Timing Analysis of Parallel Software Using Abstract Execution

Andreas Gustavsson, Jan Gustafsson, and Björn Lisper

Mälardalen University, Västerås, Sweden
{andreas.sg.gustavsson,jan.gustafsson,bjorn.lisper}@mdh.se

Abstract. A major trend in computer architecture is multi-core processors. To fully exploit this type of parallel processor chip, programs running on it will have to be parallel as well. This means that even hard real-time embedded systems will be parallel. Therefore, it is of utmost importance that methods to analyze the timing properties of parallel real-time systems are developed.
This paper presents an algorithm that is founded on abstract interpretation and derives safe approximations of the execution times of parallel programs. The algorithm is formulated and proven correct for a simple parallel language with parallel threads, shared memory and synchronization via locks.

Keywords: WCET, Parallelism, Multi-core, Abstract interpretation, Abstract execution

1 Introduction

A *real-time* system is a system for which the timing behavior is of great importance. *Hard* real-time systems are such that failure to produce the computational result within certain timing bounds could have catastrophic consequences. One example of a hard real-time system is the airbag system in automotive vehicles, another is the control system in airplanes.

A major trend in computer hardware design is *multi-core* processors. The processor cores on such a chip typically share some resources, such as some level of on-chip cache memory, which introduces dependencies and conflicts between the cores. Processor chips of this kind are already (and will, in the future, be even more extensively) incorporated in real-time systems.

To fully utilize the multi-core architecture, algorithms will have to be parallelized over multiple tasks (e.g. threads). This means that the tasks will have to share resources and communicate and synchronize with each other. There already exist software libraries for explicitly parallelizing sequential code automatically. One example of such a library available for C/C++ and Fortran code running on shared-memory machines is OpenMP [1]. The conclusion is that parallel software running on parallel hardware is already available today and will probably be the standard way of computing in the future, also for real-time systems. Thus, it is of crucial importance that methods to derive safe estimations on the lower and upper bounds of the execution times (also referred to as the Best-Case and Worst-Case Execution Times – BCET and WCET– respectively: see [2]) of parallel systems are derived.

This paper presents a novel method that derives safe estimations on the timing bounds for parallel software. The method mainly targets hard real-time systems but can be applied to any computer system that can be modeled using the presented method. More specifically, the main contributions of this paper are the following.

1. A formally defined parallel programming language (PPL) with shared memory, locks, and a timing model.
2. An algorithm that derives safe approximations of the BCET and WCET of PPL programs.

The rest of the paper is organized as follows. Section 2 describes the ideas behind abstract execution for sequential programs. Section 3 presents some research related to the method presented in this paper. Section 4 presents PPL, a parallel programming language. Section 5 abstractly interprets the semantics of PPL. Section 6 presents an algorithm that abstractly executes PPL programs to find safe approximations of their timing behaviors. Section 7 uses the presented algorithm to derive safe bounds on the BCET and WCET for an example PPL program given a simple timing model. Section 8 concludes the paper and presents directions for future research.

2 Abstract Execution for Sequential Programs

Abstract execution (AE) [3, 4] was originally designed as a method to derive program flow constraints on imperative sequential programs, like bounds on the number of iterations in loops and infeasible program path constraints. This information can be used by a subsequent *WCET analysis* [2] to compute a safe WCET bound. AE is based on abstract interpretation, and is basically a very context sensitive value analysis which can be seen as a form of symbolic execution [3]. The program is hence executed in the abstract domain; i.e. abstract versions of the program operators are executed and the program variables have abstract values (which thus correspond to sets of concrete values).

The main difference between AE and a traditional value analysis is that in the former, an abstract state is not calculated for each program point. Instead, the abstract state is propagated on transitions in a way similar to the concrete state for concrete executions of the program. Note that since values are abstracted, a state can propagate to several new states on a single transition (e.g. when both branches of a conditional statement could be taken given the abstract values of the program variables in the current abstract state). Therefore, a worklist algorithm that collects all possible transitions is needed to safely approximate all concrete executions. There is a risk that AE does not terminate (e.g. due to an infinite loop in the analyzed program): however, if it terminates then all final states of the concrete executions have been safely approximated [3]. Furthermore, nontermination can be completely dealt with by setting a timeout, e.g. as an upper limit on the number of abstract transitions.

If timing bounds on the statements of the program are known, then AE is easily extended to calculate BCET and WCET bounds by treating time as a regular program variable that is updated on each state transition – as with all other variables, its set of possible final values is then safely approximated when the algorithm terminates [5].

The approach used in this paper is to calculate safe BCET and WCET estimations by abstract execution of the analyzed program. The timing bounds are derived based on a safe timing model of the underlying architecture.

3 Related Work

WCET-related research started with the introduction of timing-schemas by Shaw in 1989 [6]. Shaw presents rules to collapse the CFG (Control Flow Graph) of a program until a final single value represents the WCET. An excellent overview of the field of WCET research was presented by Wilhelm et al. in 2008 [2]. The field of WCET analysis for parallel software is quite new, so there is no solid foundation of previous research.

Model-checking has been shown adequate for timing analysis of small parts of single-core systems [7, 8]. There are also attempts to analyze parallel systems using model-checking [9–11]. However, complexity matters is a common big issue for these attempts.

This paper uses a more approximate approach (abstract execution). If analyzing a program consisting of only one thread, the method presented in this paper becomes comparable to the methods presented by Gustafsson et al. [4] and Ermedahl et al. [5]. An early version of the analysis presented here [12] could analyze a subset of PPL (without locks); the version here can analyze any PPL program.

There are several other approaches toward WCET analysis of parallel and concurrent programs that are not defined based on abstract execution. Mittermayr and Blieberger [13] use a graph based approach and Kronecker algebra to calculate an estimation on the WCET of a concurrent program. Potop-Butucaru and Puaut [14] target static timing analysis of parallel processors where "channels" are used to communicate between, and synchronize, the parallel tasks. The goal of this approach is to enable the use of the traditional abstract interpretation techniques for sequential software when analyzing parallel systems. Ozaktas et al. [15] focus on analyzing synchronization delays experienced by tasks executing on time-predictable shared-memory multi-core architectures.

The work presented in these publications targets parallel systems with quite specific restrictions, whereas our analysis targets general parallel systems. We focus on analyzing parallel systems on code level, where the underlying architecture could be sequential or parallel, bare metal or an operating system. The only assumption is that the temporal behavior of the underlying architecture, and thus of the threads in the analyzed program, can be safely approximated.

4 PPL: A Parallel Programming Language

In this section, a rudimentary, parallel programming language, PPL, whose semantics includes timing behavior, will be presented. The purpose of PPL is to put focus on communication through shared memory and synchronization on shared resources.

The parallel entities of execution will be referred to as threads. PPL provides both thread-private and globally shared memory, referred to as registers, $r \in \mathrm{Reg}$, and variables, $x \in \mathrm{Var}$, respectively. Arithmetical operations etc. within a thread can be performed using the values of the thread's registers. PPL also provides shared resources, referred to as locks, $lck \in \mathrm{Lck}$, that can be acquired in a mutually exclusive manner by the threads. The operations (statements) provided by the instruction set may have variable execution times. (C.f. multi-core CPUs, which have both local and global memory, a shared memory bus and atomic, i.e. mutually exclusive, operations.) Note that Reg, Var and Lck are finite sets of identifiers that are specific to each defined PPL program.

