Toward Static Timing Analysis of Parallel Software*

Andreas Gustavsson, Jan Gustafsson, and Björn Lisper

School of Innovation Design and Engineering, Mälardalen University, Sweden
{andreas.sg.gustavsson,jan.gustafsson,bjorn.lisper}@mdh.se

Abstract

The current trend within computer, and even real-time, systems is to incorporate parallel hardware, e.g., multicore processors, and parallel software. Thus, the ability to safely analyse such parallel systems, e.g., regarding the timing behaviour, becomes necessary. Static timing analysis is an approach to mathematically derive *safe* bounds on the execution time of a program, when executed on a given hardware platform. This paper presents an algorithm that statically analyses the timing of parallel software, with threads communicating through shared memory, using abstract interpretation. It also gives an extensive example to clarify how the algorithm works.

1998 ACM Subject Classification F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*Program Analysis*

Keywords and phrases Parallelism, BCET, WCET, Static analysis, Abstract interpretation

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

Many safety-critical embedded systems have hard real-time requirements. For these, safe bounds on the Best- and Worst-Case Execution Times (BCET/WCET) of the tasks in the system are key measures. Together, they define an interval in time within which the execution of the task is guaranteed to finish. In particular WCET bounds are needed by, e.g., schedulability analyses.

For reasons of energy consumption and performance, development in hardware today strives toward massively parallel architectures, like many-core, GPU and even special purpose, heterogeneous platforms. Thus, it is very likely that software tasks in future real-time systems will be parallel in order to utilise the provided computing power. Therefore, efforts must be made in providing WCET analyses for such systems.

This paper focuses on analysing the timing behaviour of *parallel software* with *dependent* sub-tasks, using a programming model with threads, shared memory, and locks. This kind of programming model is commonly used in parallel software today. It is assumed that an arbitrary underlying timing model, which can predict safe bounds on the BCET and WCET of individual instructions given a certain system state, is provided. An algorithm to statically derive the BCET and WCET of parallel software using abstract interpretation is presented. A technical report [8] covers the details and correctness proofs of the algorithm.

The rest of this paper is organised as follows. Section 2 presents related work on static timing analysis for parallel systems. Section 3 introduces a small model parallel language, with threads, thread-local and global memory, and locks. We also give a formal semantics for the language, including time, and we then present an analysis based on abstract interpretation. Section 4 clarifies how the analysis works by instantiating it for a given example program. Section 5 concludes the presentation with some discussion and directions for the future.

* This work was partly funded by the Swedish Research Council (VR) through project 2008-4650, "Worst-Case Execution Time Analysis of Parallel Systems".

2 Related Work

As far as we know, there have not been many attempts to statically analyse the execution time of explicitly parallel software. The parMERASA project provides a timing analysable multicore CPU with a system level software (c.f., operating system). In [11], a case study is performed in which the WCET of a parallel 3D multigrid solver, executing on the MERASA platform, is derived. In [7], model-checking is used to derive the WCET of a minimal parallel program. It is shown that, since model-checking is based on exhaustive exploration of concrete states, it is difficult to achieve scalability using only the presented approach. In [9], abstract interpretation is combined with model-checking to avoid the found scalability problems. This work does not focus on explicitly parallel (e.g., threaded) software, though.

In [3], an approach to directly calculate the BCET and WCET for sequential programs using abstract execution [6] is presented. Our work takes basically the same approach, but for explicitly parallel programs.

There is also some research on static low-level analysis of parallel systems. In [1] and [12], static methods for analysing multicores with a shared L2 instruction cache are presented. In [1], effects from timing anomaly influenced pipelines are also taken into account.

3 Timing Analysis

In this section, an algorithm for timing analysis of programs containing *dependent* parallel threads will be defined. It is assumed that the underlying architecture consists of both thread-private and global memory, referred to as registers, $r \in \mathbf{Reg}$, and variables, $x \in \mathbf{Var}$, respectively, and that arithmetical operations etc. can be performed using values of registers. It will also be assumed that shared resources that can be acquired in a mutually exclusive manner by the threads are provided, and that the operations provided by the instruction set (statements) may have variable execution times. (C.f., multicore CPU:s, where you have local and global memory, a shared memory bus and mutual exclusion operations.) No further assumptions on the underlying architecture, e.g., the number of CPU:s, the memory hierarchy or whether an operating system is used, are made. Timing effects from such features should not be considered in the software model but in the model of the underlying architecture.

3.1 Abstract Interpretation

In general, a timing analysis based on the concrete semantic of a program is infeasible due to the enormous number of states that must be explored. Abstract interpretation [2, 4, 10] is a method for *safely* approximating the concrete program semantics and can be used to obtain a set of possible abstract states for each point in a program. An abstract state describes, and sometimes over-approximates, the information given by a *set* of concrete semantic states. This means that an analysis based on abstractly interpreting the semantics of a program can become less complex and more efficient, but might suffer from imprecision, compared to an analysis based on the concrete semantics.

