Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs

Martin D. Schwarz[1], Helmut Seidl[1], Vesal Vojdani[2], and Kalmer Apinis[1]

[1] Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{schwmart,seidl,apinis}@in.tum.de
[2] Deptartment of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
vesal@cs.ut.ee

Abstract. Although priority scheduling in concurrent programs provides a clean way of synchronization, developers still additionally rely on hand-crafted schemes based on integer variables to protect critical sections. We identify a set of sufficient conditions for variables to serve this purpose. We provide efficient methods to verify these conditions, which enable us to construct an enhanced analysis of mutual exclusion in interrupt-driven concurrent programs. All our algorithms are build upon off-the-shelf inter-procedural analyses alone. We have implemented this approach for the analysis of automotive controllers, and demonstrate that it results in a major improvement in the precision of data race detection compared to purely priority-based techniques.

1 Introduction

Embedded computing is omnipresent in the automotive industry. Dedicated operating systems and standards, such as Autosar/OSEK[1, 11], have been created and are used by many car manufacturers. These operating systems provide sophisticated synchronization primitives, such as priority-driven scheduling and resource acquisition. Still, developers sometimes rely on hand-crafted mechanisms for ensuring safe concurrent execution, e.g., for synchronizing two cooperating interrupts that do not affect any further interrupts. One such mechanism is to use global program variables as *flags* whose values control the access to critical sections. Accordingly, any analysis of OSEK programs, such as [15], which only takes resources and priorities into account, will produce a large number of false alarms on real-world automotive code.

An example of a typical flag-based synchronization pattern used by developers is shown in Fig. 1. This program consists of two interrupt service routines that are executed by a priority driven scheduler. The low-priority interrupt I sets a flag f to 1 before entering its critical section and resets it to 0 afterwards, whereas the higher-priority interrupt Q first checks whether the flag f has been set and only enters its critical section if f equals 0. This ensures, in a priority-driven single-core concurrency setting, that the accesses to x will always be

K.L. McMillan and X. Rival (Eds.): VMCAI 2014, LNCS 8318, pp. 21–38, 2014.

```
int f = 0;                          /* Priority 1 */
int x = 0;                          isr_I () {
/* Priority 3 */                      f = 1;
isr_Q() {                             x++; /* Ix */
  if (f == 0)                         f = 0;
    x--; /* Qx */                   }
}
```

Fig. 1. Example code of program with a flag f

exclusive. Note that priorities are crucial for such non-symmetric idioms: The higher-priority interrupt Q may safely assume that it cannot itself be preempted by the lower-priority interrupt I between checking the flag and accessing the data. Conversely, having checked the flag, it would be redundant for the high priority interrupt to set and reset the flag. Idioms like this, when properly used and implemented, can indeed protect critical sections. Still, being hand-crafted, they are error-prone and may be rendered insufficient if further priority levels are introduced. If, for example, another interrupt R is introduced whose priority exceeds that of Q, but uses the same pattern to access the variable x, a potential race condition arises between interrupts Q and R.

Our goal in this paper therefore is to provide practical methods for identifying and analyzing a wide range of synchronization patterns based on global variables, in order to provide a more accurate data-race analysis.

In principle, interrupt-driven programs with priority-based scheduling on a single-core can be analyzed by interpreting interrupts as function calls possibly occurring at every program point [4, 15]. Practically, though, context- and flow-sensitive handling of the global program state is prohibitively expensive. The approach outlined in [15] is to arrive at a practical analysis of mutual exclusion for interrupt-driven programs by analyzing local data such as dynamic priorities and resources context-sensitively, while summarizing the global state into a single flow- and context-insensitive invariant. However, when global variables are used as flags, that approach is insufficient in precision.

The key contributions of this paper are, first, to identify general properties of global variables used as flags in a wide range of hand-crafted synchronization patterns, and second, using these properties to construct efficient dedicated analysis methods based on off-the-shelf inter-procedural analysis. In particular, the resulting analysis turns out to be *interrupt-modular*, meaning that each interrupt can be analyzed independently. In a second stage, the resulting summaries are combined to precisely and efficiently compute the set of values that a flag variable may take at any given program point. Finally, this information is exploited to ensure that two program points are not part of a data race.

