Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs

Antoine Miné

Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs*

Antoine Miné

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract. We present a static analysis by Abstract Interpretation to check for run-time errors in *parallel* C programs. Following our work on Astrée, we focus on embedded critical programs without recursion nor dynamic memory allocation, but extend the analysis to a static set of threads. Our method iterates a slightly modified non-parallel analysis over each thread in turn, until thread interferences stabilize. We prove the soundness of the method with respect to a sequential consistent semantics and a reasonable weakly consistent memory semantics. We then show how to take into account mutual exclusion and thread priorities through partitioning over the scheduler state. We present preliminary experimental results analyzing a real program with our prototype, Thésée, and demonstrate the scalability of our approach.

Keywords: Parallel programs, static analysis, Abstract Interpretation, run-time errors.

1 Introduction

Ensuring the safety of critical embedded software is important as a single "bug" can have catastrophic consequences. Previous work on the Astrée analyzer [5] demonstrated that static analysis by Abstract Interpretation could help, when specializing an analyzer to a class of programs (synchronous control/command avionics C software) and properties (run-time errors). In this paper, we describe ongoing work to achieve similar results for *parallel* embedded programs. We wish to match the current trend in critical embedded systems to switch from large numbers of single-program processors communicating through a common bus to single-processor multi-threaded applications communicating through a shared memory (for instance, in the context of Integrated Modular Avionics). We focus on detecting the same kinds of run-time errors as Astrée does (arithmetic and memory errors) and take data-races into account (as they may cause such errors), but we ignore other concurrency errors (such as dead-locks, live-locks, or priority inversions), which are orthogonal.

* This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.

Our method is based on Abstract Interpretation [7], a general theory of the approximation of semantics, which allows designing static analyzers that are sound by construction, i.e., consider a superset of all program behaviors and thus cannot miss any bug, but can cause spurious alarms due to over-approximations. At its core, our method performs an analysis of each thread considering an abstraction of the effects of the other threads (called interferences). Each analysis generates a new set of interferences, and threads are re-analyzed until a fixpoint is reached. Thus, few modifications are required for a non-parallel analyzer to analyze parallel programs. Moreover, we show that few thread re-analyses are required in practice, resulting in a scalable analysis.

As we target embedded software, we can safely assume that there is no recursion nor dynamic allocation of memory, threads, or locks, which makes the analysis easier. In return, we handle two subtle points. Firstly, we consider a weakly consistent memory model: memory accesses not protected by mutual exclusion may cause behaviors that are not the result of any thread interleaving to appear, as they expose to concurrent threads compiler optimizations that are transparent on non-parallel programs. We handle this by proving that our semantics is invariant by some classes of program transformations. Secondly, we take into account the effect of a real-time scheduler that schedules the threads on a single processor according to strict, fixed priorities: only the unblocked thread of highest priority may run. This ensures some mutual exclusion properties that our target program exploits, and so should our analysis. This is achieved by partitioning with respect to an abstraction of the global scheduling state.

Our paper is organized as follows: Sec. 2 presents a classic non-parallel semantics, Sec. 3 considers threads in a shared memory, and Sec. 4 adds support for locks and priorities; Sec. 5 presents our prototype and experimental results, Sec. 6 discusses related work, and Sec. 7 concludes and envisions future work. We alternate between two kinds of semantics: semantics based on control paths, that can model precisely thread interleavings, and semantics by structural induction on the syntax, that give rise to effective abstract interpreters. Figure 1 summarizes the semantics introduced in the paper, using $\subseteq$ to denote the "is less abstract than" relation. Our analysis has already been mentioned, briefly and informally, in [4, § VI]. We offer here a formal, rigorous treatment.

2 Non-parallel Programs

This section recalls a classic static analysis by Abstract Interpretation of the run-time errors of *non-parallel* programs, as performed for instance by Astrée [5]. The formalization introduced here will be extended later to parallel programs, and it will be apparent that an analyzer for parallel programs can be constructed by extending an analyzer for non-parallel programs with few changes.

2.1 Syntax

For the sake of exposition, we reason on a vastly simplified programming language. However, the results extend naturally to a realistic language, such as the

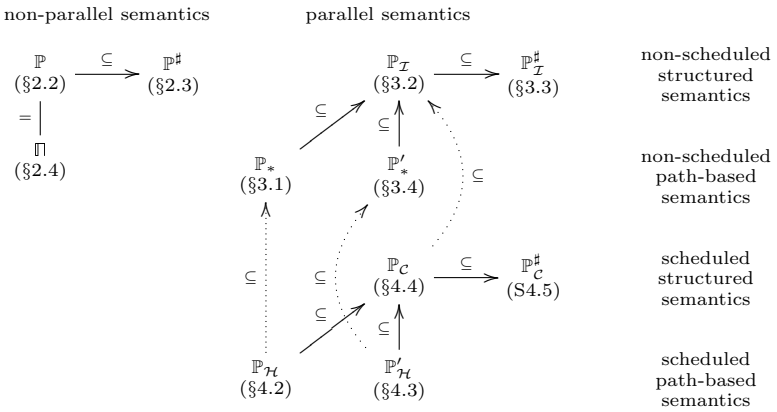


Fig. 1. Semantics defined in this paper.

subset of C excluding recursion and dynamic memory allocation. We assume a fixed, finite set $\mathcal{V}$ of variables and $\mathcal{F}$ of function names. A program has an entry-point $\mathit{entry} \in \mathcal{F}$ and associates to each function name $f \in \mathcal{F}$ a structured statement $\mathit{body}(f) \in \mathit{stat}$ in the following grammar:

$$\begin{array}{lll} \mathit{stat} ::= X \leftarrow \mathit{expr} & & \textit{(assignment, } X \in \mathcal{V}\textit{)} \\ \quad | \quad \textbf{if } \mathit{expr} \textbf{ then } \mathit{stat} & & \textit{(conditional)} \\ \quad | \quad \textbf{while } \mathit{expr} \textbf{ do } \mathit{stat} & & \textit{(loop)} \\ \quad | \quad \mathit{stat}; \mathit{stat} & & \textit{(sequence)} \\ \quad | \quad f() & & \textit{(function call, } f \in \mathcal{F}\textit{)} \end{array} \tag{1}$$

$$\mathit{expr} ::= X \mid [c_1, c_2] \mid \mathit{expr} \diamond_\ell \mathit{expr}$$
$$\text{where } X \in \mathcal{V},\ c_1, c_2 \in \mathbb{R} \cup \{\pm\infty\},\ \diamond \in \{+, -, \times, /\},\ \ell \in \mathcal{L}$$

For the sake of simplicity, we do not handle local variables (all variables are visible at all program points) nor function arguments and returns. Due to the absence of recursion, these could be easily simulated by using a finite set of global variables. Our toy language is limited to a single data-type (real numbers in $\mathbb{R}$) and numeric expressions. Constants are actually constant intervals $[c_1, c_2]$, which return a fresh value between c_1 and c_2 when evaluated. This allows modeling non-deterministic expressions and inputs from the environment. Each operator $\diamond_\ell$ is tagged with a syntactic location ℓ and we denote by $\mathcal{L}$ the finite set of all syntactic locations. The output of an analyzer will be the set of locations ℓ with errors (or rather, a superset of them, due to approximations).

2.2 Concrete Structured Semantics $\mathbb{P}$

We present a concrete semantics, that is, the most precise mathematical expression of program semantics we consider. As it is undecidable, it will be abstracted in the next section to obtain a sound static analysis.

A program environment $\rho \in \mathcal{E}$ maps each variable to a value, i.e., $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$. The semantics $\mathbb{E}[\![\, e \,]\!]$ of an expression $e \in \mathit{expr}$ maps an environment to a set

of values in $\mathcal{P}(\mathbb{R})$ (sets accounting for non-determinism) and a set of run-time error locations in $\mathcal{P}(\mathcal{L})$ (in our simple case, only for divisions by zero). It is defined by structural induction as follows:

$$\begin{array}{l}
\forall e \in \mathit{expr},\ \mathbb{E}[\![\, e \,]\!] : \mathcal{E} \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathcal{L})) \\
\mathbb{E}[\![\, X \,]\!]\rho \stackrel{\text{def}}{=} (\{\, \rho(X) \,\},\ \emptyset) \\
\mathbb{E}[\![\, [c_1, c_2] \,]\!]\rho \stackrel{\text{def}}{=} (\{\, c \in \mathbb{R} \mid c_1 \leq c \leq c_2 \,\},\ \emptyset) \\
\mathbb{E}[\![\, e_1 \diamond_\ell e_2 \,]\!]\rho \stackrel{\text{def}}{=} \\
\quad \text{let } (V_1, \Omega_1) = \mathbb{E}[\![\, e_1 \,]\!]\rho \text{ in} \\
\quad \text{let } (V_2, \Omega_2) = \mathbb{E}[\![\, e_2 \,]\!]\rho \text{ in} \\
\quad (\{\, x_1 \diamond x_2 \mid x_1 \in V_1,\ x_2 \in V_2,\ \diamond \neq / \vee x_2 \neq 0 \,\}, \\
\quad\ \Omega_1 \cup \Omega_2 \cup \{\, \ell \mid \diamond = / \wedge 0 \in V_2 \,\}) \\
\text{where } \diamond \in \{\, +, -, \times, / \,\}
\end{array} \tag{2}$$