The concrete semantics of a programming language can be abstracted in many different ways. The choice of abstraction is done by defining an abstract domain. An abstract domain is essentially the set of all possible abstract states that fit the definition of the domain. It is often shown that the abstract domain is a safe over-approximation of the concrete domain by deriving a Galois connection (an abstraction function, α, and a concretisation function, γ) between the two domains [10]. An example of an abstract value domain is $\mathbf{Intv} = \{[z_1, z_2] \mid int_{min} \le z_1 \le z_2 \le int_{max} \wedge z_1, z_2, int_{min}, int_{max} \in \mathbb{Z}\}$, i.e., the set of all intervals that "fit in" $[int_{min}, int_{max}]$. This domain can be used to over-approximate

$$P ::= T \mid P \parallel T \qquad s ::= [\texttt{halt}]^l \mid [\texttt{skip}]^l \mid [r \texttt{ := } a]^l \mid [\texttt{if } b \texttt{ goto } l']^l \mid s_1 \,\texttt{;}\, s_2 \mid$$
$$[\texttt{load } r \texttt{ from } x]^l \mid [\texttt{store } r \texttt{ to } x]^l \mid [\texttt{lock } lck]^l \mid [\texttt{unlock } lck]^l$$
$$T ::= (N, s) \qquad a ::= n \mid r \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2 \mid a_1 \texttt{ / } a_2$$
$$b ::= \texttt{true} \mid \texttt{false} \mid \texttt{!}b \mid b_1 \texttt{ \&\& } b_2 \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ <= } a_2$$

Figure 1 The parallel programming language.

$\text{STM}(T, pc)$	$\langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc+1, \mathbb{r}[r \mapsto \mathcal{A}[\![a]\!]\mathbb{r}], \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc+1, \mathcal{R}(r, \mathbb{r}, x, \mathbb{x}), \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}[x \mapsto (\mathbb{x}\ x)[T \mapsto \{(\mathbb{r}\ r, t)\}]], \mathbb{l} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\neg\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\text{OWN}(\mathbb{l}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (locked, T)] \rangle$	$\text{OWN}(\mathbb{l}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

where $\mathcal{R}(r, \mathbb{r}, x, \mathbb{x}) = \mathbb{r}[r \mapsto v]$ and $\{(v, t')\} = \bigcup_{T' \in \mathbf{Thrd}}((\mathbb{x}\ x)\ T')$

Figure 2 Semantics of concrete axiom transitions: $\langle T, pc, \mathbb{r}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[ax]{} \langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$

the concrete domain $\{z \mid int_{min} \leq z \leq int_{max} \wedge z, int_{min}, int_{max} \in \mathbb{Z}\}$, i.e., the set of all integers between (and including) int_{min} and int_{max}. It is easy to show that there exists a Galois connection between the domains **Intv** and $\mathcal{P}(\mathbb{Z})$ (see e.g., [4, 8, 10]), and thus the approximation is safe, given the abstraction function $\alpha_{int}(Z) = [\min(Z), \max(Z)]$ and the concretisation function $\gamma_{int}([z_1, z_2]) = \{z \in \mathbb{Z} \mid z_1 \leq z \leq z_2\}$.

3.2 A Parallel Programming Language

The analysis will be based on the parallel programming language defined in Fig. 1, which is a set of operations using the discussed architectural features. $P \in \mathbf{Prg}$ denotes a program, which simply is a number of threads, denoted by $T \in \mathbf{Thrd}$. A thread is a pair of a statement, $s \in \mathbf{Stm}$, and a unique identifier, $N \in \mathbf{ThrdID}$. This makes every thread unique and distinguishable from other threads, even if several threads contain the same statement. To increase the readability of the semantics, it will be assumed that the axiom-statements (all statements except the sequentially composed statement, $s_1 \,\texttt{;}\, s_2$) of each thread are uniquely labelled with consecutive labels, $l \in \mathbf{Lbl}$, and stored in an array-like fashion in ascending order of their labels. $a \in \mathbf{Aexp}$ and $b \in \mathbf{Bexp}$ denote an arithmetic and a boolean expression, respectively, $n \in \mathbf{Val}$ is an integer value, and $lck \in \mathbf{Lck}$ denotes a lock. Locks can be acquired in a mutually exclusive manner using `lock` and released using `unlock`. Values can be transferred between variables and registers using `load` and `store`. Conditional branching is performed using `if`, a register is assigned a value using `:=`, a no-operation is performed using `skip`, and `halt` stops the execution of the issuing thread. The arithmetical, boolean and relational operators are self-explanatory and will not be discussed further.

The semantics of the language is formally defined in Fig. 2 (individual axiom statements) and 3 (system of threads). $\mathbb{x} \in \mathbf{Var} \to \mathbf{Thrd} \to \mathcal{P}(\mathbf{Val} \times \mathbf{Time})$, $\mathbb{l} \in \mathbf{Lck} \to (\mathbf{Lck_{stt}} \times \mathbf{Thrd} \cup \{\bot_{thrd}\})$, where $\mathbf{Lck_{stt}} = \{unlocked, locked\}$, and $t \in \mathbf{Time}$ are the states for variables and locks, and the current time. For each thread, T, in the program, there is also $pc_T \in \mathbf{Lbl}_T$, $\mathbb{r}_T \in \mathbf{Reg}_T \to \mathbf{Val}$, $t^r_T \in \mathbf{Time}$ and $t^a_T \in \mathbf{Time}$, which are the states of the program counter and registers of T, the relative execution time of T's active statement, $\text{STM}(T, pc_T)$, and the accumulated execution time for T, respectively. The tuple collecting

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \mathbb{r}_T, \mathbb{x}, \mathbb{l}'', {t^a_T}' \rangle \xrightarrow[ax]{} \langle pc'_T, \mathbb{r}'_T, \mathbb{x}'_T, \mathbb{l}'_T \rangle}{\begin{array}{l}\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[prg]{} \\ \qquad\qquad \langle \{(T, pc'_T, \mathbb{r}'_T, {t^r_T}', {t^a_T}') \mid T \in \mathbf{Thrd}\}, \mathbb{x}', \mathbb{l}', t' \rangle\end{array}}$$

where

$${t^r_T}' = \begin{cases} \textsc{finTime}(\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle, T) & \textbf{if } t = t^a_T \\ t^r_T & \textbf{otherwise} \end{cases}$$

$$t' = \min(\{t^a_T + {t^r_T}' \mid T \in \mathbf{Thrd}\})$$

$${t^a_T}' = \begin{cases} t^a_T + {t^r_T}' & \textbf{if } t' = t^a_T + {t^r_T}' \\ t^a_T & \textbf{otherwise} \end{cases}$$