In this paper we first consider a class of primitive flag-based synchronization patterns which allow a low-priority interrupt to protect its critical sections against cooperating interrupts from higher priority levels. For example, the synchronization pattern used in Fig. 1 falls into this class. For this restricted class of flags, it suffices to only consider the intra-interrupt value-sets of the flag

variable. For that, we verify that program points can only be part of a data race if the protecting flag variable has overlapping value sets. In a second step we generalize the conditions on flag variables in order to handle a richer class of synchronization patterns, while keeping interrupt-modularity. An example for such a pattern is provided in Fig. 2 (Section 5.1).

A critical feature of both techniques is that they are sound on all programs; that is, they do not merely assume that the flags are well-behaved. Instead, they also verify the assumptions on the flag variables themselves. Thus, the techniques can be applied one after the other on-demand to ensure race freedom when purely resource-based analyses fail. We have implemented the core technique in the Goblint analyzer [18]. This not only lead to a decrease in the number of warnings on a suite of toy examples, but also resulted in a drastic reduction of alarms in our key industrial benchmarks.

2 OSEK Model

An OSEK program consists of tasks and interrupts. Tasks are activated by direct calls or timers. Interrupts can occur at any time. The priority ceiling protocol [16] is used for scheduling. Tasks and interrupts have a static initial priority, which may be dynamically increased during execution by acquiring resources and possibly decreased again by releasing resources. The dynamic priority never drops below the static initial priority. An interrupt will preempt the running execution if its static priority exceeds both the static and dynamic priority of the running task or interrupt.

Since we do not have timing information, we treat time-triggered tasks as interrupts. For the purpose of this paper, we therefore consider a programming model which consists of a finite set of procedures Proc, a finite collection of interrupt routines Irpt, and a starting node T that serves as an initial task for interrupts to preempt. A procedure or interrupt g is given by a control flow graph (N_g, E_g). Let N denote the union of all sets N_g of program points, extended with T. Likewise, let E denote the union of all sets E_g of control flow edges. Entry and return nodes of g are denoted as s_g and r_g, respectively. Additionally we have a finite set of priorities $\mathbb{P} = \{0, \ldots, p_{max}\}$. In this paper, we assume that each node $u \in N$ is always reached with the same priority $\mathcal{P}(u) \in \mathbb{P}$, where the static initial priority of an interrupt $q \in \mathsf{Irpt}$ is attained at program point s_g and T is the only node with minimal priority, i.e., $\mathcal{P}(T) = 0$. Edges in E are triples (u, cmd, v) leading from program point u to program point v by means of the command cmd. The command cmd either is a basic statement s like an assignment $x = 42$; or a guard $(x\ != y)$?, or a procedure call $h()$ otherwise. For the purpose of this paper we restrict ourselves to global variables only. Therefore, procedures neither need parameters nor return values. The execution of basic statements is assumed to be atomic. How to deal with local variables and parameters is an orthogonal matter and discussed in [6].

In general, the assumption that the dynamic priority is a function of the program point only need not be met by arbitrary OSEK programs. The issue

of computing and tracking dynamic priorities, however, is orthogonal to the problem of dealing with flag variables. Therefore, we restrict ourselves to this simple setting, in particular, since it always can be achieved by introducing distinct copies of program points for distinct priorities. For a detailed discussion on computing context and resource aware priorities, see [15].