The syntax of PPL, which is a set of operations using the discussed architectural features, is defined in Fig. 1. Π denotes a program, which simply is a (constant and finite) set of threads, i.e. $\Pi = \mathrm{Thrd}$, where each thread, $\mathrm{T} \in \mathrm{Thrd}$, is a pair of a unique identifier, $d \in \mathbb{Z}$, and a statement, $s \in \mathrm{Stm}$. This makes every thread unique and distinguishable from other threads, even if several threads consist of the same statement. The axiom-statements (all statements except the sequentially composed statement, $s_1 \, ; s_2$) of each thread are assumed to be uniquely labeled with consecutive labels, $l \in \mathbb{Z}$. $a \in \mathrm{Aexp}$ and $b \in \mathrm{Bexp}$ denote an arithmetic and a boolean expression, respectively, and $n \in \mathbb{Z}$ is an integer value.

$$
\begin{aligned}
\Pi &::= \{\mathrm{T}_1, \ldots, \mathrm{T}_m\} \\
\mathrm{T} &::= (d, s) \\
s &::= [\texttt{halt}]^l \;\big|\; [\texttt{skip}]^l \;\big|\; [r := a]^l \;\big|\; [\texttt{if}\ b\ \texttt{goto}\ l']^l \;\big|\; [\texttt{store}\ r\ \texttt{to}\ x]^l \;\big|\; \\
&\qquad [\texttt{load}\ r\ \texttt{from}\ x]^l \;\big|\; [\texttt{unlock}\ lck]^l \;\big|\; [\texttt{lock}\ lck]^l \;\big|\; s_1 \, ; s_2 \\
a &::= n \;\big|\; r \;\big|\; a_1 \texttt{+} a_2 \;\big|\; a_1 \texttt{-} a_2 \;\big|\; a_1 \texttt{*} a_2 \;\big|\; a_1 \texttt{/} a_2 \\
b &::= \texttt{true} \;\big|\; \texttt{false} \;\big|\; \texttt{!}b \;\big|\; b_1 \,\texttt{\&\&}\, b_2 \;\big|\; a_1 \texttt{==} a_2 \;\big|\; a_1 \texttt{<=} a_2
\end{aligned}
$$

Fig. 1. Syntax of the parallel programming language, PPL

The semantic state of a program is described by a configuration, c, defined as $\langle [\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^a_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \mathbb{x}, \mathbb{l} \rangle$. The notation $[\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^a_{\mathrm{T}}]_{\mathrm{T} \in \{\mathrm{T}_1, \ldots, \mathrm{T}_m\}}$ expands to $\langle \mathrm{T}_1, pc_{\mathrm{T}_1}, \mathbb{r}_{\mathrm{T}_1}, t^a_{\mathrm{T}_1}, \ldots, \mathrm{T}_m, pc_{\mathrm{T}_m}, \mathbb{r}_{\mathrm{T}_m}, t^a_{\mathrm{T}_m} \rangle$. This notation is needed since the number of threads in a program is not known before the program is defined.

pc_{T} is a program counter, pointing to the current statement in T. Note that the tuple $\langle pc_{\mathrm{T}_1}, \ldots, pc_{\mathrm{T}_m}\rangle$, assuming that $\mathrm{Thrd} = \{\mathrm{T}_1, \ldots, \mathrm{T}_m\}$, defines a unique program point. $\mathbb{r}_{\mathrm{T}}$ is a mapping from T's registers to their values. t^a_{T} is the accumulated execution time of T. $\mathbb{x}$ is a mapping from variables and threads to a set of timestamped values. $\mathbb{l}$ is a mapping from locks to their states. The state for a lock is a tuple containing information on (in the following order) whether the lock is acquired or not, which thread owns it, a deadline for when the lock must be acquired by the owning thread, the previous owner, and when it was last released. (If the reader finds the variable and lock domains peculiar, the need for their definitions will become clear in Sects. 5 and 6.) The BCET and WCET for a set of configurations are given in Definition 1.

Definition 1. *Given a set of configurations, C, the* BCET *and* WCET *for that set are defined as:*

$$\begin{cases} \mathrm{BCET} ::= \min(\{\max(\{t^a_{\mathrm{T}} \mid \mathrm{T} \in \mathrm{Thrd}\}) \mid \langle [\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \mathbb{x}, \mathbb{l}\rangle \in C\}) \\ \mathrm{WCET} ::= \max(\{\max(\{t^a_{\mathrm{T}} \mid \mathrm{T} \in \mathrm{Thrd}\}) \mid \langle [\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \mathbb{x}, \mathbb{l}\rangle \in C\}) \end{cases}$$

The semantics of transitions between configurations is described by $\xrightarrow[prg]{}$ as defined in Fig. 2. *exp₁* ? *exp₂* : *exp₃* is *exp₂* if *exp₁*, and *exp₃* otherwise. $\lambda p \in P.exp(p)$ is a function from p (an element of P) to $exp(p)$. STM gives the current statement of the issuing thread. TIME gives a relative execution time for the current statement of the calling thread (the definition of this function is out of this paper's scope, but it is assumed to be non-negative). Given some lock state mapping and some lock, OWN gives the owner of the lock (which is $\bot_{thrd}$ iff the lock is free; note however that Thrd is not a complete lattice), POWN gives the previous owner of the lock, REL gives the time at which the lock was last released. Note that similar functions can be defined to mask out the current state – *taken* or *free* – and the lock owner assignment deadline [16].

$\xrightarrow[ax]{}$ (whose formal definition is omitted due to space limitations; see [16]) describes the semantics of a single statement within a thread when considered in isolation from other threads: `halt` stops the execution of the issuing thread (i.e. none of the input states are changed), `halt` must be the last statement of each thread in the program, but could also occur anywhere "within" a thread; `skip` performs a no-operation (i.e. it only increments the thread's program counter); a register is assigned a value using := (the semantics of evaluating arithmetic and boolean expressions are defined in the standard fashion [16, 17] and will not be further discussed); conditional branching to an arbitrary axiom-statement is performed using `if` (thus, `if` is used when e.g. implementing loops); `store` makes the thread's set of timestamped values for the given variable consist only of a tuple consisting of the value of the given register and the value of $t^{a\prime}_{\mathrm{T}}$ (i.e. t); `load` takes one of the stored timestamped values for the given variable (after any `store`, there is only one such value for the given variable since only one of the values stored to a variable by a set of threads is saved; c.f. Fig. 2) and puts the value into the given register; `unlock` releases the given lock (i.e. sets the lock's state to *free*, its owner to $\bot_{thrd}$, its previous owner to the issuing

$$\frac{\mathrm{Thrd}_{exe} \neq \emptyset \wedge \forall \mathrm{T} \in \mathrm{Thrd}_{exe} : \langle \mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, \mathbb{x}, \mathbb{l}'', t^{a\prime}_{\mathrm{T}} \rangle \xrightarrow[ax]{} \langle pc'_{\mathrm{T}}, \mathbb{r}'_{\mathrm{T}}, \mathbb{x}'_{\mathrm{T}}, \mathbb{l}'_{\mathrm{T}} \rangle}{\begin{array}{l} c = \langle [\mathrm{T}, pc_{\mathrm{T}}, \mathbb{r}_{\mathrm{T}}, t^{a}_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \mathbb{x}, \mathbb{l} \rangle \xrightarrow[prg]{} \\ \quad c' = \langle [\mathrm{T}, (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ pc'_{\mathrm{T}} : pc_{\mathrm{T}}), (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ \mathbb{r}'_{\mathrm{T}} : \mathbb{r}_{\mathrm{T}}), t^{a\prime}_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \mathbb{x}', \mathbb{l}' \rangle \end{array}}$$

where

$$t = \min(\{ t^{a}_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) \mid \mathrm{T} \in \mathrm{Thrd} \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\texttt{halt}]^{pc_{\mathrm{T}}} \})$$

$$\mathrm{Thrd}_{exe} = \{ \mathrm{T} \in \mathrm{Thrd} \mid t = t^{a}_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\texttt{halt}]^{pc_{\mathrm{T}}} \}$$

$$t^{a\prime}_{\mathrm{T}} = \begin{cases} t^{a}_{\mathrm{T}} + \textsc{time}(c, \mathrm{T}) & \textbf{if } \mathrm{T} \in \mathrm{Thrd}_{exe} \\ t^{a}_{\mathrm{T}} & \textbf{otherwise} \end{cases}$$

$$\mathbb{x}'\ x = \begin{cases} \mathbb{x}\ x & \textbf{if } \mathrm{Thrd}_{x} = \emptyset \\ \lambda \mathrm{T} \in \mathrm{Thrd}. (\mathrm{T} = \mathrm{T}' \ ?\ (\mathbb{x}'_{\mathrm{T}'}\ x)\ \mathrm{T}' : \emptyset) & \textbf{otherwise} \\ \qquad \textbf{where } \mathrm{T}' \text{ is one of the threads in } \mathrm{Thrd}_{x} = & \\ \qquad \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \exists r \in \mathrm{Reg}_{\mathrm{T}} : \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{store}\ r\ \texttt{to}\ x]^{pc_{\mathrm{T}}} \} & \end{cases}$$