We now consider the complete lattice $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{L})$ with partial order $\sqsubseteq$ defined as the pairwise set inclusion $(A, B) \sqsubseteq (A', B') \iff^{\text{def}} A \subseteq A' \wedge B \subseteq B'$. We denote by $\sqcup$ the associated join, i.e., pairwise set union. The *structured* semantics $\mathbb{S}[\![\, s \,]\!]$ of a statement s is a morphism in $\mathcal{D}$ that, given a set of environments R and errors Ω before a statement s, returns the reachable environments after s as well as Ω enriched with the errors encountered during the execution of s:

$$\begin{array}{l}
\forall s \in \mathit{stat},\ \mathbb{S}[\![\, s \,]\!] : \mathcal{D} \to \mathcal{D} \\
\mathbb{S}[\![\, X \leftarrow e \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad (\emptyset, \Omega) \sqcup \bigsqcup_{\rho \in R} \text{let } (V, \Omega') = \mathbb{E}[\![\, e \,]\!]\rho \text{ in } (\{\, \rho[X \mapsto v] \mid v \in V \,\},\ \Omega') \\
\mathbb{S}[\![\, s_1;\ s_2 \,]\!](R, \Omega) \stackrel{\text{def}}{=} (\mathbb{S}[\![\, s_2 \,]\!] \circ \mathbb{S}[\![\, s_1 \,]\!])(R, \Omega) \\
\mathbb{S}[\![\, \textbf{if } e \textbf{ then } s \,]\!](R, \Omega) \stackrel{\text{def}}{=} (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \neq 0? \,]\!])(R, \Omega) \sqcup \mathbb{S}[\![\, e = 0? \,]\!](R, \Omega) \\
\mathbb{S}[\![\, \textbf{while } e \textbf{ do } s \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad \mathbb{S}[\![\, e = 0? \,]\!](\text{lfp } \lambda X.\, (R, \Omega) \sqcup (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \neq 0? \,]\!])X) \\
\mathbb{S}[\![\, f() \,]\!](R, \Omega) \stackrel{\text{def}}{=} \mathbb{S}[\![\, \mathit{body}(f) \,]\!](R, \Omega) \\
\mathbb{S}[\![\, e \bowtie 0? \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad (\emptyset, \Omega) \sqcup \bigsqcup_{\rho \in R} \text{let } (V, \Omega') = \mathbb{E}[\![\, e \,]\!]\rho \text{ in } (\{\, \rho \mid \exists v \in V,\ v \bowtie 0 \,\},\ \Omega') \\
\text{with } \bowtie \in \{\, =, \neq \,\}
\end{array} \tag{3}$$

where $\rho[X \mapsto x]$ is the environment that maps X to x, and elements $Y \neq X$ to $\rho(Y)$. Loops and conditionals use the synthetic "guard" statements $e = 0?$ and $e \neq 0?$ that filter their argument and keep only those environments that may evaluate, respectively, to null (i.e., false) or non-null (i.e., true) values. Guards and assignments are collectively called *atomic statements*. The semantics of loops uses a least fixpoint operator lfp to compute a loop invariant. We have the following property:

Theorem 1. *$\forall s$, $\mathbb{S}[\![\, s \,]\!]$ is well defined and a strict, complete $\sqcup-$morphism.*

We can now define the concrete structured semantics of the program as follows:

$$\mathbb{P} \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{S}[\![\, \mathit{entry}() \,]\!](\mathcal{E}_0, \emptyset) \tag{4}$$

where $\mathcal{E}_0 \subseteq \mathcal{E}$ is a set of initial environments (e.g., $\mathcal{E}_0 \stackrel{\text{def}}{=} \{\, \rho_0 \,\}$ where $\forall X \in \mathcal{V},\ \rho_0(X) = 0$). Thus, we observe the set of run-time errors that can appear in

executions starting at the beginning of *entry* in an initial environment. Note that, as $\forall s$, $\mathbb{S}[\![s]\!](\emptyset, \Omega) = (\emptyset, \Omega)$, we also observe errors occurring in executions that loop forever or halt before the end of *entry*.

2.3 Abstract Structured Semantics $\mathbb{P}^\sharp$

The semantics $\mathbb{P}$ is not computable as it involves least fixpoints in an infinite-height domain $\mathcal{D}$. An effective analysis will instead compute an abstract semantics over-approximating the concrete one. This semantics is parametrized by an abstract domain of environments, i.e., a set $\mathcal{E}^\sharp$ of computer-representable abstract elements, with a partial order $\subseteq^\sharp$. Each abstract element represents a set of concrete environments through a monotonic concretization function $\gamma_\mathcal{E} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E})$. In particular, there is an element $\mathcal{E}_0^\sharp \in \mathcal{E}^\sharp$ representing initial environments: $\gamma_\mathcal{E}(\mathcal{E}_0^\sharp) \supseteq \mathcal{E}_0$. We also require a sound and effective abstract version of every concrete operator:

$$\begin{array}{l} \cup_\mathcal{E}^\sharp : (\mathcal{E}^\sharp \times \mathcal{E}^\sharp) \to \mathcal{E}^\sharp \\ \text{with } \forall A^\sharp, B^\sharp \in \mathcal{E}^\sharp,\ \gamma_\mathcal{E}(A^\sharp \cup_\mathcal{E}^\sharp B^\sharp) \supseteq \gamma_\mathcal{E}(A^\sharp) \cup \gamma_\mathcal{E}(B^\sharp) \end{array}$$

and, for atomic statements s, i.e., $s \in \{\, X \leftarrow e,\ e = 0?,\ e \neq 0? \,\}$:

$$\begin{array}{l} \mathbb{S}[\![s]\!]^\sharp : (\mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L})) \to (\mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L})) \\ \text{with } \forall (R^\sharp, \Omega),\ (\mathbb{S}[\![s]\!] \circ \gamma)(R^\sharp, \Omega) \sqsubseteq (\gamma \circ \mathbb{S}[\![s]\!]^\sharp)(R^\sharp, \Omega) \\ \text{where } \gamma(R^\sharp, \Omega) \stackrel{\text{def}}{=} (\gamma_\mathcal{E}(R^\sharp), \Omega) \end{array}$$

i.e., the soundness condition requires an abstract operator to output supersets of the environments and error locations returned by the concrete one. Finally, when $\mathcal{E}^\sharp$ has infinite strictly increasing chains, we require a widening operator $\triangledown_\mathcal{E}$ to ensure the convergence of abstract fixpoint computations in finite time:

$$\begin{array}{l} \triangledown_\mathcal{E} : (\mathcal{E}^\sharp \times \mathcal{E}^\sharp) \to \mathcal{E}^\sharp \\ \text{with } \forall A^\sharp, B^\sharp \in \mathcal{E}^\sharp,\ \gamma_\mathcal{E}(A^\sharp \triangledown_\mathcal{E} B^\sharp) \supseteq \gamma_\mathcal{E}(A^\sharp) \cup \gamma_\mathcal{E}(B^\sharp) \\ \text{and } \forall (Y_i^\sharp)_{i \in \mathbb{N}}, \text{ the sequence } X_0^\sharp \stackrel{\text{def}}{=} Y_0^\sharp,\ X_{i+1}^\sharp \stackrel{\text{def}}{=} X_i^\sharp \triangledown_\mathcal{E} Y_{i+1}^\sharp \\ \qquad\qquad \text{reaches a fixpoint } X_k^\sharp = X_{k+1}^\sharp \text{ for some } k \in \mathbb{N}. \end{array}$$

There exist many such abstract domains, for instance the interval domain [7], where an element of $\mathcal{E}^\sharp$ associates an interval to each variable, or the octagon domain [17], where an element of $\mathcal{E}^\sharp$ is a conjunction of constraints of the form $\pm X \pm Y \leq c$ with $X, Y \in \mathcal{V}$, $c \in \mathbb{R}$.

Given an abstract domain, we can provide an abstract semantics for non-atomic statements by induction, similarly to the concrete semantics (3), except that loops use the widening operator $\triangledown_\mathcal{E}$:

$$\begin{array}{l} \mathbb{S}[\![s_1;\ s_2]\!]^\sharp(R^\sharp, \Omega) \stackrel{\text{def}}{=} (\mathbb{S}[\![s_2]\!]^\sharp \circ \mathbb{S}[\![s_1]\!]^\sharp)(R^\sharp, \Omega) \\ \mathbb{S}[\![\textbf{if } e \textbf{ then } s]\!]^\sharp(R^\sharp, \Omega) \stackrel{\text{def}}{=} \\ \quad (\mathbb{S}[\![s]\!]^\sharp \circ \mathbb{S}[\![e \neq 0?]\!]^\sharp)(R^\sharp, \Omega) \ \cup^\sharp\ \mathbb{S}[\![e = 0?]\!]^\sharp(R^\sharp, \Omega) \\ \mathbb{S}[\![\textbf{while } e \textbf{ do } s]\!]^\sharp(R^\sharp, \Omega) \stackrel{\text{def}}{=} \\ \quad \mathbb{S}[\![e = 0?]\!]^\sharp(\text{lfp}\, \lambda X.\ X \triangledown ((R^\sharp, \Omega) \ \cup^\sharp\ (\mathbb{S}[\![s]\!]^\sharp \circ \mathbb{S}[\![e \neq 0?]\!]^\sharp)X)) \\ \mathbb{S}[\![f()]\!]^\sharp(R^\sharp, \Omega) \stackrel{\text{def}}{=} \mathbb{S}[\![\mathit{body}(f)]\!]^\sharp(R^\sharp, \Omega) \end{array} \tag{5}$$

where $(R_1^\sharp, \Omega_1) \cup^\sharp (R_2^\sharp, \Omega_2) \stackrel{\text{def}}{=} (R_1^\sharp \cup_{\mathcal{E}}^\sharp R_2^\sharp,\ \Omega_1 \cup \Omega_2)$ and $(R_1^\sharp, \Omega_1) \triangledown (R_2^\sharp, \Omega_2) \stackrel{\text{def}}{=} (R_1^\sharp \triangledown_{\mathcal{E}} R_2^\sharp,\ \Omega_1 \cup \Omega_2)$. The program semantics is, similarly to (4):

$$\mathbb{P}^\sharp \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{S}[\![\, \mathit{entry}() \,]\!]^\sharp(\mathcal{E}_0^\sharp, \emptyset) \ . \tag{6}$$

The following theorem states the soundness of the abstract semantics:

Theorem 2. $\mathbb{P} \subseteq \mathbb{P}^\sharp$.