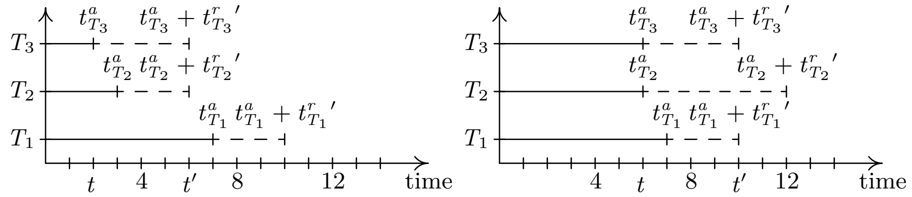
$$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd} \mid t' = {t^a_T}'\}$$

$$(\mathbb{x}' \ x) \ T = \begin{cases} \begin{cases} (\mathbb{x}'_T \ x) \ T & \text{for some } T \in \mathbf{Thrd}_{exe} : \exists r \in \mathbf{Reg}_T : \textsc{stm}(T, pc_T) = [\texttt{store } r \texttt{ to } x]^{pc_T} \\ \emptyset & \text{for } T' \in \mathbf{Thrd} \setminus \{T\}, \textbf{ if } \text{such a } T \text{ exists} \end{cases} \\ (\mathbb{x} \ x) \ T & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}'' \ lck = \begin{cases} (unlocked, T) & \text{for some } T \in \mathbf{Thrd}_{exe} : \textsc{stm}(T, pc_T) = [\texttt{lock } lck]^{pc_T}, \textbf{ if } \text{such} \\ & T \text{ exists, } \textsc{stt}(\mathbb{l} \ lck) = unlocked \text{ and } \textsc{own}(\mathbb{l} \ lck) = \bot_{thrd} \\ \mathbb{l} \ lck & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}' \ lck = \begin{cases} \mathbb{l}'_T \ lck & \text{for some } T \in \mathbf{Thrd}_{exe} : (\textsc{stm}(T, pc_T) = [\texttt{unlock } lck]^{pc_T} \vee \\ & (\textsc{own}(\mathbb{l}'' \ lck) = T \wedge \textsc{stm}(T, pc_T) = [\texttt{lock } lck]^{pc_T})), \textbf{ if } \text{such } T \text{ exists} \\ \mathbb{l} \ lck & \textbf{otherwise} \end{cases}$$

Figure 3 Semantics of concrete program transitions: $\langle \mathbf{Ts}, \mathbb{x}, \mathbb{l}, t \rangle \xrightarrow[prg]{} \langle \mathbf{Ts}', \mathbb{x}', \mathbb{l}', t' \rangle$



(a) c_1: $\mathbf{Thrd}_{exe} = \{T_2, T_3\}$ **(b)** c_2: $\mathbf{Thrd}_{exe} = \{T_1, T_3\}$

Figure 4 Illustration of how $\mathbf{Thrd}_{exe}$ is determined ($c_1 \xrightarrow[prg]{} c_2$).

all these states will be referred to as a configuration, c, i.e., $c = \langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t \rangle$. Note that states are updated on transitions, i.e., when pc is updated.

The state for locks keeps track of the state and owner of each lock. The owner is $\bot_{thrd}$ if no thread currently has the lock acquired. The state for registers of thread T simply keeps track of the current value of each register within T. The state for variables is not as intuitive. To be precise, the abstraction of the state for variables will need to save write history, i.e., what abstract writes (a pair of value and time) have been performed by each thread on each variable (see Section 3.3). Therefore, to derive a Galois connection (and hence implicitly get a safe approximation), the concrete state for variables has to be defined accordingly. In the concrete semantics, only one single write is saved for each variable, though. This write is non-deterministically chosen from one of the threads, if any, writing the variable at any given point in time (see Fig. 3). $\mathcal{R}$ is defined to return the value of the saved write (see Fig. 2).

$\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{Reg} \rightarrow \mathbf{Val}) \rightarrow \mathbf{Val}$ and $\mathcal{B} : \mathbf{Bexp} \rightarrow (\mathbf{Reg} \rightarrow \mathbf{Val}) \rightarrow \mathbf{Bool}$ evaluate arithmetic and boolean expressions, respectively, given a particular register state. The details of these functions are straightforward and can be found in [8]. $\textsc{finTime}$ is assumed to be provided by a timing-model of the underlying hardware. It should return a relative execution time for the statement of thread T, i.e., $\textsc{stm}(T, pc_T)$, based on the current system state. The set of threads to execute, $\mathbf{Thrd}_{exe}$, is determined based on t, $t^{r\prime}$ and t^a. It simply consists of the threads that will update their pc:s at the nearest point in time, t'. An illustration of how ${t^r_T}'$, t^a_T, t and t' are used to determine $\mathbf{Thrd}_{exe}$ is given in Fig. 4. For the arbitrary configuration c_1 in Fig. 4a, $t' = 6$ and hence $\mathbf{Thrd}_{exe} = \{T_2, T_3\}$. For c_2 (note that $c_1 \xrightarrow[prg]{} c_2$)

$\text{STM}(T, pc)$	$\langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \mathbb{l}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \tilde{\mathcal{A}}[\![a]\!]\tilde{\mathbb{r}}], \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \text{READ}(\tilde{\mathbb{x}}, x, T, \tilde{t})]\ \tilde{\mathbb{x}}, \mathbb{l} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \text{WRITE}(T, \tilde{\mathbb{x}}, x, (\tilde{\mathbb{r}}\ r, \tilde{t})), \mathbb{l} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc+1, \tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l} \rangle$	$\text{OWN}(\mathbb{l}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (locked, T)] \rangle$	$\text{OWN}(\mathbb{l}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

where $\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}} = \alpha_{reg}(\{\mathbb{r} \in \gamma_{reg}(\tilde{\mathbb{r}}) \mid \mathcal{B}[\![b]\!]\mathbb{r}\})$

Figure 5 Semantics of abstract axiom transitions: $\langle T, pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[ax]{\sim} \langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \mathbb{l}' \rangle$

in Fig. 4b, $t' = 10$ and hence $\mathbf{Thrd}_{exe} = \{T_1, T_3\}$.