$$\mathbb{l}''\ lck = \begin{cases} (\mathit{free}, \mathrm{T}', t, & \textbf{for some } \mathrm{T}' \in \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \\ \quad \textsc{pown}(\mathbb{l}\ lck), & \qquad \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{lock}\ lck]^{pc_{\mathrm{T}}} \}, \\ \qquad \textsc{rel}(\mathbb{l}\ lck)) & \textbf{if } \{ \mathrm{T} \in \mathrm{Thrd}_{exe} \mid \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = \\ & \quad [\texttt{lock}\ lck]^{pc_{\mathrm{T}}} \} \neq \emptyset \wedge \textsc{own}(\mathbb{l}\ lck) = \bot_{thrd} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}'\ lck = \begin{cases} \mathbb{l}'_{\textsc{own}(\mathbb{l}''\ lck)}\ lck & \textbf{if } \textsc{own}(\mathbb{l}''\ lck) \in \mathrm{Thrd}_{exe} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

Fig. 2. $c \xrightarrow[prg]{} c'$, the semantics of concrete transitions thread, and its release time to t) if the issuing thread is the owner of the lock, otherwise `unlock` is a no-operation; `lock` is used to acquire the given lock in a mutually exclusive manner. Note that $\mathbb{l}''$ is used to choose which thread in a set of competing threads is successful in acquiring the lock and that the unsuccessful threads wait in a spin-lock fashion until the lock is released – which means that configurations can deadlock.

It should be apparent that the threads included in a transition between two configurations (i.e. the threads included in Thrd_{exe}) are such that they execute their respective current statement at the earliest point in time at which any such event occurs (i.e. at t). When a thread issuing `lock` is assigned the given lock, it sets the lock's state to *taken*. As can be seen, the lock assignment deadline is always t; i.e. the time at which a `lock`-statement is issued on the free lock and a lock owner assignment occurs (which means that the deadline will always be met by the assigned lock owner since the owner is guaranteed to be one of the threads in Thrd_{exe} that issue a `lock`-statement on the given lock – which also means that the state of the lock is *taken* iff the owner of it is not $\bot_{thrd}$). The complete semantics of PPL is formally defined and more extensively discussed in [16].

5 Abstractly Interpreting PPL

In the following, time will be assumed to be abstractly interpreted as an interval (c.f. [17]). For simplicity, values are also abstractly interpreted using the interval domain. However, several other domains for values could be used instead.

The abstract semantic state of a program is described by an abstract configuration, $\tilde{c} = \langle [\text{T}, pc_{\text{T}}, \tilde{\mathbb{r}}_{\text{T}}, \tilde{t}^{a}_{\text{T}}]_{\text{T}\in\text{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle$. Like for the concrete configuration, pc_{T} is a program counter, pointing to the current statement in T. $\tilde{\mathbb{r}}_{\text{T}}$ is a mapping (i.e. a function) from T's registers to their abstract values (i.e. intervals). $\tilde{t}^{a}_{\text{T}}$ is the accumulated execution time of T (i.e. an interval). $\tilde{\mathbb{x}}$ is a mapping from variables and threads to a set of timestamped values (i.e. pairs of intervals), where each such value might represent the actual value stored to the variable at the interval in time represented by $\tilde{c}$. $\tilde{\mathbb{l}}$ is a mapping from locks to tuples containing information on (in the following order) whether the lock is acquired or not, which thread owns it, a deadline for when the lock must be acquired by the owning thread, the previous owner, and when it was last released. It should thus be apparent that an abstract configuration corresponds to a set of concrete configurations. Thus, these domains safely over-approximate the corresponding concrete domains (Lemma 1).

Lemma 1. *An abstract configuration safely approximates a set of concrete configurations.*

Proof (sketch). This proof is conducted by first showing that there are Galois Connections [17] between the concrete and abstract domains for register, variable and lock mappings. Note that since the interval domain is used to approximate values and times, there exist Galois Connections between the concrete and abstract domains for values and time [17]. Finally, it is shown that there is a Galois Connection between the configuration domains [16]. □

From Definition 1, it is easy to see that for the interval domain, the BCET and WCET must be as given by Definition 2. α_t and γ_t are the abstraction and concretization functions between the concrete time and abstract time (i.e. interval) domains [17].

Definition 2. *Given a set of abstract configurations, $\tilde{C}$, the concrete* BCET *and* WCET *for that set are defined as:*

$$\begin{cases} \text{BCET} ::= \min(\{\max(\{\min(\gamma_t(\tilde{t}^a_{\text{T}})) \mid \text{T} \in \text{Thrd}\}) \mid \\ \qquad\qquad\qquad\qquad \langle [\text{T}, pc_{\text{T}}, \tilde{\mathbb{r}}_{\text{T}}, \tilde{t}^a_{\text{T}}]_{\text{T}\in\text{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle \in \tilde{C}\}) \\ \text{WCET} ::= \max(\{\max(\{\max(\gamma_t(\tilde{t}^a_{\text{T}})) \mid \text{T} \in \text{Thrd}\}) \mid \\ \qquad\qquad\qquad\qquad \langle [\text{T}, pc_{\text{T}}, \tilde{\mathbb{r}}_{\text{T}}, \tilde{t}^a_{\text{T}}]_{\text{T}\in\text{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle \in \tilde{C}\}) \end{cases}$$

The semantics of transitions between abstract configurations is described by $\xrightarrow[prg]{\sim}$ as defined in Fig. 3. Note that: ABSTIME, although its definition is out of scope for this paper, is assumed to be a safe approximation of TIME (Assumption 1); DLLOCK gives a safe approximation of the concrete point in time when the given lock must be acquired by some thread [16]; ACCTIME, considering some thread, T, gives a safe approximation of $t^{a\prime}_{\text{T}}$ as defined in Fig. 2 [16]; $\widetilde{\text{OWN}}$, $\widetilde{\text{POWN}}$ and $\widetilde{\text{REL}}$ are the abstract counterparts of the masking functions OWN, POWN and REL, respectively. (The definitions of the above functions are omitted due to space limitations; see [16].) $i_1 \tilde{+}_t i_2$ is the sum of the two intervals i_1 and i_2 [16]. $\xrightarrow[ax]{\sim}$ is further discussed below.

Assumption 1. *It is assumed that* ABSTIME *is a "non-negative" function in the interval domain that safely approximates* TIME *for any thread in any configuration, given a specific value of the thread's program counter, at a specific point (interval) in time.*

Like in the concrete semantics, which threads that execute their respective current statement on a given abstract transition is determined based on when in time this would happen. However, since time is approximated using intervals, it might not be possible to determine the exact order in which certain events occur in the abstract case:

1. The sets of threads that will execute their current statements on a transition (i.e. Thrd_{exe}) might differ between the concrete and abstract cases even if the given concrete configuration is safely approximated by the abstract one. Because of this, different program points might be "visited" in the concrete and abstract cases, and thus, the concrete collecting semantics (i.e. all configurations that are reachable from a set of initial configurations [18, 3]) cannot be safely over-approximated using $\xrightarrow[prg]{\sim}$.