As our programs have no recursive procedures, the recursion in $\mathbb{S}[\![\, \cdot \,]\!]^\sharp$ is bounded and we obtain an effective and sound static analysis. It is flow-sensitive and fully context-sensitive (behaving as if all function calls were inlined). It is relational whenever $\mathcal{E}^\sharp$ is (e.g., with octagons [17]). Moreover, the number of abstract elements to keep in memory does not depend on the program size but on the maximum nesting of conditionals and loops: the analyzer is thus very memory friendly, which is critical to analyze large programs, as in Astrée [5].

2.4 Concrete Path-Based Semantics $\mathbb{\Pi}$

We now propose an alternative concrete semantics based on control paths, which will come handy when considering parallel programs interleaving several threads. For non-parallel programs, its output is equal to that of the structured one.

A control path p is a finite sequence of atomic statements (i.e., $X \leftarrow e$, $e = 0?$, $e \neq 0?$). We denote by Π the set of all control paths. The set of paths $\pi(s) \subseteq \Pi$ of a statement s is defined as follows:

$$\begin{array}{l}
\pi(X \leftarrow e) \stackrel{\text{def}}{=} \{\, X \leftarrow e \,\} \\
\pi(s_1;\, s_2) \stackrel{\text{def}}{=} \pi(s_1); \pi(s_2) \\
\pi(\textbf{if } e \textbf{ then } s) \stackrel{\text{def}}{=} (\{\, e \neq 0? \,\}; \pi(s)) \cup \{\, e = 0? \,\} \\
\pi(\textbf{while } e \textbf{ do } s) \stackrel{\text{def}}{=} (\mathrm{lfp}\, \lambda X.\, \{\, \epsilon \,\} \cup (X; \{\, e \neq 0? \,\}; \pi(s))); \{\, e = 0? \,\} \\
\pi(f()) \stackrel{\text{def}}{=} \pi(\mathit{body}(f))
\end{array} \tag{7}$$

where ϵ denotes then empty path, and ; denotes path concatenation (by analogy with statement sequencing $s_1;\, s_2$) and is naturally extended to sets of paths. When s contains a loop, $\pi(s)$ is infinite, although many paths may be infeasible, i.e., have no corresponding execution (e.g., if all loops have a static bound).

Using the definitions from the structured semantics (3), we can define the semantics $\mathbb{\Pi}[\![\, P \,]\!]$ of a set of paths $P \subseteq \Pi$ as:

$$\mathbb{\Pi}[\![\, P \,]\!](R, \Omega) \stackrel{\text{def}}{=} \bigsqcup \{\, \mathbb{S}[\![\, s_1;\, \ldots;\, s_n \,]\!](R, \Omega) \mid s_1;\, \ldots;\, s_n \in P \,\} \tag{8}$$

which is similar to the standard *meet over all paths* solution[1] of data-flow problems [18, § 2], but for concrete executions in the infinite-height lattice $\mathcal{D}$. The meet over all paths and maximum fixpoint solutions of data-flow problems are equal for distributive frameworks; similarly, our structured and path-based concrete semantics (based on complete $\sqcup$−morphisms) are equal:

Theorem 3. $\forall s \in \mathit{stat},\ \mathbb{\Pi}[\![\, \pi(s) \,]\!] = \mathbb{S}[\![\, s \,]\!]$.

[1] The lattices used in data-flow analysis and abstract interpretation are dual: the former uses a meet to join paths while we employ a join.

3 Parallel Programs in Shared Memory

In this section, we consider several threads that communicate through a shared memory, without any synchronization primitive. We also discuss memory consistency models and their effect on the semantics and static analysis.

A program has now a fixed, finite set $\mathcal{T}$ of threads. Each thread $t \in \mathcal{T}$ has an entry-point function $entry_t \in \mathcal{F}$. All the variables in $\mathcal{V}$ are shared and can be accessed by all threads.[2]

3.1 Concrete Interleaving Semantics $\mathbb{P}_*$

The simplest and most natural model of parallel program execution considers all possible interleavings of control paths from all threads. These correspond to *sequentially consistent executions*, as coined by Lamport [15]. A parallel control path p is a finite sequence of pairs (s, t), where s is an atomic statement and $t \in \mathcal{T}$. The semantics $\mathbb{\Pi}_*[\![P]\!]$ of a set of parallel control paths P is:

$$\mathbb{\Pi}_*[\![P]\!](R, \Omega) \overset{\text{def}}{=} \bigsqcup \{ \mathbb{S}[\![s_1; \ldots; s_n]\!](R, \Omega) \,|\, (s_1, -); \ldots; (s_n, -) \in P \} \tag{9}$$

We denote by π_* the set of all parallel control paths in the program:

$$\pi_* \overset{\text{def}}{=} \{ p \,|\, \forall t \in \mathcal{T},\, \text{proj}_t(p) \in \pi(body(entry_t)) \} \tag{10}$$

where $\text{proj}_t(p)$ projects p on a thread t by extracting the maximal path $s_1; \ldots; s_n$ such that $(s_1, t); \ldots; (s_n, t)$ is a sub-path of p. The semantics $\mathbb{P}_*$ of the parallel program is then:

$$\mathbb{P}_* \overset{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{\Pi}_*[\![\pi_*]\!](\mathcal{E}_0, \emptyset) \ . \tag{11}$$

3.2 Concrete Interference Semantics $\mathbb{P}_{\mathcal{I}}$

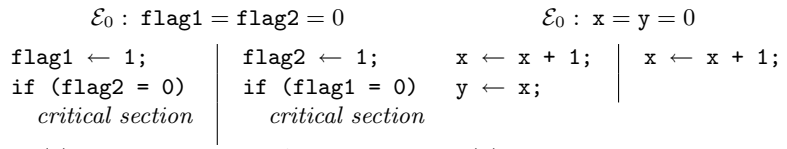
Because it reasons on infinite sets of paths, the interleaving concrete semantics is not easily amenable to flow-sensitive abstractions. We propose here a more abstract semantics that can be expressed by induction on the syntax and will lead to an effective static analysis after further abstraction.

We start by enriching the non-parallel semantics of Sec. 2.2 with a notion of interference. We call *interference* a triple $(t, X, v) \in \mathcal{I}$, where $\mathcal{I} \overset{\text{def}}{=} \mathcal{T} \times \mathcal{V} \times \mathbb{R}$, indicating that the thread t can set the variable X to the value v. The semantics of expressions is updated to take as extra arguments the current thread t and an interference set $I \subseteq \mathcal{I}$. When fetching a variable $X \in \mathcal{V}$, each interference on X from other threads is applied:

$$\mathbb{E}_{\mathcal{I}}[\![X]\!](t, \rho, I) \overset{\text{def}}{=} (\{ \rho(X) \} \cup \{ v \,|\, (t', X, v) \in I,\, t \neq t' \}, \emptyset) \tag{12}$$

while other functions are not changed with respect to (2), apart from propagating t and I recursively. As the interference is chosen non-deterministically, distinct occurrences of X in an expression may evaluate to different values. The semantics of statements is also enriched with interferences and is now a complete

[2] As the set of threads is finite, thread-local variables, such as function locals and parameters, could be handled by duplicating the functions and renaming the variables.



(a) Mutual Exclusion Algorithm. (b) Parallel Incrementation.

Fig. 2. Incompleteness examples for the interference semantics.

$\sqcup-$morphism in the complete lattice $\mathcal{D}_\mathcal{I} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I})$. The semantics of an assignment in a thread t both uses and enriches the interference set:

$$\begin{array}{l} \mathbb{S}_\mathcal{I}[\![X \leftarrow e,\, t]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\emptyset, \Omega, I) \sqcup \bigsqcup_{\rho \in R} (\{\, \rho[X \mapsto v] \mid v \in V \,\},\, \Omega',\, \{\, (t, X, v) \mid v \in V \,\}) \\ \qquad\qquad \text{where } (V, \Omega') = \mathbb{E}_\mathcal{I}[\![e]\!](t, \rho, I) \ . \end{array} \tag{13}$$

The other functions (not presented here) are easily derived: guards $e \bowtie 0$? pass I to $\mathbb{E}_\mathcal{I}[\![e]\!]$ and return I unchanged, while non-atomic statements are similar to (3), replacing $\mathbb{S}[\![\cdot]\!]$ with $\mathbb{S}_\mathcal{I}[\![\cdot]\!]$. Moreover, using $\mathbb{S}_\mathcal{I}[\![\cdot]\!]$ in (8) defines a path-based semantics with interference $\mathbb{\Pi}_\mathcal{I}[\![P,\, t]\!]$. Theorem 3 naturally becomes:

Theorem 4. $\forall t \in \mathcal{T},\, s \in \mathit{stat},\, \mathbb{\Pi}_\mathcal{I}[\![\pi(s),\, t]\!] = \mathbb{S}_\mathcal{I}[\![s,\, t]\!]$.