In order to define data races in a single-core concurrency setting, it is not enough to know that two accesses are both reachable. Whether two access are safe or not is decided by the path the program takes from one access to the other. Therefore, we have chosen a path-based concrete semantics for our model. A similar formalization can be found, e.g., in [8]. Here, a *path* is a sequence of edges where the empty path is denoted by ϵ. For two sets of paths M_1, M_2 the concatenation operator @ appends every path in M_2 to every path in M_1:

$$M_1 @ M_2 = \{\pi_1\pi_2 | \pi_1 \in M_1, \pi_2 \in M_2\}$$

In a first step we characterize the sets of *same-level* paths, i.e., paths leading from the entry node to the return node of the same procedure or interrupt as the least solution of the following constraint system:

$$\begin{array}{lll}
[\text{S0}] & \mathbf{S}[s_g] \supseteq \{\epsilon\} & g \in \mathsf{Irpt} \cup \mathsf{Proc} \\
[\text{S1}] & \mathbf{S}[v] \supseteq \mathbf{S}[u]@\{(u,s,v)\} & (u,s,v) \in E \\
[\text{S2}] & \mathbf{S}[v] \supseteq \mathbf{S}[u]@\mathbf{S}[r_h] & (u,h(),v) \in E \\
[\text{S3}] & \mathbf{S}[u] \supseteq \mathbf{S}[u]@\mathbf{S}[r_q] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q)
\end{array}$$

Given these sets of same-level paths, the sets of paths *reaching* program points from the initial node T are characterized by the following constraint system:

$$\begin{array}{lll}
[\text{R0}] & \mathbf{R}[T] \supseteq \{\epsilon\} & \\
[\text{R1}] & \mathbf{R}[v] \supseteq \mathbf{R}[u]@\{(u,s,v)\} & (u,s,v) \in E \\
[\text{R2}] & \mathbf{R}[v] \supseteq \mathbf{R}[u]@\mathbf{S}[r_h] & (u,h(),v) \in E \\
 & \mathbf{R}[s_h] \supseteq \mathbf{R}[u] & (u,h(),v) \in E \\
[\text{R3}] & \mathbf{R}[s_q] \supseteq \mathbf{R}[u] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q) \\
 & \mathbf{R}[u] \supseteq \mathbf{R}[u]@\mathbf{S}[r_q] & q \in \mathsf{Irpt},\ \mathcal{P}(u) < \mathcal{P}(s_q)
\end{array}$$

In the next step, we introduce a *value* semantics that allows to check whether a path is executable for a given initial state or not. Let D denote the set of program states σ which assign a value to every variable x, e.g. $\sigma\ x = 0$ The concrete value semantics $[\![\cdot]\!]_V : E \to D \dashrightarrow D$ defines for every basic s a *partial* transformer $[\![s]\!] : D \dashrightarrow D$, which we lift to control flow edges (u,s,v) by $[\![(u,s,v)]\!] = [\![s]\!]$. The partial state transformer $[\![\pi]\!]$ for a path $\pi = e_1 \dots e_n$ then is obtained as the composition of the state transformers $[\![e_i]\!]$ of the edges contained in π, i.e.,

$$[\![\pi]\!] = [\![e_n]\!] \circ \dots \circ [\![e_1]\!]$$

A path π is *executable* for an initial state σ if $[\![\pi]\!]\ \sigma$ is defined. Let M denote any set of paths. Then the set of paths in M which are executable for σ is given by $\Pi(M,\sigma) = \{\pi \in M \mid \pi \text{ is executable for } \sigma\}$.

Intuitively, a data race occurs at variable x if a low-priority interrupt q_1 accesses x in an edge reaching a program point v_1, at which point it is immediately interrupted by a higher-priority interrupt q and before returning to v_1 the execution reaches another access to x at a program point v_2. Note that, while q_1 is not allowed to continue, q still may itself be interrupted. Therefore v_2 does not necessarily belong to q. Instead, it could belong to another interrupt of even higher priority. This notion is formalized by the following definition.

Definition 1. *There is a data race at variable x with initial state σ if there exists an executable path $\pi \in \Pi(\mathbf{R}[v_2], \sigma)$ reaching some program point v_2 where*

$$\pi = \pi_1(\_, s_1, v_1)(s_q, \_, \_)\pi_2(\_, s_2, v_2)$$

such that the following holds:

(a) s_1 and s_2 are accesses to x;
(b) $\mathcal{P}(v_1) < \mathcal{P}(s_q)$, $\mathcal{P}(v_1) < \mathcal{P}(v_2)$ and also $\mathcal{P}(v_1) < \mathcal{P}(v)$ for every program point v occurring in π_2.