$$\frac{\mathrm{Thrd}_{exe} \neq \emptyset \wedge \forall \mathrm{T} \in \mathrm{Thrd}_{exe} : \langle \mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}'', \tilde{t}^{a\prime}_{\mathrm{T}} \rangle \xrightarrow[ax]{\sim} \langle pc'_{\mathrm{T}}, \tilde{\mathbb{r}}'_{\mathrm{T}}, \tilde{\mathbb{x}}'_{\mathrm{T}}, \tilde{\mathbb{l}}'_{\mathrm{T}} \rangle}{\begin{array}{l}\tilde{c} = \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^{a}_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle \xrightarrow[prg]{\sim} \\ \quad \tilde{c}' = \langle [\mathrm{T}, (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ pc'_{\mathrm{T}} : pc_{\mathrm{T}}), (\mathrm{T} \in \mathrm{Thrd}_{exe} \ ?\ \tilde{\mathbb{r}}'_{\mathrm{T}} : \tilde{\mathbb{r}}_{\mathrm{T}}), \tilde{t}^{a\prime}_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}', \tilde{\mathbb{l}}' \rangle\end{array}}$$

where

$$\tilde{t}^{r}_{\mathrm{T}} = \textsc{absTime}(\tilde{c}, \mathrm{T})$$

$$\tilde{t}_{all} = \alpha_t(\{\min(\{\min(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid B\}), \min(\{\max(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid B\})\})$$

$$\textbf{where } B \iff \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\texttt{halt}]^{pc_{\mathrm{T}}} \wedge \forall lck \in \mathrm{Lck} : (\textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{lock}\ lck]^{pc_{\mathrm{T}}} \Rightarrow \widetilde{\textsc{own}}(\tilde{\mathbb{l}}\ lck) \in \{\bot_{thrd}, \mathrm{T}\})$$

$$\mathrm{Thrd}^{all}_{exe} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \tilde{t}_{all} \tilde{\sqcap}_t (\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}}) \neq \tilde{\bot}_t \wedge \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) \neq [\texttt{halt}]^{pc_{\mathrm{T}}}\}$$

$$\tilde{\mathbb{l}}''\ lck = \begin{cases} (free, \mathrm{T}', & \textbf{for some } \mathrm{T}' \in \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \\ \quad \textsc{dlLock}(\tilde{c}, lck), & \quad \exists l \in \mathbb{Z} : \textsc{stm}(\mathrm{T}, l) = [\texttt{lock}\ lck]^{l}\}, \\ \quad \textsc{p}\widetilde{\textsc{own}}(\tilde{\mathbb{l}}\ lck), & \textbf{if } \exists \mathrm{T} \in \mathrm{Thrd}^{all}_{exe} : \widetilde{\textsc{own}}(\tilde{\mathbb{l}}\ lck) = \bot_{thrd} \wedge \\ \quad \widetilde{\textsc{rel}}(\tilde{\mathbb{l}}\ lck)) & \quad \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{lock}\ lck]^{pc_{\mathrm{T}}} \\ \tilde{\mathbb{l}}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathrm{Thrd}_{\textbf{hold}} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \mid \exists lck \in \mathrm{Lck} : (\textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{lock}\ lck]^{pc_{\mathrm{T}}} \wedge \widetilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck) \neq \mathrm{T}) \vee \textsc{stm}(\mathrm{T}, pc_{\mathrm{T}}) = [\texttt{halt}]^{pc_{\mathrm{T}}}\}$$

$$\tilde{t} = \alpha_t(\{\min(\{\min(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\textbf{hold}}\}), \min(\{\max(\gamma_t(\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\textbf{hold}}\})\})$$

$$\mathrm{Thrd}_{exe} = \{\mathrm{T} \in \mathrm{Thrd}_{\tilde{c}} \setminus \mathrm{Thrd}_{\textbf{hold}} \mid \tilde{t} \tilde{\sqcap}_t (\tilde{t}^{a}_{\mathrm{T}} \tilde{+}_t \tilde{t}^{r}_{\mathrm{T}}) \neq \tilde{\bot}_t\}$$

$$\tilde{\mathbb{l}}'\ lck = \begin{cases} \tilde{\mathbb{l}}'_{\widetilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck)}\ lck & \textbf{if } \widetilde{\textsc{own}}(\tilde{\mathbb{l}}''\ lck) \in \mathrm{Thrd}_{exe} \\ \tilde{\mathbb{l}}''\ lck & \textbf{otherwise} \end{cases}$$

$$\tilde{\mathbb{x}}' = \begin{cases} \textsc{trim}(\tilde{\mathbb{x}}'', \tilde{t}) & \textbf{if } \mathrm{Thrd}_{\tilde{c}} = \mathrm{Thrd} \\ \tilde{\mathbb{x}}'' & \textbf{otherwise} \end{cases}$$

$$\textbf{where } (\tilde{\mathbb{x}}''\ x)\ \mathrm{T} = \begin{cases} (\tilde{\mathbb{x}}'_{\mathrm{T}}\ x)\ \mathrm{T} & \textbf{if } \mathrm{T} \in \mathrm{Thrd}_{exe} \\ (\tilde{\mathbb{x}}\ x)\ \mathrm{T} & \textbf{otherwise} \end{cases}$$

$$\tilde{t}^{a\prime}_{\mathrm{T}} = \textsc{accTime}(\langle [\mathrm{T}', pc_{\mathrm{T}'}, \tilde{\mathbb{r}}_{\mathrm{T}'}, \tilde{t}^{a}_{\mathrm{T}'}]_{\mathrm{T}' \in \mathrm{Thrd}_{\tilde{c}}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}'' \rangle, \mathrm{Thrd}_{exe}, \mathrm{T})$$

Fig. 3. $\tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'$, semantics of abstract transitions

2. The execution of `load`-statements cannot be safely approximated using the semantic transition rules if the `load` is not the sole statement executed in the transition and the value of a global variable (i.e. a variable that might be read by at least one thread and that might be written by at least one other thread) is to be loaded. The reason for this is that other threads might execute `store`-statements, writing to the loaded variable, in succeeding transitions that could semantically occur before the `load`-statement in the concrete case.
3. A similar reasoning to that for `load`-statements holds for `lock`-statements; a non-acquired lock cannot simply be assigned to one of the threads in Thrd_{exe} that are trying to acquire it, because in the concrete case, some other thread might be the first to acquire the lock.
4. Since threads are spinning on locks that are owned by some other thread in the concrete case, but are frozen (see below) in the abstract case, the timing behavior of deadlocked transitions cannot be safely approximated.

$\xrightarrow[ax]{\sim}$ (whose formal definition is omitted due to space limitations; see [16]) describes the abstract semantics of a single statement within a thread when considered in isolation from other threads. There is no difference between the concrete and abstract behavior of the `halt`-, `skip`-, :=- and `unlock`-statements; however, the abstract semantics of evaluating arithmetic (and boolean) expressions is safely induced from the concrete semantics [16, 17]. The abstract semantics of the `if`-statement is equivalent to the concrete semantics, with the exception that the register mapping for the issuing thread is restricted to exclude cases for which the given boolean expression cannot possibly hold [3, 16]. `store` now adds a tuple consisting of the value of the given register (i.e. an interval) and the value of $\tilde{t}_{\mathrm{T}}^{a\prime}$ (i.e. an interval) to the issuing thread's (i.e. T's) set of timestamped values for the given variable. `load` now loads the given register of the issuing thread (i.e. T) with the least upper bound of all values that could be the actual value of the given variable at $\tilde{t}_{\mathrm{T}}^{a\prime}$. Note that TRIM removes timestamped values from the given variable mapping that will never affect a `load`-statement for the given variable in any thread at the given point (i.e. interval) in time or in the future [16]. `lock` still acquires the given lock only if the issuing thread is the owner of the lock (in $\tilde{\mathbb{I}}''$). A difference between the concrete and abstract semantics for `lock` is that whenever some thread issues a `lock`-statement on a free lock in the abstract case, any thread that might want to acquire the lock somewhere in the program could be assigned the ownership of the lock; note that this means that a lock can be owned by some thread without actually being acquired by that thread (i.e. the state of the lock is *free* even if the owner is not $\perp_{thrd}$). Another difference is that in the abstract case, the issuing thread will be frozen (not at all considered in transitions) if the given lock is owned by some other thread. The issuing thread remains frozen until the lock becomes free again.

If a `lock`-issuing thread has not already acquired the given lock, then it must be that $\tilde{t}_{\mathrm{T}}^{a\prime}$ has not passed the deadline for the lock owner assignment and that the release time of the lock is not in the future for `lock` to successfully acquire the lock. If $\tilde{t}_{\mathrm{T}}^{a\prime}$ has passed the deadline for the lock owner assignment, then the lock owner assignment, and thus the configuration, has no concrete counterpart since it must be that some other thread has already acquired the given lock [16]. If the lock's release time is in the future, then $\tilde{t}^{a\prime}_{\mathrm{T}}$ will be increased to safely approximate the concrete spin-waiting [16].