The semantics $\mathbb{S}_\mathcal{I}[\![s, t]\!]$ still considers a statement s from a single thread t. To take into account multiple threads, we iterate the analysis of all threads until errors and interferences are stable:

$$\begin{array}{l} \mathbb{P}_\mathcal{I} \stackrel{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \stackrel{\text{def}}{=} \\ \quad \mathrm{lfp}\, \lambda(\Omega, I).\, \bigsqcup_{t \in \mathcal{T}} \text{ let } (-, \Omega', I') = \mathbb{S}_\mathcal{I}[\![\mathit{entry}_t(),\, t]\!](\mathcal{E}_0, \Omega, I) \text{ in } (\Omega', I') \ . \end{array} \tag{14}$$

The interference semantics is sound with respect to the interleaving one (11):

Theorem 5. $\mathbb{P}_* \subseteq \mathbb{P}_\mathcal{I}$.

However, it is generally not complete. Consider, for instance the program fragment in Fig. 2(a) inspired from Dekker's mutual exclusion algorithm. According to the interleaving semantics, both threads can never be in their critical section simultaneously. The interference semantics, however, allows thread 1 to read `flag2` as either 0 (from $\mathcal{E}_0$) or 1 (from interferences) at any program point, and likewise for thread 2 and `flag1`, and so, there is no mutual exclusion. In Fig. 2(b), two threads increment the same zero-initialized variable `x`. According to the interleaving semantics, either the value 1 or 2 is stored into `y`. However, in the interference semantics, the fixpoint builds a growing set of interferences, up to $\{\, (t, \mathtt{x}, i) \mid t \in \mathcal{T},\, i \in \mathbb{N} \,\}$, as each thread increments the possible values written by the other thread, resulting in any positive value being written into `y`.

3.3 Abstract Interference Semantics $\mathbb{P}^\sharp_\mathcal{I}$

The concrete interference semantics is defined by structural induction. It can thus be easily abstracted. We assume, as in Sec. 2.3, the existence of an abstract

domain $\mathcal{E}^\sharp$ abstracting sets of environments, with a concretization $\gamma_\mathcal{E}$ and an element $\mathcal{E}_0^\sharp$ abstracting $\mathcal{E}_0$. Additionally, we assume the existence of an abstract domain $\mathcal{N}^\sharp$ that abstracts sets of reals. It is equipped with a concretization $\gamma_\mathcal{N} : \mathcal{N}^\sharp \to \mathcal{P}(\mathbb{R})$, a least element $\bot_\mathcal{N}^\sharp$ such that $\gamma_\mathcal{N}(\bot_\mathcal{N}^\sharp) = \emptyset$, an abstract join $\cup_\mathcal{N}^\sharp$ and, if it has strictly increasing infinite chains, a widening $\triangledown_\mathcal{N}$. Interferences are then abstracted using the domain $\mathcal{I}^\sharp \stackrel{\text{def}}{=} (\mathcal{T} \times \mathcal{V}) \to \mathcal{N}^\sharp$, with concretization $\gamma_\mathcal{I}(I^\sharp) \stackrel{\text{def}}{=} \{ (t, X, v) \,|\, v \in \gamma_\mathcal{N}(I^\sharp(t, X)) \}$, and $\cup_\mathcal{I}^\sharp$ and $\triangledown_\mathcal{I}$ defined point-wise. Abstract semantic functions now have the form: $\mathbb{S}_\mathcal{I}[\![\cdot]\!]^\sharp : \mathcal{D}_\mathcal{I}^\sharp \to \mathcal{D}_\mathcal{I}^\sharp$ with $\mathcal{D}_\mathcal{I}^\sharp \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L}) \times \mathcal{I}^\sharp$, and the soundness condition becomes:

$$(\mathbb{S}_\mathcal{I}[\![s,\, t]\!] \circ \gamma)(R^\sharp, \Omega, I^\sharp) \sqsubseteq (\gamma \circ \mathbb{S}_\mathcal{I}[\![s,\, t]\!]^\sharp)(R^\sharp, \Omega, I^\sharp)$$

where $\gamma(R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\gamma_\mathcal{E}(R^\sharp), \Omega, \gamma_\mathcal{I}(I^\sharp))$, i.e., the abstract function over-approximates environment, error, and interference sets.

Classic abstract domains can be easily converted to the interference semantics. Consider, for instance, the case of an assignment $(R^{\sharp\prime}, \Omega', I^{\sharp\prime}) = \mathbb{S}_\mathcal{I}[\![X \leftarrow e,\, t]\!]^\sharp(R^\sharp, \Omega, I^\sharp)$, when $\mathcal{N}^\sharp$ is the interval domain [7] and $\mathcal{E}^\sharp$ is an arbitrary domain. For each variable Y occurring in e, we compute its abstract interference: $Y^\sharp \stackrel{\text{def}}{=} \bigcup_\mathcal{N}^\sharp \{ I^\sharp(t', Y) \,|\, t \neq t' \}$. If $Y^\sharp \neq \bot_\mathcal{N}^\sharp$, then Y is substituted in e with the interval constant $Y^\sharp \cup_\mathcal{N}^\sharp itv_Y(R^\sharp)$ (where $itv_Y(R^\sharp)$ extracts the bounds of Y in the abstract environment $R^\sharp$) to get an expression e'. The result abstract environment and error set can now be computed using the native operators on $\mathcal{E}^\sharp$ as $(R^{\sharp\prime}, \Omega') = \mathbb{S}[\![X \leftarrow e']\!]^\sharp(R^\sharp, \Omega)$. Finally, $I^{\sharp\prime} = I^\sharp[(t, X) \mapsto itv_X(R^{\sharp\prime}) \cup_\mathcal{N}^\sharp I^\sharp(t, X)]$. Note that $\mathcal{I}^\sharp$ is not isomorphic to the interval domain [7]: the former abstracts $\mathcal{V} \to \mathcal{P}(\mathbb{R})$ and the later $\mathcal{P}(\mathcal{V} \to \mathbb{R})$. Sound abstractions for atomic statements then lift by induction on the syntax to sound abstractions for all statements, as in (5). Finally, an abstraction of the interference fixpoint (14) can be computed by iteration on abstract interferences, using a widening to ensure termination:

$$\begin{array}{l} \mathbb{P}_\mathcal{I}^\sharp \stackrel{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \stackrel{\text{def}}{=} \\ \quad \text{lfp}\, \lambda(\Omega, I^\sharp).\, \forall t \in \mathcal{T}, \text{ let } (-, \Omega_t', I_t^{\sharp\prime}) = \mathbb{S}_\mathcal{I}[\![entry_t(),\, t]\!]^\sharp(\mathcal{E}_0^\sharp, \Omega, I^\sharp) \text{ in} \\ \quad\quad (\bigcup_{t \in \mathcal{T}} \Omega_t',\; I^\sharp \triangledown_\mathcal{I} \bigcup_\mathcal{I}^\sharp \{ I_t^{\sharp\prime} \,|\, t \in \mathcal{T} \}) \;. \end{array} \tag{15}$$

The following theorem states the soundness of the analysis:

Theorem 6. $\mathbb{P}_\mathcal{I} \subseteq \mathbb{P}_\mathcal{I}^\sharp$.

The obtained analysis remains flow-sensitive and can be relational (provided that $\mathcal{E}^\sharp$ is relational) within each thread, but abstracts interferences in a flow-insensitive and non-relational way. It is expressed as an outer iteration that completely re-analyzes each thread until the abstract interferences stabilize, and so, can be implemented easily on top of existing non-parallel analyzers. Compared to a non-parallel program analysis, the cost is multiplied by the number of outer iterations required to stabilize interferences. This number remained very low in our experiments (Sec. 5). More importantly, the overall cost is not related to the (combinatorial) number of interleavings, but rather to the amount of abstract interferences $I^\sharp$, i.e., of actual communications between the threads. The speed of convergence can be controlled by adapting the widening $\triangledown_\mathcal{N}$.

3.4 Weakly Consistent Memory Semantics $\mathbb{P}'_*$

The interleaving concrete semantics $\mathbb{P}_*$ of Sec. 3.1, while simple, is not realistic. A first issue is that, as noted by Reynolds in [21], such a semantics requires choosing a level of granularity, i.e., some basic set of operations that are assumed to be atomic. In our case, we assumed assignments and guards to be atomic. In contrast, an actual system may schedule a thread within an assignment and cause `x` to be 1 at the end of the program in Fig. 2(b) instead of the expected value, 2. A second issue, noted by Lamport in [14], is that the latency of loads and stores in a shared memory may break the sequential consistency in true multiprocessor systems: threads running on different processors may not agree on the value of a shared variable. E.g., in Fig. 2(a), each thread may acknowledge the change of value of a flag after it has tested the other one, causing both critical sections to be entered simultaneously. Moreover, Lamport noted in [15] that reordering of independent loads and stores in one thread by the processor can also break sequential consistency (for instance performing the load from `flag2` after the store to `flag1` instead of before in the left thread of Fig. 2(a)). More recently, it has been observed [16] that optimizations in modern compilers have the same ill-effect even on mono-processor systems: program transformations that are perfectly safe on a thread considered in isolation (for instance, reordering the assignment `flag1` $\leftarrow 1$ and the test `flag2 = 0`) can cause non sequentially consistent behaviors to appear. In this section, we show that the interference semantics correctly handles these issues, by proving that it is invariant under a "reasonable" class of program transformations.