The behaviour of locks needs to be explained. Assume that some threads in $\mathbf{Thrd}_{exe}$ execute a `lock`-statement on some lock, *lck*, and that *lck* is *unlocked* in the given configuration. In the resulting configuration, $\text{STT}(\mathbb{l}'\ lck) = locked$ and the owner will be one of the threads that tried to acquire *lck*. The chosen thread is given by $\text{OWN}(\mathbb{l}''\ lck)$; note that $\mathbb{l}''$ is only used to control the behaviour of the rules for `lock` in Fig. 2. This thread will have incremented its *pc* and thus moved on to executing its next statement. All other threads that tried to acquire *lck* will again try to acquire *lck* since their *pc*:s are not changed. Note that the latter would also be the case for *all* threads in $\mathbf{Thrd}_{exe}$ that try to acquire an already locked lock that is not owned by themselves. Also note that a thread who owns a lock is allowed to repeatedly acquire this lock any number of times.

3.3 Abstractly Interpreting the Language Semantics

First, it must be decided what parts of the system state to interpret in an abstract way. To allow for the hardware timing-model to be abstracted as well, **Time** will be approximated using the interval domain, i.e., $\tilde{\mathbf{Time}} = \mathbf{Intv}$. This approach is also taken by Chattopadhyay et al. [1] to approximate the execution time of pipeline stages in order to deal with timing anomalies in multicore platforms. **Val** will also be abstracted using intervals, i.e., $\tilde{\mathbf{Val}} = \mathbf{Intv}$, to allow for an efficient handling of data flow. Since **Thrd**, **Lbl**, **Var**, **Reg**, **Lck**, **Aexp** and **Bexp** are defined by the software, it does not make any sense to abstract them for the defined analysis (see Section 3.4). And, since $\mathbf{Lck_{stt}}$ is comparable to **Bool**, an abstraction of it would not be very beneficial. The states implicitly affected by the abstractions of **Time** and **Val** are $\mathbb{r}$, $\mathbb{x}$, t^r, t^a, t, and thus c. The abstraction of these will be referred to as $\tilde{\mathbb{r}}$, $\tilde{\mathbb{x}}$, $\tilde{t}^r$, $\tilde{t}^a$, $\tilde{t}$ and $\tilde{c}$, respectively. In [8], it is shown that Galois connections (with the corresponding abstraction and concretisation functions) can be established between the concrete and abstract domains for these states, and thus, that the approximations are safe. It is also shown that the abstract axiom transition rules (including the abstract version of $\mathcal{A}$, i.e., $\tilde{\mathcal{A}}$) in Fig. 5 are safe approximations of the concrete rules in Fig. 2, and that the boolean restriction function, $\tilde{\mathcal{BR}}$, is safe. Note that the concretisation of $\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}}$ will always contain (at least) the concrete stores, derived from $\tilde{\mathbb{r}}$, in which b evaluates to `true`.

$\tilde{\mathbb{x}} \in \mathbf{Var} \to \mathbf{Thrd} \to \mathcal{P}(\tilde{\mathbf{Val}} \times \tilde{\mathbf{Time}})$ can save any number (i.e., the history) of abstract writes, $\tilde{w} \in \tilde{\mathbf{Val}} \times \tilde{\mathbf{Time}}$, for each thread that occur on some variable. This is done to increase the precision in the analysis, since then, sequence (within each thread) and timing information (between threads) can be used to get a tight value when reading a variable. $\text{WRITE}(T, \tilde{\mathbb{x}}, x, \tilde{w})$ is thus defined to simply add the write, $\tilde{w}$, to the set of write-history for

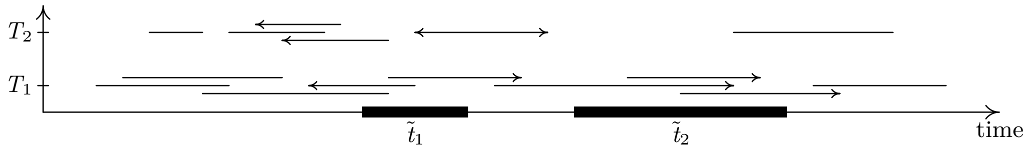


Figure 6 The time-stamps of the writes considered by $\textsc{read}(\tilde{\mathbb{x}}, x, T_1, \tilde{t}_1)$ and $\textsc{read}(\tilde{\mathbb{x}}, x, T_2, \tilde{t}_2)$.

