \subseteq S$ that load at least one instruction of basic block b. The transitions are determined by the timing model. Its final states F are those in which all instruction within basic block b have been completed. The transition system is acyclic and finite. Let k be the maximal length of a path from initial to the final states, then $[\![op]\!]^\sharp(a) = (k, F)$.

4 Related Approaches

Timed automata [26] (or networks thereof) have been used to express timing constraints of real-time systems and require durations and time bounds. Timing analysis can deliver such bounds in the form of lower and upper bounds on the execution time for a realistic architecture.

Campos et al. [27–29] leverage finite-state BDD-based model checking for timing analysis. This work is not comparable with the approach proposed in this tutorial since results were only obtained for highly simplified architectures without typical features of modern processors such as caches and pipelining.

The works by Logothetis, Schuele and Schneider [30–32] describe timing analyses of assembler programs using symbolic simulation. This work remains at the machine-independent level and is based on the unit-time assumption.

Metzner [33] proposed to use BDD-based model checking for cache behavior prediction instead of abstract interpretation and reported some gain (1.5-5%) in precision over cache analysis by abstract interpretation [34] because joins at control-flow merge points are avoided. The experiments considered a very small cache and an extremely simple pipeline. Scalability of the analysis to industrial-scale benchmarks was not shown. Furthermore, the experimental results are limited to instruction caches for which cache analysis is easier than for data caches because the addresses are statically known and access patterns are more regular.

5 Conslusions

A short introduction into the theory of abstract interpretation was given, and several instances of abstract interpretations are described that are used in timing analysis. The different analyses have quite different characteristics. Constant propagation, interval analysis, and cache analysis live on the design of the right

abstract domain. They represent sets of concrete values by single abstract values. Further, while cache and pipeline analysis employ a finite domain, constant propagation and interval analysis exhibit infinite domains, yet only the interval domain has infinite ascending chains and requires widening for termination.

For pipeline analysis, a suitable representation of sets of concrete pipeline states by single abstract states has not been found and is probably hard to find. Pipeline analysis is not a typical static program analysis. It can rather be seen as a hybrid: it employs both state traversal of the pipeline evolution and join operations typical for static analysis.

References

1. Kildall, G.A.: A Unified Approach to Global Program Optimization. In: Proceedings of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts (October 1973) 194–206
2. Kam, J., Ullman, J.D.: Monotone Data Flow Analysis Frameworks. Acta Informatica **7**(3) (1977) 305–318
3. Cousot, P.: Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treilis, analyse sémantique des programmes. PhD thesis, Université de Grenoble (1978)
4. Lacan, J., Monfort, J.N., Ribal, V.Q., Deutsch, A., Gonthier, G.: The software reliability verification process: The Ariane 5 example. DAta Systems In Aerospace (1998) SP-422.
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), San Diego, California, USA, ACM Press (June 7–14 2003) 196–207
6. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: EMSOFT. Volume 2211 of LNCS. (2001) 469 – 485
7. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software systems. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003), IEEE Computer Society (June 2003) 625–632
8. Mathworks: Polyspace http://www.polyspace.com.
9. Coverity: Coverity Prevent http://www.coverity.com.
10. Fasoo.com, Seoul University: Sparrow http://www.polyspace.com.
11. GrammaTech Inc.: Codesurfer http://www.grammatech.com.
12. AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers
13. Nielson, F., Nielson, H.R., da Rosa, D.S., Priami, C.: Static analysis for systems biology. In: Proc. of workshop on Systeomatics - dynamic biological systems informatics, Computer Science Press, Trinity College Dublin (2004)
14. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security **13**(3) (2005) 347–390
15. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking Abstractions. In: Proceedings of the 17th European Symposium on Programming. (April 2008) To appear.

16. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. In: Proceedings of SAS'96, Static Analysis Symposium. Volume 1145 of Lecture Notes in Computer Science., Springer-Verlag (September 1996) 52–66
17. Ferdinand, C., Martin, F., Wilhelm, R.: Cache Behavior Prediction by Abstract Interpretation. Science of Computer Programming **35** (1999) 163 – 189
18. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. IEEE Proceedings on Real-Time Systems **91**(7) (2003) 1038–1054
19. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California (1977) 238–252
20. Cousot, P., Cousot, R.: Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL'91, Bordeaux. BIGRE **74** (October 1991) 107–110
21. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. Sci. Comput. Program. **58**(1-2) (2005) 28–56
22. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. **19**(1) (2006) 31–100
23. Wilhelm, R., Seidl, H.: Übersetzerbau: Analyse und Transformation. Springer (2008)
24. Ferdinand, C.: Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes (September 1997)
25. Thesing, S.: Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models. PhD thesis, Saarland University (2004)
26. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2) (1994) 183–235
27. S. Campos, O. Grumberg: Selective quantitative analysis and interval model checking: Verifying different facets of a system. In Rajeev Alur, Thomas A. Henzinger, eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., New Brunswick, NJ, USA, Springer Verlag (/ 1996) 257–268
28. Campos, S.V.A., Clarke, E.M.: Analysis and verification of real-time systems using quantitative symbolic algorithms. International Journal on Software Tools for Technology Transfer **2**(3) (1999) 260–269
29. Hartonas-Garmhausen, V., Campos, S.V.A., Cimatti, A., Clarke, E.M., Giunchiglia, F.: Verification of a safety-critical railway interlocking system with real-time constraints. Sci. Comput. Program. **36**(1) (2000) 53–64
30. Schüle, T., Schneider, K.: Exact runtime analysis using automata-based symbolic simulation. In: MEMOCODE. (2003) 153–162
31. Schüle, T., Schneider, K.: Abstraction of assembler programs for symbolic worst case execution time analysis. In: DAC. (2004) 107–112
32. Logothetis, G., Schneider, K.: Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: DATE. (2003) 10196–10203
33. Metzner, A.: Why model checking can improve WCET analysis. In: CAV. (2004) 334–347
34. Ferdinand, C., Wilhelm, R.: Fast and Efficient Cache Behavior Prediction for Real-Time Systems. Real-Time Systems **17** (1999) 131 – 181