Acceptable program transformations of a thread are defined with respect to the path-based semantics $\mathbb{\Pi}$ of Sec. 2.4. A transformation of a thread t is acceptable if it gives rise to a set $\pi'(t) \subseteq \Pi$ of control paths such that every path $p' \in \pi'(t)$ can be obtained from a path $p \in \pi(\mathit{body}(\mathit{entry}_t))$ by a sequence of elementary transformations from Def. 1 below, $q \rightsquigarrow q'$ indicating that the statement sequence q in a path can be replaced with q'. These transformations can only reduce the amount of errors and interferences, so that an analysis of the original program is sound with respect to the transformed one. In Def. 1, we say that $X \in \mathcal{V}$ is fresh if it does not occur in any thread; $X \in \mathcal{V}$ is local if it occurs in the current thread only; $s[e'/e]$ is the statement s where some but not necessarily all occurrences of expression e may be changed into e'; $\mathrm{var}(e)$ is the set of variables appearing in e; $\mathrm{lval}(s)$ is the set of variables modified by s; $\mathrm{nonblock}(e)$ hods if evaluating e cannot block the program: $\forall \rho,\ \mathbb{E}[\![\, e \,]\!]\rho = (V, -)$ with $V \neq \emptyset$; e is deterministic if, moreover, $|V| = 1$; $\mathrm{noerror}(e)$ holds if evaluating e is always error-free: $\forall \rho,\ \mathbb{E}[\![\, e \,]\!]\rho = (-, \emptyset)$.

Definition 1 (Elementary path transformations).

1. *Redundant store elimination:* $X \leftarrow e_1; X \leftarrow e_2 \rightsquigarrow X \leftarrow e_2$ *when* $X \notin \mathrm{var}(e_2)$ *and* $\mathrm{nonblock}(e_1)$.
2. *Identity store elimination* $X \leftarrow X \rightsquigarrow \epsilon$.
3. *Reordering of assignments:* $X_1 \leftarrow e_1; X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2; X_1 \leftarrow e_1$ *when* $X_1 \notin \mathrm{var}(e_2)$, $X_2 \notin \mathrm{var}(e_1)$, *and* $\mathrm{nonblock}(e_1)$.

4. *Reordering of guards:* $e_1 \bowtie 0?; e_2 \bowtie 0? \ \rightsquigarrow \ e_2 \bowtie 0?; e_1 \bowtie 0?$
   when $\mathrm{noerror}(e_2)$.
5. *Reorder guard before assignment:* $X_1 \leftarrow e_1; e_2 \bowtie 0? \ \rightsquigarrow \ e_2 \bowtie 0?; X_1 \leftarrow e_1$
   when $X_1 \notin \mathrm{var}(e_2)$ *and either* $\mathrm{nonblock}(e_1)$ *or* $\mathrm{noerror}(e_2)$.
6. *Reorder assignment before guard:* $e_1 \bowtie 0?; X_2 \leftarrow e_2 \ \rightsquigarrow \ X_2 \leftarrow e_2; e_1 \bowtie 0?$
   when $X_2 \notin \mathrm{var}(e_1)$, X_2 *is local, and* $\mathrm{noerror}(e_2)$.
7. *Assignment propagation:* $X \leftarrow e; s \ \rightsquigarrow \ X \leftarrow e; s[e/X]$
   when $X \notin \mathrm{var}(e)$, $\mathrm{var}(e)$ *are local, and* e *is deterministic.*
8. *Sub-expression elimination:* $s_1; \ldots; s_n \ \rightsquigarrow \ X \leftarrow e; s_1[X/e]; \ldots; s_n[X/e]$
   when X *is fresh,* $\forall i$, $\mathrm{var}(e) \cap \mathrm{lval}(s_i) = \emptyset$, *and* $\mathrm{noerror}(e)$.
9. *Expression simplification:* $s \ \rightsquigarrow \ s[e'/e]$
   when $\forall \rho, I$, $\mathbb{E}_{\mathcal{I}}[\![\, e \,]\!](t, \rho, I) \sqsupseteq \mathbb{E}_{\mathcal{I}}[\![\, e' \,]\!](t, \rho, I)$.

Store latency can be simulated using rules 8 and 3. Breaking a statement into several ones (i.e., reducing the granularity of atomicity) is possible with rules 7 and 8. Global optimizations, such as constant propagation and folding, can be achieved using rules 7 and 9, while rules 1–6 allow peephole optimizations. Additionally, thread transformations that respect the set of control paths (such as loop unrolling or function inlining) are acceptable.

Given the set of transformed paths $\pi'(t)$, the interleaved executions π'_* and the semantics $\mathbb{P}'_*$ can be defined as in (10), (11):

$$\begin{array}{l} \pi'_* \stackrel{\text{def}}{=} \{\, p \,|\, \forall t \in \mathcal{T},\ \mathrm{proj}_t(p) \in \pi'(t) \,\} \\ \mathbb{P}'_* \stackrel{\text{def}}{=} \Omega, \ \text{where } (-, \Omega) = \mathbb{\Pi}_*[\![\, \pi'_* \,]\!](\mathcal{E}_0, \emptyset) \ . \end{array} \tag{16}$$

The following theorem extends Thm. 5 to transformed programs:

Theorem 7. $\mathbb{P}'_* \subseteq \mathbb{P}_{\mathcal{I}}$.

However, it is not complete. The two semantics coincide, for instance, in the program of Fig. 2(a). However, in the case of Fig. 2(b), the interference semantics assumes that `y` can take any positive value, while the interleaving semantics after program transformation still only allows the values 1 and 2. Note also that Thm. 7 holds for our "reasonable" collection of program transformations, but may not hold when considering "unreasonable" ones. For instance, `flag1 ← 1` should not be replaced (e.g., by a misguided prefetching optimizer) with `flag1 ← 42; flag1 ← 1` in Fig. 2(a), as this would cause the value 42 to be possibly seen by the other thread. Our interference semantics disallows such "out-of-thin-air" values to be introduced. This is consistent with other semantics, such as the Java one [16,22]. Another example of invalid transformation is the reordering of assignments $X_1 \leftarrow e_1; X_2 \leftarrow e_2 \ \rightsquigarrow \ X_2 \leftarrow e_2; X_1 \leftarrow e_1$ when e_1 may block the program (e.g., due to a division by 0) as the transformed program could expose errors in e_2 that cannot occur in the original program. Nevertheless, Def. 1 is not exhaustive and could be extended with some other "reasonable" transformations.

4 Parallel Programs With a Scheduler

The language and semantics of the preceding section are now extended to handle explicit synchronization primitives and a real-time scheduler.

4.1 Priorities and Synchronization Primitives

We denote by $\mathcal{M}$ a finite, fixed set of non-recursive mutual exclusion locks, so-called *mutexes*. The language (1) of Sec. 2.1 is enriched with primitives to control mutexes and scheduling as follows:

$$\begin{array}{lll} stat ::= \mathbf{lock}(m) & \textit{(mutex locking, } m \in \mathcal{M}\textit{)} \\ \quad | \ \mathbf{unlock}(m) & \textit{(mutex unlocking, } m \in \mathcal{M}\textit{)} \\ \quad | \ X \leftarrow \mathbf{islocked}(m) & \textit{(mutex testing, } X \in \mathcal{V}\textit{, } m \in \mathcal{M}\textit{)} \\ \quad | \ \mathbf{yield} & \textit{(thread pause)} \end{array} \tag{17}$$

These new statements are considered to be atomic and the set of paths of a program (7) is extended by stating $\pi(s) \stackrel{\text{def}}{=} \{\, s \,\}$ for them. We also assume that threads have fixed and distinct priorities. Thus, we denote threads in $\mathcal{T}$ simply by numbers from 1 to $|\mathcal{T}|$, being understood that thread t has a strictly higher priority than thread t' when $t > t'$.

Our scheduling model is that of real-time processes, found in embedded systems (e.g, the ARINC 653 specification [2]) and as an extension to POSIX threads. Moreover, we consider that a single thread can execute at a given time (e.g., when all threads share a single processor). In this model, the unblocked thread with the highest priority always runs. All threads start unblocked but may block voluntarily by locking a mutex that is already locked or by yielding, which allows lower priority threads to run. Yielding denotes blocking for a non-deterministic amount of time, which is useful to model timers (as we abstract away actual time) or waiting for some external resource. A lower priority thread can be preempted when unlocking a mutex if a higher priority thread is waiting for this mutex. It can also be preempted at any point by a yielding higher priority thread that wakes up non-deterministically. Thus, we cannot assume that a blocked thread is necessarily waiting at a synchronization statement.

This scheduling model is precise enough to take into account fine mutual exclusion properties that would not hold if we considered arbitrary preemption or true parallel executions on concurrent processors (as found, e.g, in desktops). For instance, in Fig. 3, the high priority thread avoids a call to **lock** / **unlock** by testing with **islocked** whether the low priority thread acquired the lock and, if not, executes its critical section (modifying Y and Z) confident that the low priority thread cannot execute and enter its critical section before the high priority thread explicitly **yield**s. Such reasoning is required to analyze precisely our target application (Sec. 5), and requires the real-time scheduler and single-processor hypotheses assumed in this section.