The abstract transitions described by $\xrightarrow[prg]{\sim}$ safely approximate the corresponding concrete transitions, for each thread individually, if they do not include the loading of a global variable in some thread and all threads wanting to acquire some lock are eventually able to do so (Lemma 2); i.e. if the hazards in the problems described by 2 and 4 above do not occur. The problems described by 1–4 above will be further discussed in the next section.

Lemma 2. *For each possible chain of transitions (as described by $\xrightarrow[prg]{}$) given some concrete configuration, there is an abstract chain of transitions (as described by $\xrightarrow[prg]{\sim}$) that safely approximates the concrete chain, for each thread considered individually, given that the initial abstract configuration safely approximates the initial concrete configuration, the thread is eventually able to acquire any lock it wants to, and either $|\mathrm{Thrd}_{exe}| \not> 1$ or there is no thread in Thrd_{exe} that loads the value of a global variable for each transition on the chain.*

Proof (sketch). First note that the timing behavior of each thread can be considered in isolation from any other thread (follows from Assumption 1) and that $\xrightarrow[ax]{\sim}$ safely approximates $\xrightarrow[ax]{}$ (which partly follows from Lemma 1) [16].

Since either $|\mathrm{Thrd}_{exe}| \not> 1$ or there is no thread in Thrd_{exe} that loads the value of a global variable for each transition on the chain, it must be that the loading of global variables' values are never under-approximated since there cannot be any `store`-statements in any thread that can be issued in future transitions and that could semantically affect the loaded value [16].

And, since any thread that might want to acquire a lock somewhere in the program can become the owner of that lock when some thread issues a `lock`-statement on the given lock, and since all threads that want to acquire a lock will eventually be able to do so, it must be that there exist safe approximations of all concrete scenarios including synchronization on locks [16].

Thus, since $\xrightarrow[ax]{\sim}$ safely approximates $\xrightarrow[ax]{}$ and the timing behavior of each thread can be considered in isolation from any other thread, it must be that the lemma holds. □

6 Analyzing PPL Programs Using Abstract Execution

The abstract execution function, ABSEXE, defined in Algorithm 1, is a worklist algorithm that encapsulates $\xrightarrow[prg]{\sim}$ and explicitly handles the problems discussed in the previous section. A configuration is said to be in the final state if all threads are issuing the `halt`-statement. A configuration is said to be deadlocked if it cannot possibly reach the final state according to the semantic transition rules. A configuration is said to be timed-out if the final state cannot possibly be reached before a given point in time (i.e. $\tilde{t}_{to}$) according to the semantic transition rules. A configuration is said to have valid concrete counterparts if it represents at least one concrete configuration that can semantically occur.

Two cases for which a configuration lacks concrete counterparts are when a deadlock involves a non-acquired lock and when the owner of a non-acquired lock misses to acquire it before the expiration of the owner assignment deadline. Such configurations are discontinued. Note that a configuration representing a lock owner assignment where the owner of some lock has not yet acquired the lock, and the owner's accumulated execution time has not passed the owner assignment deadline, reaches a configuration with valid concrete counterparts if the owner issues a `lock`-statement on (i.e. acquires) the lock before the expiration of the deadline. A formal definition of ABSEXE is found in [16].

Algorithm 1 Abstract Execution

```
 1: function ABSEXE($\tilde{C}, \tilde{t}_{to}$)
 2:   $\tilde{C}^w \leftarrow \tilde{C}$ ;   $\tilde{C}^f \leftarrow \emptyset$ ;   $\tilde{C}^d \leftarrow \emptyset$ ;   $\tilde{C}^t \leftarrow \emptyset$
 3:   while $\tilde{C}^w \neq \emptyset$ do
 4:     extract a configuration, $\tilde{c}$, from $\tilde{C}^w$
 5:     $\tilde{C}^w \leftarrow \tilde{C}^w \setminus \{\tilde{c}\}$
 6:     if $\tilde{c}$ is in the final state then
 7:       $\tilde{C}^f \leftarrow \tilde{C}^f \cup \{\tilde{c}\}$
 8:     else if $\tilde{c}$ is in a deadlocked state then
 9:       $\tilde{C}^d \leftarrow \tilde{C}^d \cup \{\tilde{c}\}$
10:     else if $\tilde{c}$ is timed-out given $\tilde{t}_{to}$ then
11:       $\tilde{C}^t \leftarrow \tilde{C}^t \cup \{\tilde{c}\}$
12:     else if $\tilde{c}$ has, or could reach a $\tilde{c}'$ with, valid concrete counterparts then
13:       if a transition from $\tilde{c}$ includes more than one thread and some thread
              would load the value of a global variable then
14:         for each thread, T, that loads global data in the transition from $\tilde{c}$
15:           let $\tilde{c}^{\mathrm{T}}$ be like $\tilde{c}$, but with T and all its local states removed
16:           let $\tilde{t}_{to}^{\mathrm{T}}$ be such that, after this time, the data can be safely loaded
17:           let $(\tilde{C}_{\mathrm{T}}^f, \tilde{C}_{\mathrm{T}}^d, \tilde{C}_{\mathrm{T}}^t)$ be ABSEXE($\{\tilde{c}^{\mathrm{T}}\}, \tilde{t}_{to}^{\mathrm{T}}$)
18:           let T's loaded value be the least upper bound of the values that would
              be loaded for all configurations in $\tilde{C}_{\mathrm{T}}^f \cup \tilde{C}_{\mathrm{T}}^d \cup \tilde{C}_{\mathrm{T}}^t \cup \{\tilde{c}\}$
19:         end for
20:         let $\tilde{c}'$ be like $\tilde{c}$, but with the loading of global data safely approximated
21:         $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{\tilde{c}'\}$
22:       else
23:         $\tilde{C}^w \leftarrow \tilde{C}^w \cup \{\tilde{c}' \mid \tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'\}$
24:       end if
25:     end if
26:   end while
27:   return $(\tilde{C}^f, \tilde{C}^d, \tilde{C}^t)$
28: end function
```

The overall strategy of the algorithm is depicted in Fig. 4 (α_{conf} and γ_{conf} are the abstraction and concretization functions for configurations, respectively [16]); i.e. given some safely approximated (by $\tilde{c}_0$) concrete configuration, c_0, there is an abstract transition sequence (which is safe for each thread individually) for each possible concrete transition sequence starting from c_0, and if the concrete sequence reaches a final state configuration (c_q), then so will the corresponding abstract sequence and the concrete final state configuration will be safely approximated (considering all threads) by the abstract final state configuration ($\tilde{c}_w$). Note that c_1, c_2, ..., c_{q-1} might not be safely approximated to their entirety by any of the abstract configurations $\tilde{c}_1$, $\tilde{c}_2$, ..., $\tilde{c}_{w-1}$ because of problem 1, defined on page 8. Although, it should be noted that for each thread individually, there are abstract configurations among these that safely approximate all the concrete states of that thread on the given concrete transition sequence.

$$\begin{tikzcd}
c_0 \arrow[r, "prg"'] \arrow[d, shift right, "\alpha_{conf}"'] & c_1 \arrow[r, "prg"'] & c_2 \arrow[r, "prg"'] & \cdots \arrow[r, "prg"'] & c_q \arrow[d, shift right, "\alpha_{conf}"'] \\
\tilde{c}_0 \arrow[u, shift right, "\gamma_{conf}"'] \arrow[r, "\sim", "prg"'] & \tilde{c}_1 \arrow[r, "\sim", "prg"'] & \tilde{c}_2 \arrow[r, "\sim", "prg"'] & \cdots \arrow[r, "\sim", "prg"'] & \tilde{c}_w \arrow[u, shift right, "\gamma_{conf}"']
\end{tikzcd}$$

Fig. 4. Relation between concrete and abstract transitions

For each thread that issues a `load`-statement on some global variable while not being the sole thread in Thrd_{exe}, absExe removes that thread from the configuration and calls itself recursively (with an adapted timeout value) to derive all the possible values that could be loaded by the thread. Note that this is possible since the state for variables is a mapping from variables and threads to a set of timestamped values. This strategy addresses problem 2. Problem 3 is partly addressed in the definition of $\xrightarrow[prg]{\sim}$ (c.f. Fig. 3), as discussed in the previous section. absExe fully addresses the problem by collecting all the possible transitions (i.e. resulting configurations) and adding them to the worklist. Problem 4 is addressed by identifying deadlocked configurations and aborting their transitions.

absExe($\tilde{C}, \tilde{t}_{to}$) hence safely approximates the timing behavior of all threads in any configuration in the input set, $\tilde{C}$, up until $\tilde{t}_{to}$ (Theorem 1). It should be noted that if a transition sequence is aborted before a final state configuration is reached (e.g. because a deadlocked or timed-out configuration is identified), then an infinite WCET must be assumed for that transition sequence.