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \tilde{\mathbb{r}}_T, \tilde{\mathbb{x}}, \mathbb{l}'', \tilde{t}^{a\prime}_T \rangle \xrightarrow[ax]{\sim} \langle pc'_T, \tilde{\mathbb{r}}'_T, \tilde{\mathbb{x}}'_T, \mathbb{l}'_T \rangle}{\begin{array}{l}\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[prg]{\sim} \\ \qquad\qquad \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^{r\prime}_T, \tilde{t}^{a\prime}_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}' \rangle\end{array}}$$

where

$$\tilde{t}^{r\prime}_T = \begin{cases} \textsc{AbsFinTime}\ (\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle, T) & \textbf{if } \tilde{t} \mathbin{\tilde{\sqcap}_t} \tilde{t}^a_T \neq \tilde{\bot}_t \\ \tilde{t}^r_T & \textbf{otherwise} \end{cases}$$

$$\tilde{t}' = \alpha_t(\{t_{min}, t_{max}\}) \qquad \textbf{where } t_{min} = \min\{\min(\gamma_t(\tilde{t}^a_T +_t \tilde{t}^{r\prime}_T)) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$
$$t_{max} = \min\{\max(\gamma_t(\tilde{t}^a_T +_t \tilde{t}^{r\prime}_T)) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$

$$\tilde{t}^{a\prime}_T = \begin{cases} \tilde{t}^a_T +_t \tilde{t}^{r\prime}_T & \textbf{if } \tilde{t}' \mathbin{\tilde{\sqcap}_t} (\tilde{t}^a_T +_t \tilde{t}^{r\prime}_T) \neq \tilde{\bot}_t \\ \tilde{t}^a_T & \textbf{otherwise} \end{cases}$$

$$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd}_{\tilde{c}} \mid \tilde{t}' \mathbin{\tilde{\sqcap}_t} \tilde{t}^{a\prime}_T \neq \tilde{\bot}_t\}$$
$$(\tilde{\mathbb{x}}'\ x)\ T = (\tilde{\mathbb{x}}'_T\ x)\ T$$
$$\mathbb{l}''\ lck = \ldots \ \text{(same as in Fig. 3)}$$
$$\mathbb{l}'\ lck = \ldots \ \text{(same as in Fig. 3)}$$

Figure 7 Semantics of abstract program transitions: $\langle \tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t} \rangle \xrightarrow[prg]{} \langle \tilde{\mathbf{T}}\mathbf{s}', \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}' \rangle$

thread T, i.e., to $((\tilde{\mathbb{x}}\ x)\ T)$. Using the sequence and timing information, $\textsc{read}(\tilde{\mathbb{x}}, x, T, \tilde{t})$ is defined to only take the writes that might be valid at $\tilde{t}$ (the point in time when T issues the $\textsc{read}$) into consideration for its returned value $\tilde{v} \in \mathbf{V\tilde{a}l}$. These writes, $\tilde{w} = (\tilde{v}', \tilde{t}')$, come from two categories. The first category covers the writes on x for threads $T' \neq T$ whose "time-stamps" overlap in time with $\tilde{t}$, i.e., $\tilde{t} \mathbin{\tilde{\sqcap}_t} \tilde{t}' \neq \tilde{\bot}_t$. The second category covers the most recent write on x for all threads (including T) such that its time-stamp overlaps with the overall most recent write of any write, not belonging to the first category. Note that any write for thread T with a time-stamp that begins after the beginning of $\tilde{t}$ is discarded. So is any write for $T' \neq T$ such that its time-stamp completely succeeds $\tilde{t}$. This is because such writes can simply not have occurred at the time of the $\textsc{read}$ (and will thus usually not be included in $\tilde{\mathbb{x}}$ at all). An illustration of the time-stamps of the writes on x, by some threads T_1 and T_2, stored in $\tilde{\mathbb{x}}$, that must be considered by $\textsc{read}(\tilde{\mathbb{x}}, x, T_1, \tilde{t}_1)$ (lines with arrow heads pointing left) and $\textsc{read}(\tilde{\mathbb{x}}, x, T_2, \tilde{t}_2)$ (lines with arrow heads pointing right) is given in Fig. 6. The returned value, $\tilde{v}$, is the least upper bound of the values of the considered writes.

The abstract transition rule for program configurations in Fig. 7 is an approximation of the concrete rule in Fig. 3. The abstract rule now defines a window in time, $\tilde{t}'$, that determines which threads are included in $\mathbf{Thrd}_{exe}$. The window reaches from the earliest point in time when some thread might update its pc, to the earliest point in time when some pc must be updated. $\textsc{AbsFinTime}$ is assumed to be a safe approximation of $\textsc{finTime}$.

The abstract rule in Fig. 7 is a safe approximation of the concrete rule in Fig. 3 only if some certain conditions are met. It is safe given that $|\mathbf{Thrd}_{\tilde{c}}| = 1$, or if a `load`-, `lock`- or `unlock`-statement is not executed by any thread in $\mathbf{Thrd}_{exe}$ [8]. This is easy to see since if these conditions are met, the threads in $\mathbf{Thrd}_{exe}$ execute independently from each other. If some thread in $\mathbf{Thrd}_{exe}$ would execute for example a `load`-statement, dependencies are introduced between the threads, and the $\textsc{read}$ function could return a value for which all possible writes have not been taken into account. Let's assume that $\mathbf{Thrd}_{exe} = \{T_1, T_2\}$, $\textsc{stm}(T_1, pc_{T_1}) = [\texttt{load}\ r\ \texttt{from}\ x]^{pc_{T_1}}$, $\textsc{stm}(T_2, pc_{T_2}) = [\texttt{skip}]^{pc_{T_2}}$