4.2 Concrete Scheduled Interleaving Semantics $\mathbb{P}_{\mathcal{H}}$

We now refine the semantics of Sec. 3 to take scheduling into account, starting with the concrete interleaving semantics $\mathbb{P}_*$ of Sec. 3.1. Interleavings that do not respect mutual exclusion or priorities are excluded, and thus, we observe fewer behaviors. This is materialized by the dotted $\subseteq$ arrows between concrete semantics in Fig. 1 (no such property holds for abstract semantics as they are generally non-monotonic due to the use of widenings).

low priority	high priority
lock(m);	$X \leftarrow$ **islocked**(m);
$Y \leftarrow 1$;	**if** $X = 0$ **then**
$Z \leftarrow 1$;	$Z \leftarrow 2$;
$T \leftarrow Y - Z$;	$Y \leftarrow 2$;
unlock(m);	**yield**;

Fig. 3. Using priorities to ensure mutual exclusion.

We define a domain of scheduler states $\mathcal{H}$ that associates to each thread whether it is ready, yielding, or waiting for some mutex, as well as the set of mutexes it holds: $\mathcal{H} \stackrel{\text{def}}{=} (\mathcal{T} \to \{\, \mathit{ready}, \mathit{yield}, \mathit{wait}(m) \mid m \in \mathcal{M} \,\}) \times (\mathcal{T} \to \mathcal{P}(\mathcal{M}))$. The domain of statements becomes: $\mathcal{D}_{\mathcal{H}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{H} \times \mathcal{E}) \times \mathcal{P}(\mathcal{L})$. The semantics of atomic statements is decomposed into three steps. Firstly, the function $\text{enabled}_t : \mathcal{D}_{\mathcal{H}} \to \mathcal{D}_{\mathcal{H}}$ that keeps only the states where a given thread t can run:

$$\text{enabled}_t(R, \Omega) \stackrel{\text{def}}{=} (\{\, ((b, l), \rho) \in R \mid b(t) = \mathit{ready} \wedge \forall t' > t,\, b(t') \neq \mathit{ready} \,\}, \Omega)\ .$$

Secondly, the semantic function $\mathbb{S}'_{\mathcal{H}}[\![\, s, t \,]\!]$ for atomic statements s in thread t:

$$\begin{array}{l}
\mathbb{S}'_{\mathcal{H}}[\![\, \textbf{yield}, t \,]\!](R, \Omega) \stackrel{\text{def}}{=} (\{\, ((b[t \mapsto \mathit{yield}], l), \rho) \mid ((b, l), \rho) \in R \,\}, \Omega) \\
\mathbb{S}'_{\mathcal{H}}[\![\, \textbf{lock}(m), t \,]\!](R, \Omega) \stackrel{\text{def}}{=} (\{\, ((b[t \mapsto \mathit{wait}(m)], l), \rho) \mid ((b, l), \rho) \in R \,\}, \Omega) \\
\mathbb{S}'_{\mathcal{H}}[\![\, \textbf{unlock}(m), t \,]\!](R, \Omega) \stackrel{\text{def}}{=} (\{\, ((b, l[t \mapsto l(t) \setminus \{\, m \,\}]), \rho) \mid ((b, l), \rho) \in R \,\}, \Omega) \\
\mathbb{S}'_{\mathcal{H}}[\![\, X \leftarrow \textbf{islocked}(m), t \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad (\{\, ((b, l), \rho[X \mapsto 0]) \mid ((b, l), \rho) \in R,\, \forall t' \in \mathcal{T},\, m \notin l(t') \,\} \cup \\
\quad \{\, ((b, l), \rho[X \mapsto 1]) \mid ((b, l), \rho) \in R,\, \exists t' \in \mathcal{T},\, m \in l(t') \,\}, \Omega) \\
\mathbb{S}'_{\mathcal{H}}[\![\, s, t \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad (\{\, ((b, l), \rho') \mid ((b, l), \rho) \in R,\, (R', -) = \mathbb{S}[\![\, s \,]\!](\{\, \rho \,\}, \Omega),\, \rho' \in R' \,\}, \Omega') \\
\quad \text{where } (-, \Omega') = \mathbb{S}[\![\, s \,]\!](R, \Omega), \text{ for all other statements } s, \text{ using (3).}
\end{array}$$

Thirdly, a scheduler step that wakes up yielding threads non-deterministically and gives each available mutex to the highest priority thread waiting for it:

$$\begin{array}{l}
\text{sched}(R, \Omega) \stackrel{\text{def}}{=} (\{\, ((b', l'), \rho) \mid ((b, l), \rho) \in R \,\}, \Omega), \text{ where} \\
\quad \forall t, \text{ if } b(t) = \mathit{wait}(m) \wedge \forall t' \neq t,\, m \notin l(t') \wedge \forall t' > t,\, b(t') \neq \mathit{wait}(m) \\
\qquad \text{then } b'(t) = \mathit{ready} \text{ and } l'(t) = l(t) \cup \{\, m \,\} \\
\qquad \text{else } l'(t) = l(t) \text{ and } b'(t) = b(t) \vee (b'(t) = \mathit{ready} \wedge b(t) = \mathit{yield})\ .
\end{array}$$

The semantics of an atomic statement s in a thread t combines all three steps:

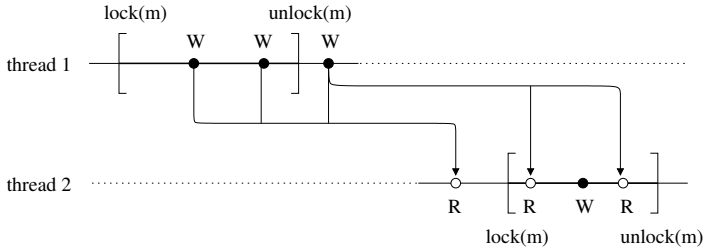
$$\mathbb{S}_{\mathcal{H}}[\![\, s, t \,]\!] \stackrel{\text{def}}{=} \text{sched} \circ \mathbb{S}'_{\mathcal{H}}[\![\, s, t \,]\!] \circ \text{enabled}_t\ . \tag{18}$$

The semantics $\mathbb{\Pi}_{\mathcal{H}}[\![\, P \,]\!]$ of a set P of interleaved paths and the semantics $\mathbb{P}_{\mathcal{H}}$ of the program are then defined, similarly to Sec. 3.1, (9)–(11), as:

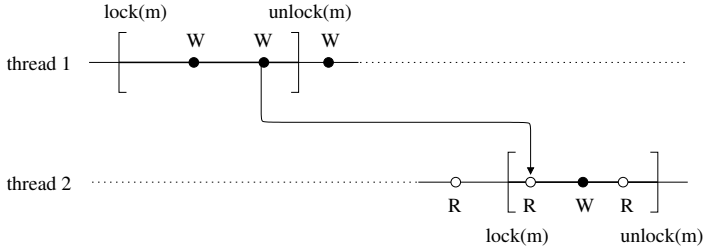
$$\begin{array}{l}
\mathbb{\Pi}_{\mathcal{H}}[\![\, P \,]\!](R, \Omega) \stackrel{\text{def}}{=} \\
\quad \bigsqcup \{\, (\mathbb{S}_{\mathcal{H}}[\![\, s_n, t_n \,]\!] \circ \cdots \circ \mathbb{S}_{\mathcal{H}}[\![\, s_1, t_1 \,]\!])(R, \Omega) \mid (s_1, t_1); \ldots; (s_n, t_n) \in P \,\}
\end{array} \tag{19}$$

$$\mathbb{P}_{\mathcal{H}} \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{\Pi}_{\mathcal{H}}[\![\, \pi_* \,]\!](\{h_0\} \times \mathcal{E}_0, \emptyset)$$

where $h_0 \stackrel{\text{def}}{=} (\lambda t.\, \mathit{ready}, \lambda t.\, \emptyset)$ denotes the initial scheduler state. As in Sec. 3.1, many control paths in π_* are unfeasible, i.e., return an empty set of environments,



(a) Weakly consistent interferences.



(b) Well synchronized interferences.

Fig. 4. Weakly consistent versus well synchronized scheduled interferences.

some of which are now ruled-out by the enabled$_t$ function. Nevertheless, errors from a feasible prefix of an unfeasible path are still taken into account. This includes, in particular, any error that occurs before a deadlock.

4.3 Scheduled Weakly Consistent Memory Semantics $\mathbb{P}'_{\mathcal{H}}$

In addition to restricting the interleaving of threads, synchronization primitives also have an effect when considering weakly consistent memory semantics: they enforce some form of sequential consistency at a coarse granularity level. More precisely, the compiler and processor handle synchronization statements specially, introducing the necessary flushes into memory and register reloads, and refraining from optimizing across them.

We thus adapt the semantics $\mathbb{P}'_*$ of Sec. 3.4 as follows. We consider a transformed thread as a set of paths $\pi'(t)$ obtained from $\pi(\mathit{body}(\mathit{entry}_t))$ using elementary path transformations from Def. 1, but no transformation should cross any synchronization primitive $\mathbf{lock}(m)$, $\mathbf{unlock}(m)$, **yield** or $X \leftarrow \mathbf{islocked}(m)$. Let π'_* be defined as before as the interleaving of paths from all $\pi'(t)$. The scheduled weakly consistent memory semantics is, based on (19):

$$\mathbb{P}'_{\mathcal{H}} \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{\Pi}_{\mathcal{H}}[\![\, \pi'_* \,]\!](\{h_0\} \times \mathcal{E}_0, \emptyset) \ . \tag{20}$$

4.4 Concrete Scheduled Interference Semantics $\mathbb{P}_{\mathcal{C}}$

We now provide a structured version of the scheduled interleaving semantics $\mathbb{P}_{\mathcal{H}}$. Similarly to Sec. 3.2, it is based on a notion of interferences, and it is not

complete. To avoid considering interferences between parts of threads that are in mutual exclusion, interferences are partitioned with respect to a thread-local view of scheduler configurations. The finite set of configurations $\mathcal{C}$ is defined as:

$$\mathcal{C} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \times \{\, \mathit{weak}, \mathit{sync}(m) \,|\, m \in \mathcal{M} \,\}$$

where the first subset of $\mathcal{M}$ denotes the mutexes locked by the thread while the second one denotes the mutexes held by no thread at all (as tested with **islocked**). The last component in $\mathcal{C}$ allows distinguishing between two kinds of interferences, which are depicted in Fig. 4: weakly consistent interferences (*weak* component in $\mathcal{C}$) corresponding to read / write pairs not protected by mutual exclusion (Fig. 4.(a)), and well synchronized interferences ($\mathit{sync}(m)$ component in $\mathcal{C}$) where both the read and the write are protected by the same mutex m (Fig. 4.(b)). Weakly consistent interferences behave as in Sec. 3.2. For well synchronized accesses, only the last write before unlocking a mutex affects a read, and only until the variable read is overwritten while the mutex is held. The partitioned domain of interferences is then: $\mathcal{I} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{C} \times \mathcal{V} \times \mathbb{R}$.