Theorem 1. *For each final state configuration in the concrete collecting semantics, given some initial set of configurations, C,* absExe($\tilde{C}, \tilde{t}_{to}$) *derives either a safe approximation of that configuration or aborts the transition sequence at some point due to reaching a timed-out configuration with respect to $\tilde{t}_{to}$ whenever it terminates, given that $\forall c \in C : \exists \tilde{c} \in \tilde{C} : c \in \gamma_{conf}(\tilde{c})$. Likewise, for all configurations in the concrete collecting semantics that are deadlocked,* absExe *derives either a deadlocked or timed-out configuration, whenever it terminates.*

Proof (sketch). First note that

1. there are Galois Connections between all concrete and abstract domains, including the domains for configurations (Lemma 1),

2. all concrete transitions described by $\xrightarrow[prg]{}$ are safely approximated by $\xrightarrow[prg]{\sim}$, provided that whenever a thread issues a `load`-statement on a global variable, that thread is the sole thread in Thrd_{exe} (Lemma 2),
3. all possible transitions for a given configuration, as described by $\xrightarrow[prg]{\sim}$ are collected and added to the worklist (note that this safely approximates all concrete orders in which threads can be assigned the ownerships of locks – this solves the problem discussed in 3 in the previous section),
4. if a thread issues a `load`-statement on a global variable and that thread is not the sole thread in Thrd_{exe}, then it is easy to see that the **for each**-loop (i.e. the recursive use of absExe) derives a safe approximation of the value as seen by that thread when issuing the `load`-statement (follows from Assumption 1 and Lemma 2) – this solves the problem discussed in 2 in the previous section,
5. the recursive calling of absExe eventually stops since the set of threads in any PPL program is finite,
6. if any of the added configurations lacks (and cannot reach a configuration that has) valid concrete counterparts, it is trivially safe to discontinue it,
7. deadlocked transition sequences are aborted, but remembered – this solves the problem discussed in 4 in the previous section, and
8. timed-out transition sequences are aborted, but remembered.

It is thus easy to see that the combined use of $\xrightarrow[prg]{\sim}$ and the explicit handling of each thread loading the value of a global variable when that thread is not alone in Thrd_{exe} means that all concrete transition sequences are safely approximated for each thread individually – this solves the problem discussed in 1 in the previous section.

But then it must be that for each final state configuration in the concrete collecting semantics, absExe (whenever it terminates) derives either an over-approximating final state configuration, or a timed-out configuration. Likewise, it must be that for each deadlocked configuration in the concrete collecting semantics, absExe (whenever it terminates) derives either a deadlocked configuration or a timed-out configuration. □

All the details and the complete soundness proof of the presented algorithm are given in [16]. Note that Theorem 1 does not state that Algorithm 1 terminates for all possible inputs. This is because it might not terminate for some inputs – this problem is inherent in abstract execution.

Since abstract execution is not based on fixed point calculation of the collecting semantics in the traditional sense, widening and narrowing [17] cannot be used to alleviate this issue. Instead, timeouts on execution times and/or the number of transitions can be set in different ways to guarantee termination of the analysis for all cases. This is further discussed in Sect. 8.

It should also be noted that, although this paper focuses on timing analysis, the defined algorithm could also be used for deadlock analyses and termination analyses [16]. If the returned sets are such that $\tilde{C}^d = \emptyset$ and $\tilde{C}^t = \emptyset$, then the analyzed program is free of deadlocks and always terminates for all initial states given by the configurations in $\tilde{C}$. Safe timing bounds for the program are then easily extracted from the configurations in $\tilde{C}^f$. On the other hand, if $\tilde{C}^d \neq \emptyset$, then the program might deadlock for the given initial states, and if $\tilde{C}^t \neq \emptyset$, then it might be that the program does not terminate also due to other reasons, such as an infinite loop in some thread. However, deadlock and/or termination analysis is not the main focus of the presented approach and many other more specialized techniques targeting these areas exist.

7 Example Analysis

To clarify and explain Algorithm 1, this section instantiates it for an example PPL program containing a parallel loop. The example shows how communication through shared memory and synchronization on locks are handled.

The purpose of the program in Fig. 5 is to increment the value of the variable `x` with $\sum_{i=1}^{4}(2i + 3)$. The task of calculating the sum is equally divided onto two threads, T_1 and T_2. By definition, $\mathrm{Thrd} = \{\mathrm{T}_1, \mathrm{T}_2\}$, $\mathrm{Reg}_{\mathrm{T}_1} = \{\texttt{p}, \texttt{r}\}$, $\mathrm{Reg}_{\mathrm{T}_2} = \{\texttt{p}, \texttt{r}\}$, $\mathrm{Var} = \{\texttt{x}\}$ and $\mathrm{Lck} = \{\texttt{l}\}$. Note that `p` (and `r`) represents local memory within each thread; i.e. the register-name `p` (and `r`) can refer to two different memory locations – what location it refers to depends on which thread is considered. It is easy to see that `x` is a global variable when $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_1, \mathrm{T}_2\}$ and that there are no global variables when $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_1\}$ or $\mathrm{Thrd}_{\tilde{c}} = \{\mathrm{T}_2\}$.

$$\begin{aligned}
\mathrm{T}_1 = (1, &[\texttt{p} := \texttt{p+1}]^1\texttt{;}[\texttt{r} := \texttt{r+2*p+3}]^2\texttt{;}[\texttt{if p < 2 goto 1}]^3\texttt{;}[\texttt{lock l}]^4\texttt{;}\\
&[\texttt{load p from x}]^5\texttt{;}[\texttt{p} := \texttt{p+r}]^6\texttt{;}[\texttt{store p to x}]^7\texttt{;}[\texttt{unlock l}]^8\texttt{;}[\texttt{halt}]^9)\\
\mathrm{T}_2 = (2, &[\texttt{p} := \texttt{p+1}]^1\texttt{;}[\texttt{r} := \texttt{r+2*p+3}]^2\texttt{;}[\texttt{if p < 4 goto 1}]^3\texttt{;}[\texttt{lock l}]^4\texttt{;}\\
&[\texttt{load p from x}]^5\texttt{;}[\texttt{p} := \texttt{p+r}]^6\texttt{;}[\texttt{store p to x}]^7\texttt{;}[\texttt{unlock l}]^8\texttt{;}[\texttt{halt}]^9)
\end{aligned}$$

Fig. 5. Example: Program

For the sake of simplicity, the timing model (i.e. ABSTIME) as described in Table 1 gives that each statement within a thread has constant timing bounds; a '–' indicates that the entry is not applicable.