function $\textsc{abstractExecution}(\tilde{c}, \tilde{t}_{to})$
  $\textit{workset} \leftarrow \{\tilde{c}\}, \qquad \textit{finalset} \leftarrow \emptyset$
  repeat
    $\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \textbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \leftarrow \textsc{choose}(\textit{workset})$
    $\textit{workset} \leftarrow \textit{workset} \setminus \{\tilde{c}\}$
    if $\textsc{isTimeout}(\tilde{c}, \tilde{t}_{to}) \vee \textsc{isFinal}(\tilde{c})$ **then**
      $\textit{finalset} \leftarrow \textit{finalset} \cup \{\tilde{c}\}$
    else
      $\textbf{Thrd}_{\texttt{load}} \leftarrow \textsc{loadThrd}(\tilde{c})$
      if $\textbf{Thrd}_{\texttt{load}} \neq \emptyset \wedge |\textbf{Thrd}_{\tilde{c}}| > 1$ **then**
        for all $T' \in \textbf{Thrd}_{\texttt{load}}$ **do**
          $\tilde{t}^r_{T'}{}' \leftarrow \textsc{AbsFinTime}(\tilde{c}, T')$
          $x \leftarrow \textsc{getVarLoad}(\textsc{stm}(T', pc_{T'})), \qquad r \leftarrow \textsc{getRegLoad}(\textsc{stm}(T', pc_{T'}))$
          $\tilde{v} \leftarrow \tilde{\bot}_{val}, \qquad \tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \textbf{Thrd}_{\tilde{c}} \setminus \{T'\}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
          $\tilde{C}^f_{T'} \leftarrow \textsc{abstractExecution}(\tilde{c}', (\tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}') \tilde{\sqcap}_t \tilde{t}_{to})$
          for all $\langle\tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle \in \tilde{C}^f_{T'}$ **do**
            $\tilde{v} \leftarrow \tilde{v} \tilde{\sqcup}_{val} \textsc{read}(\tilde{\mathbb{x}}', x, T', \tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}')$
          end for
          $pc'_{T'} \leftarrow pc_{T'} + 1, \qquad \tilde{\mathbb{r}}'_{T'}\ r' \leftarrow \begin{cases} \tilde{v} & \textbf{if } r = r' \\ \tilde{\mathbb{r}}_{T'}\ r' & \textbf{otherwise} \end{cases}$
        end for
        $\tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \textbf{Thrd}_{\tilde{c}} \setminus \textbf{Thrd}_{\texttt{load}}\} \cup$
          $\{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T{}', \tilde{t}^a_T +_t \tilde{t}^r_T{}') \mid T \in \textbf{Thrd}_{\texttt{load}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
        $\textit{workset} \leftarrow \textit{workset} \cup \{\tilde{c}'\}$
      else
        $\tilde{C} \leftarrow \{\tilde{c}' \mid \tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'\}$
        $\tilde{C}' \leftarrow \{\langle\tilde{\mathbf{T}}\mathbf{s}, \textsc{trim}(\tilde{\mathbb{x}}, \tilde{t}), \mathbb{l}, \tilde{t}\rangle \mid \langle\tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \in \tilde{C}\}$
        $\textit{workset} \leftarrow \textit{workset} \cup \tilde{C}'$
      end if
    end if
  until $\textit{workset} = \emptyset$
  return *finalset*
end function

Figure 8 An algorithm for abstract execution.

and $\textsc{stm}(T_2, pc_{T_2} + 1) = [\texttt{store}\ r'\ \texttt{to}\ x]^{pc_{T_2}+1}$. When a transition occurs, the `load`- and `skip`-statements are considered. However, if the execution time of the `store`-statement (the abstract "point" in time when the thread's *pc* is updated) overlaps with the execution time of the `load`-statement, the resulting value of r in T_1 should be affected by the value of r' in T_2, but this will not be the case. A similar reasoning holds for `lock`- and `unlock`-statements.

3.4 Analysis by Abstract Execution

Since the abstract transition rule, $\xrightarrow[prg]{\sim}$, of Fig. 7 is not safe, one cannot simply use fixpoint-iterations [4, 10] on the abstract semantic rules to find a safe approximation to the concrete program semantics. Instead, a worklist algorithm will be defined that uses $\xrightarrow[prg]{\sim}$ in a safe way and handles the unsafe cases explicitly. The function $\textsc{abstractExecution}$ in Fig. 8 defines such an algorithm; the '@' symbol is used for denoting two ways of expressing the same thing (c.f., the "read as" operator in Haskell). Given a configuration, $\tilde{c}$, and a timeout, $\tilde{t}_{to}$, the function explores all the possible abstract transitions, until only final (all threads are standing on a `halt`-statement) and timed-out (all threads will update their *pc*:s at a point in time succeeding $\tilde{t}_{to}$) configurations remain. The function returns a set containing all the final and timed-out configurations. If a configuration is not final or timed-out, a transition will be performed. The threads executing `load`-statements are extracted and handled separately.

`thread T_1:`	`thread T_2:`	`thread T_3:`
`[load r from x]1;[1,5]`	`[load r from y]1;[1,6]`	`[if r<=3 goto 3]1;[1,3]`
`[store r to y]2;[1,3]`	`[store r to z]2;[2,3]`	`[store r to x]2;[2,3]`
`[halt]3`	`[halt]3`	`[halt]3`

Figure 9 Example program.

This is done by recursively using abstractExecution for each such thread to simulate how the rest of the threads in the configuration can affect the read value. When the effects have been derived, they are merged and put in the target register for the thread that issues the `load`-statement. Next, a new configuration, in which the `load`:s have been performed, is added to the worklist. Note that trim is used to remove parts of the history from $\tilde{\mathbb{x}}$ that cannot affect a `load`-statement in any thread at time $\tilde{t}$. This is to lower the space complexity of abstractExecution. Further details on the algorithm, definitions of the used functions and correctness proofs can be found in [8]. Note that this algorithm cannot safely analyse programs acting on locks. The algorithm will be extended with this ability (see Section 5).

Assuming that abstractExecution has been applied to some $\tilde{c}$ and that $\tilde{t}_{to} = [0, \infty]$, safe bounds on the corresponding concrete BCET and WCET can be extracted from the resulting set of configurations (details can be found in [8]).