The semantics of a variable X read from an environment $\rho \in \mathcal{E}$ by a thread t is similar to (12), but we only apply the weakly consistent interferences from $I \subseteq \mathcal{I}$ that are not in mutual exclusion with a current configuration $c \in \mathcal{C}$:

$$\begin{array}{l} \mathbb{E}_{\mathcal{C}}[\![\, X \,]\!](t, c, \rho, I) \stackrel{\text{def}}{=} \\ \quad (\{\, \rho(X) \,\} \cup \{\, v \,|\, (t', c', X, v) \in I,\, t \neq t \wedge \mathit{excl}(c, c') \,\},\, \emptyset) \\ \text{where } \mathit{excl}((l, u, s), (l', u', s')) \iff^{\text{def}} l \cap l' = u \cap l' = u' \cap l = \emptyset \wedge s = s' \ . \end{array} \tag{21}$$

Other constructions are handled as in (2), where t, c, I are passed unused and unchanged. To handle precisely the **islocked** primitive in code similar to Fig. 3, it is necessary to represent some relationship between environments and scheduler states. Hence, environments are also partitioned with respect to $\mathcal{C}$, although the third component of configurations is not used and always set to *weak*. The semantic domain of statements is now $\mathcal{D}_{\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{C} \times \mathcal{E}) \times \mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I})$, partially ordered by point-wise set inclusion. The semantics of assignments is similar to that of (13), applied point-wise to each configuration:

$$\begin{array}{l} \mathbb{S}_{\mathcal{C}}[\![\, X \leftarrow e,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} (\emptyset, \Omega, I) \sqcup \\ \quad \bigsqcup_{(c,\rho) \in R} (\{\, (c, \rho[X \mapsto v]) \,|\, v \in V \,\},\, \Omega',\, \{\, (t, c, X, v) \,|\, v \in V \,\}) \\ \qquad \text{where } (V, \Omega') = \mathbb{E}_{\mathcal{C}}[\![\, e \,]\!](t, c, \rho, I) \ . \end{array}$$

The semantics of synchronization primitives is as follows:

$$\begin{array}{l} \mathbb{S}_{\mathcal{C}}[\![\, \mathbf{lock}(m),\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\{\, ((l \cup \{\, m \,\}, \emptyset, s), \rho') \,|\, ((l, -, s), \rho) \in R,\, \rho' \in \mathit{in}(t, l, m, \rho, I) \,\}, \\ \quad \Omega,\, I \cup \bigcup \{\, \mathit{out}(t, l, m', \rho) \,|\, ((l, u, -), \rho) \in R,\, m' \in u \,\}) \\ \mathbb{S}_{\mathcal{C}}[\![\, \mathbf{unlock}(m),\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\{\, ((l \setminus \{\, m \,\}, u, s), \rho) \,|\, ((l, u, s), \rho) \in R \,\}, \\ \quad \Omega,\, I \cup \bigcup \{\, \mathit{out}(t, l \setminus \{\, m \,\}, m, \rho) \,|\, ((l, -, -), \rho) \in R \,\}) \\ \mathbb{S}_{\mathcal{C}}[\![\, \mathbf{yield},\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\{\, ((l, \emptyset, s), \rho) \,|\, ((l, -, s), \rho) \in R \,\}, \\ \quad \Omega,\, I \cup \bigcup \{\, \mathit{out}(t, l, m', \rho) \,|\, ((l, u, -), \rho) \in R,\, m' \in u \,\}) \end{array} \tag{22}$$

$$
\begin{array}{l}
\mathbb{S}_{\mathcal{C}}[\![\, X \leftarrow \textbf{islocked}(m),\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\
\quad (\{\, ((l, u \cup m_t^*, s), \rho'[X \mapsto 0]) \mid ((l, u, s), \rho) \in R,\, \rho' \in \mathit{in}(t, l, m, \rho, I) \,\} \cup \\
\quad \{\, ((l, u \setminus \{\, m \,\}, s), \rho[X \mapsto 1]) \mid ((l, u, s), \rho) \in R \,\}, \\
\quad \Omega,\, I \cup \{\, (t, c, X, v) \mid v \in \{\, 0, 1 \,\},\, (c, -) \in R \,\}) \\
\text{where } m_t^* \stackrel{\text{def}}{=} \{\, m \,\} \text{ if no thread } t' > t \text{ can lock } m, \text{ and } \emptyset \text{ otherwise} \\
\mathit{in}(t, l, m, \rho, I) \stackrel{\text{def}}{=} \\
\quad \{\, \rho' \mid \forall X \in \mathcal{V},\, \rho'(X) = \rho(X) \vee (t', (l', \emptyset, \mathit{sync}(m)), X, \rho'(X)) \in I, \\
\quad\quad t \neq t',\, l \cap l' = \emptyset \,\} \\
\mathit{out}(t, l, m, \rho) \stackrel{\text{def}}{=} \{\, (t, (l, \emptyset, \mathit{sync}(m)), X, \rho(X)) \mid X \in \mathcal{V} \,\}\ .
\end{array}
$$

The functions $\mathit{in}(t, l, m, \rho, I)$ and $\mathit{out}(t, l, m, \rho)$ respectively model entering and exiting a critical section protected by a mutex m in a thread t that also holds the mutexes in l: *out* collects a set of well synchronized interferences from an environment, while *in* applies them to an environment. Obviously, $\textbf{lock}(m)$ uses *in* on m, while **unlock** uses *out*. Additionally, $X \leftarrow \textbf{islocked}(m)$ creates two partitions: one where $X = 1$, and one where $X = 0$ and m is assumed to be unlocked, which is remembered in u. However, the assumption that m is unlocked only stands if the thread cannot be preempted at any point by a higher priority thread locking m, hence the side-condition on m_t^* — in practice, this condition is checked by remembering in the semantics which **lock**s are issued by each thread; we do not present this for lack of space. Moreover, it stands only until the thread calls a blocking primitive (i.e., **lock** or **yield**), which gives the opportunity to a lower priority thread to lock m. Thus, blocking primitives use *out* to exit critical sections protected by all $m \in u$. Note that by replacing $X \leftarrow \textbf{islocked}(m)$ with $X \leftarrow [0, 1]$, we would obtain a less precise semantics, but which is also sound for true parallel or non real-time systems. We do not show the semantics of guards and non-atomic statements; they can be derived from (3) easily. The semantics $\mathbb{P}_{\mathcal{C}}$ of a program has the same fixpoint form as (14):

$$
\begin{array}{l}
\mathbb{P}_{\mathcal{C}} \stackrel{\text{def}}{=} \Omega,\ \text{where } (\Omega, -) \stackrel{\text{def}}{=} \operatorname{lfp} \lambda(\Omega, I). \\
\quad \bigsqcup_{t \in \mathcal{T}} \text{ let } (-, \Omega', I') = \mathbb{S}_{\mathcal{C}}[\![\, \mathit{entry}_t(),\, t \,]\!](\{c_0\} \times \mathcal{E}_0, \Omega, I) \text{ in } (\Omega', I')
\end{array}
\tag{23}
$$

where the initial configuration is $c_0 \stackrel{\text{def}}{=} (\emptyset, \emptyset, \mathit{weak}) \in \mathcal{C}$.

The semantics is sound with respect to that of Secs. 4.2–4.3:

Theorem 8. *$\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$ and $\mathbb{P}'_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.*

As in Sec. 3.4, this semantics is not complete. An additional loss of precision comes from the handling of well synchronized accesses. A main limitation is that such accesses are handled in a non-relational way, hence $\mathbb{P}_{\mathcal{C}}$ cannot represent relations enforced at the boundaries of critical sections but broken within, while $\mathbb{P}_{\mathcal{H}}$ can. For instance, in Fig. 3, we cannot prove that $Y = Z$ holds outside critical sections, but only that $Y, Z \in [1, 2]$. This shows in particular that even programs without data-races have behaviors in $\mathbb{P}_{\mathcal{C}}$ outside the sequentially consistent ones. Yet, we can prove that $T = 0$, i.e., the assignment to T is free from interference. Our implementation is actually a little smarter than (22) and uses a modified *out* that does not consider interferences for variables not modified while m is held. Finally, our implementation can also report data-races by simply inspecting the set of interferences during each assignment.