Table 1. Example: Timing model

pc_{T} ($\mathrm{T} \in \mathrm{Thrd}$) :	1	2	3	4	5	6	7	8	9
ABSTIME($\tilde{c}, \mathrm{T}_1$) :	$[2,2]$	$[1,1]$	$[1,2]$	$[1,2]$	$[2,3]$	$[1,1]$	$[2,3]$	$[2,3]$	–
ABSTIME($\tilde{c}, \mathrm{T}_2$) :	$[2,2]$	$[1,1]$	$[4,5]$	$[5,6]$	$[2,5]$	$[2,2]$	$[2,4]$	$[2,3]$	–

Assume that $\tilde{c}_0 = \langle [\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T} \in \mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$ is as described in Table 2. Note that `p` and `r` for T_1, and `r` for T_2, are initialized to $[0,0]$, and that `p` for T_2 is initialized to $[2,2]$. Table 2 also collects all the configurations derived by $\textsc{absExe}(\{\tilde{c}_0\},[-\infty,\infty])$. A '$-$' indicates that the entry is not included in the configuration. Due to space limitations, the details on how $\tilde{t}^a_{\mathrm{T}_1}$ and $\tilde{t}^a_{\mathrm{T}_2}$ are calculated on each transition cannot be fully presented; please refer to [16] in case of unclarities. If a thread, $\mathrm{T} \in \mathrm{Thrd}$, is not included in Thrd_{exe} (as defined in Fig. 3), then $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_i$ is equal to $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_{i-1}$, where $i > 0$. If T is included in Thrd_{exe}, then $\tilde{t}^a_{\mathrm{T}}$ in $\tilde{c}_i$ is equal to $\tilde{t}^a_{\mathrm{T}} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}_{i-1},\mathrm{T})$ in $\tilde{c}_{i-1}$, unless T has been frozen and must have its accumulated execution time adapted to approximate the concrete spin-waiting.

Figure 6 shows the relation between the derived configurations. In the figure, final configurations are circled, timed-out configurations are circled and marked 't', and discontinued (invalid) configurations are crossed out. $\tilde{c}^1_7$ is discontinued since the timing constraints given by $\tilde{t}^a_{\mathrm{T}_2} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}^1_7,\mathrm{T}_2) = [10,11] \mathbin{\tilde{+}_t} [4,5] = [14,16]$ and the lock owner assignment deadline, $[-\infty,12]$, give that T_2 cannot acquire `l` before T_1. $\tilde{c}^1_{12}$ is discontinued since T_1 cannot acquire `l` after reaching a `halt`-statement. Due to space limitations, the algorithm for calculating the deadline for the lock owner assignments made in the transitions from $\tilde{c}_6$ and $\tilde{c}_{11}$ cannot be presented; please refer to [16] in case of unclarities. Given $\tilde{c}^2_7$, a `store` to `x` in T_2 could affect the value loaded by T_1; however, the value loaded by T_1 cannot be affected after $\tilde{t}^a_{\mathrm{T}_1} \mathbin{\tilde{+}_t} \textsc{absTime}(\tilde{c}^2_7,\mathrm{T}_1) = [9,12] \mathbin{\tilde{+}_t} [2,3] = [11,15]$, which is hence the timeout value for the recursive instance of $\textsc{absExe}$.

It is apparent that $\textsc{absExe}(\{\tilde{c}_0\},[-\infty,\infty]) = (\{\tilde{c}_{16}\},\emptyset,\emptyset)$; i.e. $\tilde{c}_{16}$ is a final-state configuration and there are no deadlocked or timed-out configurations. It is thus easy to see that the program always terminates and that the estimated timing bounds are (Definition 2):

$$\begin{cases} \mathrm{BCET} = \min(\{\max(\{\min(\gamma_t(\tilde{t}^a_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}\}) \mid \\ \qquad\qquad \langle[\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \{\tilde{c}_{16}\}\}) = 27 \\ \mathrm{WCET} = \max(\{\max(\{\max(\gamma_t(\tilde{t}^a_{\mathrm{T}})) \mid \mathrm{T} \in \mathrm{Thrd}\}) \mid \\ \qquad\qquad \langle[\mathrm{T}, pc_{\mathrm{T}}, \tilde{\mathbb{r}}_{\mathrm{T}}, \tilde{t}^a_{\mathrm{T}}]_{\mathrm{T}\in\mathrm{Thrd}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}\rangle \in \{\tilde{c}_{16}\}\}) = 42 \end{cases}$$

$$\begin{tikzcd}
\textsc{absExe}(\{\tilde{c}_0\},[-\infty,\infty]) \arrow[d] & & & & & & & \textsc{absExe}(\{\tilde{c}^2_{71}\},[11,15]) \arrow[r] \arrow[dr, dashed] & \tilde{c}^2_{71} \arrow[r] & \underset{t}{\boxed{\tilde{c}^2_{72}}} \\
\tilde{c}_0 \arrow[r] & \tilde{c}_1 \arrow[r] & \tilde{c}_2 \arrow[r] & \tilde{c}_3 \arrow[r] & \tilde{c}_4 \arrow[r] & \tilde{c}_5 \arrow[r] & \tilde{c}_6 \arrow[r] \arrow[dr] & \tilde{c}^2_7 \arrow[u] \arrow[r, dashed] & \tilde{c}_8 \arrow[dd] & \\
 & & & & \cancel{\tilde{c}^1_{12}} & & & \cancel{\tilde{c}^1_7} & & \\
\boxed{\tilde{c}_{16}} & \tilde{c}_{15} \arrow[l] & \tilde{c}_{14} \arrow[l] & \tilde{c}_{13} \arrow[l] & \tilde{c}^2_{12} \arrow[l] & \tilde{c}_{11} \arrow[l] \arrow[ul] & \tilde{c}_{10} \arrow[l] & & \tilde{c}_9 \arrow[ll] &
\end{tikzcd}$$

Fig. 6. Example: Configuration relations

Table 2. Example: Derived configurations

$\tilde{c}$	pc_{T_1}	pc_{T_2}	$\tilde{\mathtt{r}}_{\mathrm{T}_1}\ \mathtt{p}$	$\tilde{\mathtt{r}}_{\mathrm{T}_1}\ \mathtt{r}$	$\tilde{\mathtt{r}}_{\mathrm{T}_2}\ \mathtt{p}$	$\tilde{\mathtt{r}}_{\mathrm{T}_2}\ \mathtt{r}$	$\tilde{t}^a_{\mathrm{T}_1}$	$\tilde{t}^a_{\mathrm{T}_2}$	$(\tilde{\mathtt{x}}\ \mathtt{x})\ \mathrm{T}_1$	$(\tilde{\mathtt{x}}\ \mathtt{x})\ \mathrm{T}_2$	$\tilde{\mathbb{l}}\ \mathtt{l}$
$\tilde{c}_0$	1	1	$[0,0]$	$[0,0]$	$[2,2]$	$[0,0]$	$[0,0]$	$[0,0]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_1$	2	2	$[1,1]$	$[0,0]$	$[3,3]$	$[0,0]$	$[2,2]$	$[2,2]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_2$	3	3	$[1,1]$	$[5,5]$	$[3,3]$	$[9,9]$	$[3,3]$	$[3,3]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_3$	1	3	$[1,1]$	$[5,5]$	$[3,3]$	$[9,9]$	$[4,5]$	$[3,3]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_4$	2	1	$[2,2]$	$[5,5]$	$[3,3]$	$[9,9]$	$[6,7]$	$[7,8]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_5$	3	1	$[2,2]$	$[12,12]$	$[3,3]$	$[9,9]$	$[7,8]$	$[7,8]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_6$	4	2	$[2,2]$	$[12,12]$	$[4,4]$	$[9,9]$	$[8,10]$	$[9,10]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},\tilde{\bot}_t,\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}^1_7$	4	3	$[2,2]$	$[12,12]$	$[4,4]$	$[20,20]$	$[8,10]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\mathrm{T}_2,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}^2_7$	5	3	$[2,2]$	$[12,12]$	$[4,4]$	$[20,20]$	$[9,12]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}^2_{71}$	–	3	–	–	$[4,4]$	$[20,20]$	–	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}^2_{72}$	–	4	–	–	$[4,4]$	$[20,20]$	–	$[14,16]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_8$	6	3	$[0,0]$	$[12,12]$	$[4,4]$	$[20,20]$	$[11,15]$	$[10,11]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_9$	7	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[12,16]$	$[14,16]$	$\{([0,0],[0,0])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_{10}$	8	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[14,19]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_1,[-\infty,12],\bot_{thrd},\tilde{\bot}_t)$
$\tilde{c}_{11}$	9	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\bot_{thrd},[-\infty,12],\mathrm{T}_1,[16,22])$
$\tilde{c}^1_{12}$	9	4	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[14,16]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{free},\mathrm{T}_1,[-\infty,28],\mathrm{T}_1,[16,22])$
$\tilde{c}^2_{12}$	9	5	$[12,12]$	$[12,12]$	$[4,4]$	$[20,20]$	$[16,22]$	$[19,28]$	$\{([0,0],[0,0]),$ $([12,12],[14,19])\}$	$\{([0,0],[0,0])\}$	$(\mathit{taken},\mathrm{T}_2,[-\infty,28],\mathrm{T}_1,[16,22])$
$\tilde{c}_{13}$	9	6	$[12,12]$	$[12,12]$	$[12,12]$	$[20,20]$	$[16,22]$	$[21,33]$	$\{([12,12],[14,19])\}$	$\{(\tilde{\bot}_{val},\tilde{\bot}_t)\}$	$(\mathit{taken},\mathrm{T}_2,[-\infty,28],\mathrm{T}_1,[16,22])$
$\tilde{c}_{14}$	9	7	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[23,35]$	$\{([12,12],[14,19])\}$	$\{(\tilde{\bot}_{val},\tilde{\bot}_t)\}$	$(\mathit{taken},\mathrm{T}_2,[-\infty,28],\mathrm{T}_1,[16,22])$
$\tilde{c}_{15}$	9	8	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[25,39]$	$\{([12,12],[14,19])\}$	$\{([32,32],[25,39])\}$	$(\mathit{taken},\mathrm{T}_2,[-\infty,28],\mathrm{T}_1,[16,22])$
$\tilde{c}_{16}$	9	9	$[12,12]$	$[12,12]$	$[32,32]$	$[20,20]$	$[16,22]$	$[27,42]$	$\{([12,12],[14,19])\}$	$\{([32,32],[25,39])\}$	$(\mathit{free},\bot_{thrd},[-\infty,28],\mathrm{T}_2,[27,42])$