4 Example

In this section, the program in Fig. 9 is analysed (the results of AbsFinTime are given after the non-`halt`-statements). Initially, let $\tilde{c} = \langle\{(T_1, 1, \tilde{\mathbb{r}}_{T_1}, [0,0], [0,0]), (T_2, 1, \tilde{\mathbb{r}}_{T_2}, [0,0], [0,0]), (T_3, 1, \tilde{\mathbb{r}}_{T_3}, [0,0], [0,0])\}, \tilde{\mathbb{x}}, \mathbb{l}, [0,0]\rangle$, where $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} = [2,4]$, $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_2) = ((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) = \emptyset$ and $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_1) = \{([1,1],[0,0])\}$, $((\tilde{\mathbb{x}}\ \mathtt{y})\ T_1) = ((\tilde{\mathbb{x}}\ \mathtt{y})\ T_2) = \emptyset$ and $((\tilde{\mathbb{x}}\ \mathtt{y})\ T_3) = \{([5,5],[0,0])\}$, and $((\tilde{\mathbb{x}}\ \mathtt{z})\ T_2) = \emptyset$, is analysed. abstractExecution$(\tilde{c}, [0, \infty])$ is summarised in Fig. 10.

The tuples in the chart represent program points, defined as $\langle pc_{T_1}, pc_{T_2}, pc_{T_3}\rangle$. As can be seen, for $\langle 1,1,1\rangle$, T_1 and T_2 both execute a `load`-statement. This means that two new instances of abstractExecution are created, one for each thread in $\mathbf{Thrd}_{\mathtt{load}}$. Within each of these instances, a new instance is created since one other thread also executes a `load`-statement. A '_' within the tuple indicates that the corresponding thread is removed from the configuration to evaluate the effects it might see. Next to each tuple and transition arrow, there is a comment stating what happens at the corresponding step. The found bounds on the BCET and WCET are 3 and 9, respectively. Note that AbsFinTime is assumed to be defined somewhere outside the scope of this paper. Also note that programs containing loops can be analysed, but due to space reasons, this is not illustrated here.

5 Discussion & Future Work

The algorithm in Fig. 8 is based on synchronously advancing the threads of a program between their respective program points. This, together with the defined abstract domain for variables, has the advantage that the analysis result will be the same as for the sequential case [6], when $P = T$. Another advantage is that the complexity of the algorithm becomes more dependent on the number of program points than on the timing behaviour of the program. To further reduce the time complexity of the algorithm, merging of configurations could be performed. Using the control flow graph (CFG) of the program, suitable merge-points within each thread can be found [5]. Typically, such points have multiple incoming edges.

A drawback for the algorithm in Fig. 8 is that termination is not guaranteed if a program consists of *infinite* loops. This could be resolved by adjusting the initial timeout, though.

Our current focus is to extend the algorithm to support programs using locks and then to

$\textsc{abstractExecution}(\tilde{c}, [0, \infty])$	
$\langle 1, 1, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \{T_1, T_2\}, \tilde{t}^r_{T_1} = [1, 5], \tilde{t}^r_{T_2} = [1, 6], \tilde{t}^r_{T_3} = [1, 3]$

$\textsc{abstractExecution}(\tilde{c}_1, [1, 5])$	
$\langle \_, 1, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \{T_2\}, \tilde{t}^r_{T_2} = [1, 6], \tilde{t}^r_{T_3} = [1, 3]$

$\textsc{abstractExecution}(\tilde{c}_{1.2}, [1, 5])$		
$\langle \_, \_, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset$	
$\swarrow \downarrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [2, 3]$	$(\tilde{\mathcal{BR}}[\![\mathtt{r <= 3}]\!]\tilde{\mathtt{r}}_{T_3} \neq \tilde{\bot}_{reg})$
$\downarrow \langle \_, \_, 3\rangle$	final $(\tilde{t}^a_{T_3} = [1, 3])$, no effects	
$\searrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [4, 4], \tilde{t}^a_{T_3} \leftarrow [1, 3]$	$(\tilde{\mathcal{BR}}[\![\mathtt{!(r <= 3)}]\!]\tilde{\mathtt{r}}_{T_3} \neq \tilde{\bot}_{reg})$
$\langle \_, \_, 2\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2, 3]$	
$\downarrow$	$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4, 4], [3, 6])\}$	
$\langle \_, \_, 3\rangle$	final $(\tilde{t}^a_{T_3} = [3, 6])$, $\mathtt{x}$ affected	

$\downarrow$	$\tilde{\mathtt{r}}_{T_2}\ \mathtt{r} \leftarrow [5, 5]$ (no effects on $\mathtt{y}$ from T_3), $\tilde{t}^a_{T_2} \leftarrow [1, 6]$
$\langle \_, 2, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_2} = [2, 3], \tilde{t}^r_{T_3} = [1, 3]$
$\swarrow \downarrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [2, 3], ((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) \leftarrow \{([5, 5], [3, 9])\}$,
$\downarrow \langle \_, 3, 3\rangle$	final $(\tilde{t}^a_{T_2} = [3, 9], \tilde{t}^a_{T_3} = [1, 3])$, $\mathtt{z}$ affected
$\searrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [4, 4], ((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) \leftarrow \{([5, 5], [3, 9])\}, \tilde{t}^a_{T_2} \leftarrow [3, 9], \tilde{t}^a_{T_3} \leftarrow [1, 3]$
$\langle \_, 3, 2\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2, 3]$
$\downarrow$	$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4, 4], [3, 6])\}$
$\langle \_, 3, 3\rangle$	final $(\tilde{t}^a_{T_2} = [3, 9], \tilde{t}^a_{T_3} = [3, 6])$, $\mathtt{x}$ and $\mathtt{z}$ affected