4.5 Abstract Scheduled Interference Semantics $\mathbb{P}^\sharp_{\mathcal{C}}$

The interference semantics with scheduler $\mathbb{P}_{\mathcal{C}}$ can be abstracted similarly to $\mathbb{P}_{\mathcal{I}}$. As in Sec. 3.3, we assume the existence of two abstract domains $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$ abstracting respectively $\mathcal{P}(\mathcal{E})$ and $\mathcal{P}(\mathbb{R})$. We lift these domains by partitioning under $\mathcal{C}$: $\mathcal{D}^\sharp_{\mathcal{C}} \stackrel{\text{def}}{=} (\mathcal{C} \to \mathcal{E}^\sharp) \times \mathcal{P}(\mathcal{L}) \times \mathcal{I}^\sharp$, where abstract interferences are in $\mathcal{I}^\sharp \stackrel{\text{def}}{=} (\mathcal{T} \times \mathcal{C} \times \mathcal{V}) \to \mathcal{N}^\sharp$. The concretization is: $\gamma(R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\{\, (c, \rho) \,|\, \rho \in \gamma_{\mathcal{E}}(R^\sharp(c)) \,\}, \Omega, \{\, (t, c, X, v) \,|\, v \in \gamma_{\mathcal{N}}(I^\sharp(t, c, X)) \,\})$. Sound abstract transfer functions can be derived easily from those in $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$. For instance, the assignment is similar to that of Sec. 3.3, except that it is applied point-wise to each $R^\sharp(c)$ and it only considers the abstract interferences from the configurations not in mutual exclusion with c. Synchronisation primitives are implemented mostly by joining partitions (using $\cup^\sharp_{\mathcal{E}}$) and copying non-relational information between $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$ (for *in* and *out*). Transfer functions for non-atomic statements are derived as in (5). Finally, the abstract analysis $\mathbb{P}^\sharp_{\mathcal{C}}$ computes a fixpoint over the interferences identical to (15). The resulting analysis is sound:

Theorem 9. $\mathbb{P}_{\mathcal{C}} \subseteq \mathbb{P}^\sharp_{\mathcal{C}}$.

Due to partitioning, $\mathbb{P}^\sharp_{\mathcal{C}}$ is less efficient than $\mathbb{P}^\sharp_{\mathcal{I}}$. However, partitioned environments are mostly empty: Sec. 5 shows that, in practice, at most program points, $R^\sharp(c) = \bot^\sharp_{\mathcal{E}}$ except for a few of configurations. Partitioned interferences are less sparse because, being flow-insensitive, they accumulate information for configurations reachable from any program point. However, this is not problematic: as interferences are non-relational, a large number of partitions can be manipulated efficiently. Thanks to partitioning, the precision of $\mathbb{P}^\sharp_{\mathcal{C}}$ is much better than $\mathbb{P}^\sharp_{\mathcal{I}}$ in the presence of locks and priorities. For instance, $\mathbb{P}^\sharp_{\mathcal{C}}$ can discover that $T = 0$ in Fig. 3, while the analysis of Sec. 3.3 would only discover that $T \in [-1, 1]$ due to spurious interferences from the high priority thread.

5 Experimental Results

The abstract analysis of Sec. 4.5 has been implemented in Thésée, our prototype analyzer. It analyzes C without recursion nor dynamic memory allocation. It is sound, and checks for integer and float arithmetic overflows, divisions by zero, invalid array and pointer accesses, and assertion failures. It also reports data-races, but ignores other parallel-related hazards. In particular, it does not check for dead-locks nor unbounded priority inversions. In fact, they cannot occur in our target application as all locks have a timeout. Thésée is based on Astrée [5], a static analyzer for synchronous embedded C software, which was successfully applied to prove the absence of run-time errors in large critical control/command software from Airbus [10]. Thésée benefited directly from Astrée's numerous abstract domains and iteration strategies targeting embedded C code. The adaptation to the analysis of parallel programs, including the addition of the interference fixpoint iterator and the scheduler partitioning domain, required

adding approximately 6 KLines of code to the 100 KLines analyzer, and did not require any structural change.

Our target parallel application is another large program from Airbus consisting of 1.6 MLines of C code and 15 threads. It runs under an ARINC 653 real-time OS [2]. The code is quite complex as it mixes string formatting, list sorting, network protocols (e.g., TFTP), and automatically generated synchronous logic. The program was completed with a 2 500 line hand-written model of the ARINC 653 OS implementing the various API calls, in C enriched with analyzer-specific intrinsics (mutex lock, unlock, etc.).

The analysis currently takes 14h on our 2.66 GHz 64-bit intel server using one core. An important result is that only four iterations are required to stabilize abstract interferences. Moreover, there are a maximum of 52 partitions for abstract interferences and 4 partitions for abstract environments, so that the analysis fits in 32 GB of memory. The analysis generates around 7600 alarms. This high number is understandable: Thésée is naturally tuned for avionic control/command software as it inherits abstract domains $\mathcal{E}^\sharp$, $\mathcal{N}^\sharp$ from Astrée, but the analyzed program is not limited to control/command processing. We started adapting these domains and can already report some improvements compared to earlier experimental results (50h and 12000 alarms [4, § VI]), using the same iterator and scheduler partitioning. However, independently from the choice of abstract domains $\mathcal{E}^\sharp$, $\mathcal{N}^\sharp$, a better treatment of well synchronized interferences will surely be required to achieve zero false alarms. Following the design-by-refinement of Astrée [5], we have focused on the analysis of a single (albeit large and complex) real-life software and started refining the analyzer to lower the number of alarms.

6 Related Work

There are far too many works on the semantics and analysis of parallel programs to provide a fair survey and comparison here. Instead, we focus on a few works that, we hope, provide a fruitful comparison with ours.

The idea of attaching to each thread location a local invariant and to handle proofs of parallel programs as that of sequential programs with interferences dates back to the Hoare-style proof method of Owicki and Gries [19] and Lamport [13] and has been well studied since (see [9] for a modern account and a partial survey). It has been studied from an Abstract Interpretation point of view in [8] and applied to static analysis. Two examples are the analysis of C with POSIX threads by Carré and Hymans [6] and that of Java with its weak memory model by Ferrara [11]. Unlike those, we do not handle thread creation, but we do take into account scheduler properties. Fully flow-insensitive analyses, e.g. Steensgaard's popular points-to analysis [23], naturally handle parallel programs. Unfortunately, the level of accuracy required to prove safety properties demands the use of (at least partially) flow-sensitive and relational methods, which we do.

Model-checking also has a long history of verifying parallel systems, including recently weak memory models [3]. Partial order reduction methods [12] are used

to limit the number of interleavings to consider, with no impact on completeness. In contrast, we abstract the problem sufficiently so that no interleaving need to be considered at all, at the cost of completeness. Unlike context-bounded approaches [20], our method considers all executions until completion.

Weakly consistent memory models have been studied mostly for hardware [1]. Pugh pioneered its use in programming language semantics, culminating with the Java memory model [16]. It is described in terms of implicit conditions on interleaved execution traces and is quite complex. We chose instead a generative approach based on control path transformations matching closely optimization models, similarly to the work of Saraswat et al. [22]. Our focus is on models that are realistic and can be abstracted into interference semantics suitable for efficient static analysis.

7 Conclusion

We presented a static analysis to detect all run-time errors in embedded C software with several threads communicating through a shared memory with weak consistency and scheduled according to strict priorities. Our method is based on a notion of interferences and partitioning with respect to a scheduler state. It can be implemented on top of analyzers for sequential programs, leveraging a growing library of abstract domains. Promising early experimental results on real code demonstrate the scalability of the approach.

A broad avenue for future work is to bridge the gap between the interleaving semantics and its incomplete abstraction in terms of interferences. Abstracting well synchronized accesses in a non-relational way is a severe limitation that we wish to suppress. We also wish to add support for other synchronization primitives, such as condition variables and atomic variables, and exploit more properties of real-time schedulers. A more precise analysis may require the use of history-sensitive abstractions, an avenue we wish to explore. Moreover, more precise or more general interference semantics could be designed by adjusting the notion of weak memory consistency. Finally, we wish, in future work, to analyze errors specifically related to parallelism, such as dead-locks, live-locks, and priority inversions, including quantitative time-related properties (e.g., bounded priority inversions).

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp.*, 29(12):66–76, 1996.
2. Aeronautical Radio, Inc. (ARINC). ARINC 653. `http://www.arinc.com/`.
3. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *37th ACM SIGACT/SIGPLAN Symp. on Principles of Prog. Lang.*, pages 7–18. ACM, Jan. 2010.
4. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In

AIAA Infotech@Aerospace, number AIAA-2010-3385, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 196–207. ACM, June 2003.
6. J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, Oct. 2009.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Prog. Lang.*, pages 238–252. ACM, Jan. 1977.
8. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Prog. Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.
9. W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.
10. D. Delmas and J. Souyris. ASTRÉE: from research to industry. In *SAS'07*, volume 4634 of *LNCS*, pages 437–451. Springer, 22–24 Aug. 2007.
11. P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *TAP'08*, volume 4966 of *LNCS*, pages 116–133. Springer, 2008.
12. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.
13. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.
15. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. on Computers*, volume 28, pages 690–691. IEEE Comp. Soc., Sep. 1979.
16. J. Manson, B. Pugh, and S. V. Adve. The Java memory model. In *32nd ACM SIGPLAN/SIGACT Symp. on Principles of Prog. Lang.*, pages 378–391. ACM, Jan. 2005.
17. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
18. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Oct. 1999.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.
20. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
21. J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 35–48. Springer, Dec. 2004.
22. V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pages 161–172. ACM, Mar. 2007.
23. B. Steensgaard. Points-to analysis in almost linear time. In *23rd ACM SIGPLAN/SIGACT Symp. on Principles of Prog. Lang.*, pages 32–41. ACM, 1996.