8 Conclusions & Future Work

This paper has presented a parallel programming language, PPL, with shared memory and synchronization primitives acting on locks, and an algorithm that derives safe approximations of the BCET and WCET of PPL programs, given some sets of initial states and a timing model of the underlying architecture. The algorithm is based on abstract execution, which itself is based on abstract interpretation of the PPL semantics, which helps proving the soundness of the algorithm due to the existence of a Galois Connection between final concrete and abstract configurations.

The recursive definition of the algorithm means that several auxiliary states might have to be searched when some thread loads global data to make sure that the loaded value is a safe approximation of the corresponding concrete value(s). However, since this only happens for a limited amount of steps (until no thread can affect the loaded value anymore), it is expected that this will not have a huge impact on the complexity of the algorithm.

The over-approximate lock owner assignment could cause a lot of auxiliary configurations to be added to the worklist. However, this is necessary to cover all the concrete possibilities for in which orders the locks are taken by the threads. The discontinuation of cases that are guaranteed to never occur concretely both lowers the complexity and increases the precision of the algorithm, and also avoids it to deadlock (which otherwise could happen even though the analyzed program might be deadlock free [16]).

Future work includes implementing and evaluating the algorithm. This includes deriving a timing model for some more or less realistic architecture. The precision of the timing model is expected to have a great impact on the complexity of the analysis presented in this paper. Therefore, efforts will also be made to decrease the overall complexity of the algorithm. How large parallel programs that will be analyzable by the presented approach remains an open question until the implementation and evaluation have been performed. However, it is already obvious that well-written parallel programs (i.e. programs in which communication through shared memory and synchronization on locks is minimized while thread-local computations are maximized; c.f. the example presented in Sect. 7) will be less complex to analyze.

Future work also includes extending PPL with more statements and operations so that a real programming language can be modeled. One example is to include different addressing modes so that for example arrays can be introduced and operated on. Another example could be to introduce other synchronization primitives, e.g. barriers.

As previously mentioned, the risk of nontermination is inherent in abstract execution since the technique is basically a symbolic execution of the analyzed program. Detecting deadlocks partly solves this issue. Solving the issue completely can be done by setting a finite upper limit on the number of abstract transitions. If the limit is reached, the analysis could simply terminate and result in an infinite upper bound on the execution time. Other timeouts could also be set, e.g. as upper limits on the calculated execution times of the threads in the analyzed program or as an upper limit on the run (i.e. execution) time of the analysis itself. Note that terminating the analysis before all possible transition sequences have been fully evaluated (i.e. before a final configuration has been reached) must result in an infinite estimation of the upper limit on the execution time (i.e. on the WCET).

The path-explosion problem is still an open issue. In the sequential case of abstract execution, this is solved by merging states [19]. However, this technique is not expected to be very successful for the analysis presented in this paper since all the concrete parts of the system state (i.e. the threads' program counters, the lock states and owners, etc.) would have to be equal for the states to be merged. Defining a more approximate abstract lock state could resolve this issue. How to make this abstraction will be a challenge for not losing too much precision in the analysis.

Acknowledgment

The research presented in this paper was supported by the Swedish Foundation for Strategic Research (SSF) via the project RALF3[1].

References

1. OpenMP: OpenMP Application Program Interface, Version 3.0 (May 2008) `http://www.openmp.org/mp-documents/spec30.pdf`.
2. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem — overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7**(3) (2008) 1–53
3. Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden (May 2000)
4. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil, IEEE Computer Society (December 2006) 57–66
5. Ermedahl, A., Gustafsson, J., Lisper, B.: Deriving WCET bounds by abstract execution. In Healy, C., ed.: Proc. 11^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2011), Porto, Portugal (July 2011)
6. Shaw, A.C.: Reasoning about time in higher-order software. In: IEEE Transactions on Software Engineering. Volume 15. (1989) 737–750
7. Huber, B., Schoeberl, M.: Comparison of implicit path enumeration and model checking based WCET analysis. In: Proc. 9^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2009). (2009)

[1] `http://www.es.mdh.se/projects/295-RALF3___Software_for_Embedded_High_Performance_Architectures`

8. Metzner, A.: Why model checking can improve WCET analysis. In: Lecture Notes in Computer Science. Volume 3114/2004., Springer Berlin / Heidelberg (July 2004)
9. Gustavsson, A., Ermedahl, A., Lisper, B., Pettersson, P.: Towards WCET analysis of multicore architectures using UPPAAL. In Lisper, B., ed.: Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010), Brussels, Belgium, OCG (July 2010) 103–113
10. Lv, M., Guan, N., Yi, W., Deng, Q., Yu, G.: Efficient instruction cache analysis with model checking. In: Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session. (2010) 33–36
11. Wu, L., Zhang, W.: Bounding worst-case execution time for multicore processors through model checking. In: Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session. (April 2010) 17–20
12. Gustavsson, A., Gustafsson, J., Lisper, B.: Toward static timing analysis of parallel software. In Vardanega, T., ed.: Proc. 12th International Workshop on Worst-Case Execution Time Analysis (WCET'2012). Volume 23 of OpenAccess Series in Informatics (OASIcs). (July 2012) 38–47
13. Mittermayr, R., Blieberger, J.: Timing analysis of concurrent programs. In: Proc. 12th International Workshop on Worst-Case Execution Time Analysis (WCET'2012). (2012) 59–68
14. Potop-Butucaru, D., Puaut, I.: Integrated Worst-Case Execution Time Estimation of Multicore Applications. In: Proc. 13th International Workshop on Worst-Case Execution Time Analysis (WCET'2013), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
15. Ozaktas, H., Rochange, C., Sainrat, P.: Automatic WCET Analysis of Real-Time Parallel Applications. In: Proc. 13th International Workshop on Worst-Case Execution Time Analysis (WCET'2013), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
16. Gustavsson, A.: Static Timing Analysis of Parallel Software Using Abstract Execution. Licentiate thesis, Mälardalen University (2014) URL: `http://www.es.mdh.se/publications/3025-Static_Timing_Analysis_of_Parallel_Software_Using_Abstract_Execution`.
17. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edition. Springer (2005) ISBN 3-540-65410-0.
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symposium on Principles of Programming Languages, Los Angeles (January 1977) 238–252
19. Gustafsson, J., Ermedahl, A.: Merging techniques for faster derivation of WCET flow information using abstract execution. In Kirner, R., ed.: Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008), Prague, Czech Republic (July 2008)