$\textsc{abstractExecution}(\tilde{c}_2, [1, 6])$	
$\langle 1, \_, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \{T_1\}, \tilde{t}^r_{T_1} = [1, 5], \tilde{t}^r_{T_3} = [1, 3]$

$\textsc{abstractExecution}(\tilde{c}_{2.1}, [1, 5])$	
$\langle \_, \_, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset$
$\swarrow \downarrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [2, 3]$
$\downarrow \langle \_, \_, 3\rangle$	final $(\tilde{t}^a_{T_3} = [1, 3])$, no effects
$\searrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [4, 4], \tilde{t}^a_{T_3} \leftarrow [1, 3]$
$\langle \_, \_, 2\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2, 3]$
$\downarrow$	$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) = \{([4, 4], [3, 6])\}$
$\langle \_, \_, 3\rangle$	final $(\tilde{t}^a_{T_3} = [3, 6])$, $\mathtt{x}$ affected

$\downarrow$	$\tilde{\mathtt{r}}_{T_1}\ \mathtt{r} \leftarrow [1, 4]$ (effects on $\mathtt{x}$ from T_3), $\tilde{t}^a_{T_1} \leftarrow [1, 5]$
$\langle 2, \_, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_1} = [1, 3], \tilde{t}^r_{T_3} = [1, 3]$
$\swarrow \downarrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [2, 3], ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1, 4], [2, 8])\}$,
$\downarrow \langle 3, \_, 3\rangle$	final $(\tilde{t}^a_{T_1} = [2, 8], \tilde{t}^a_{T_3} = [1, 3])$, $\mathtt{y}$ affected
$\searrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [4, 4], ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1, 4], [2, 8])\}, \tilde{t}^a_{T_1} \leftarrow [2, 8], \tilde{t}^a_{T_3} \leftarrow [1, 3]$
$\langle 3, \_, 2\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2, 3]$
$\downarrow$	$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4, 4], [3, 6])\}$
$\langle 3, \_, 3\rangle$	final $(\tilde{t}^a_{T_1} = [2, 8], \tilde{t}^a_{T_3} = [3, 6])$, $\mathtt{x}$ and $\mathtt{y}$ affected

$\downarrow$	(T_1 sees effects on $\mathtt{x}$ and $\mathtt{z}$, and T_2 sees effects on $\mathtt{x}$ and $\mathtt{y}$.)
$\downarrow$	$\tilde{\mathtt{r}}_{T_1}\ \mathtt{r} \leftarrow [1, 4], \tilde{\mathtt{r}}_{T_2}\ \mathtt{r} \leftarrow [1, 5], \tilde{t}^a_{T_1} \leftarrow [1, 5], \tilde{t}^a_{T_2} \leftarrow [1, 6]$
$\langle 2, 2, 1\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_1} = [1, 3], \tilde{t}^r_{T_2} = [2, 3], \tilde{t}^r_{T_3} = [1, 3]$
$\swarrow \downarrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [2, 3], ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1, 4], [2, 8])\}, ((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) \leftarrow \{([1, 5], [3, 9])\}$
$\downarrow \langle 3, 3, 3\rangle$	final $(\tilde{t}^a_{T_1} = [2, 8], \tilde{t}^a_{T_2} = [3, 9], \tilde{t}^a_{T_3} = [1, 3])$, $\mathtt{y}$ and $\mathtt{z}$ affected
$\searrow$	$\tilde{\mathtt{r}}_{T_3}\ \mathtt{r} \leftarrow [4, 4], ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) \leftarrow \{([1, 4], [2, 8])\}, ((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) \leftarrow \{([1, 5], [3, 9])\}$
$\langle 3, 3, 2\rangle$	$\mathbf{Thrd}_{\mathtt{load}} = \emptyset, \tilde{t}^r_{T_3} = [2, 3]$
$\downarrow$	$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_3) \leftarrow \{([4, 4], [3, 6])\}$
$\langle 3, 3, 3\rangle$	final $(\tilde{t}^a_{T_1} = [2, 8], \tilde{t}^a_{T_2} = [3, 9], \tilde{t}^a_{T_3} = [3, 6])$, $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$ affected

Figure 10 The steps taken by ABSTRACTEXECUTION when analysing the program in Fig. 9.

implement and evaluate it. Allowing the use of locks introduces a risk for deadlocks (both in the analysed program and thus the algorithm). However, deadlocks could easily be detected and handled by the algorithm, because all threads, not standing on a `halt`-statement, would be waiting to acquire a lock that is locked and not owned by themselves. Thus, this detection allows termination of the analysis (with a resulting WCET of ∞) even if deadlocks occur.

References

1 Sudipta Chattopadhyay, C.-L. Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In *18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'12)*, Beijing, China, April 2012.

2 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

3 Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In Chris Healy, editor, *Proc. 11^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2011)*, Porto, Portugal, July 2011.

4 Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

5 Jan Gustafsson and Andreas Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In Raimund Kirner, editor, *Proc. 8^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2008)*, Prague, Czech Republic, July 2008.

6 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. 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)*, pages 57–66, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

7 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *Proc. 10^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

8 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel systems – technical report. Technical Report 2796, Dept. of Computer Science and Engineering, Mälardalen University, April 2012.
URL: `http://www.mrtc.mdh.se/index.php?choice=publications&id=2796`.

9 Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In Scott Brandt, editor, *Proc. 31^{th} IEEE Real-Time Systems Symposium (RTSS'10)*, pages 339–349, San Diego, CA, December 2010. IEEE.

10 Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2^{nd} edition.* Springer, 2005. ISBN 3-540-65410-0.

11 Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In Björn Lisper, editor, *Proc. 10^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 90–100, Brussels, Belgium, July 2010. OCG.

12 Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proc. 14^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 80–89, June 2008.