3 Inter-procedural Analysis of Flags

Consider the program in Fig. 1. Race freedom of this program can only be verified by taking the values of the global variable f into account. This variable is set to 1 to flag the critical section of the lower-priority interrupt, and reset to 0 again to signal that the critical section has been left. The information provided by f is respected by the higher-priority interrupt Q in that its value is tested against 0 before Q enters its own critical section. A first practical property of variables f used in such pattern therefore is:

Property 1. f is assigned constants only, f is checked against constants only, and the address of f is never taken.

Also, such synchronization patterns assume an application wide consent about the role of such a variable f and its values. This means that higher-priority interrupts will *respect* the value of f. This leads to our second property:

Property 2. Let p_f denote the least static priority of an interrupt where f is accessed. Then all interrupts of priority exceeding p_f leave f *intact*.

Property 2 means that the value of f before any interrupt q at priority $\mathcal{P}(s_q) > p_f$ equals the value of f after termination of q. Note that we do not rule out that the interrupt q may change the value of f, e.g., in order to protect its own critical section against even higher-priority interrupts as in Fig. 2. In this case, however, q must restore the old value before termination. Subsequently, variables f satisfying Properties 1 and 2 are called *flags*.

3.1 Intra-interrupt Flag Analysis

Let F denote the set of *candidate* flag variables, i.e., all variables which satisfy Property 1. For each $f \in F$ let $\mathbb{V}_f$ denote the set of possible values of f at run-time. Note that F together with all sets $\mathbb{V}_f$ can be easily constructed from the program text. For simplicity assume that for all $f \in F$, $\mathbb{V}_f \subseteq \mathbb{Z}$. Our goal is to compute for each program point u the set of all possible values of all flag variables f when reaching this program point. For this we compute for each flag f and each interrupt or procedure g a summary which maps the value of f before entering g to the set of possible values of f at a program point u in g. This analysis is not new. It is the enhancement of *simple constants* with simple guards [5, 20]. We first require summaries of basic edges. The abstract semantics $[\![\cdot]\!]^\sharp : E \to \mathbb{V}_f \to 2^{\mathbb{V}_f}$ for flag f is given by:

$$
\begin{aligned}
[\![(u, f = c;, v)]\!]^\sharp\ c' &= \{c\} \\
[\![(u, f \,\square\, c?, v)]\!]^\sharp\ c' &= \begin{cases} \{c'\} & \text{if } c' \,\square\, c \\ \emptyset & \text{otherwise} \end{cases} \\
[\![(u, \_, v)]\!]^\sharp\ c' &= \{c'\} \quad \text{otherwise}
\end{aligned}
$$

Here $\square$ represents any of the comparison operators $==, !=, <, >$. Due to Property 1, no other assignments or guards involving variable f may occur in the program. First, the effect of a procedure is given by the following constraint system where composition is lifted by $(f \circ g)\ c = \bigcup\{f\ c' \mid c' \in g\ c\}$.

$$
\begin{array}{lll}
[\mathrm{F}'0] & \mathbf{S}^\sharp[s_h] \sqsupseteq (c \mapsto \{c\}) & h \in \mathsf{Proc} \\
[\mathrm{F}'1] & \mathbf{S}^\sharp[v] \sqsupseteq [\![(u, s, v)]\!]^\sharp \circ \mathbf{S}^\sharp[u] & (u, s, v) \in E \\
[\mathrm{F}'2] & \mathbf{S}^\sharp[v] \sqsupseteq \mathbf{S}^\sharp[r_h] \circ \mathbf{S}^\sharp[u] & (u, h(), v) \in E
\end{array}
$$

Note that this is the characterization of the flag summary defined according to inter-procedural analysis only, i.e., as if no interrupts may occur. This is made possible due to Property 2 which implies that interrupts, at least when flag variables are concerned, have no impact. Using the inter-procedural summaries, the transformation of flag values from the beginning of the execution of an interrupt to any given program point is characterized by:

$$
\begin{array}{lll}
[\mathrm{F}0] & \mathbf{R}^\sharp[s_q] \sqsupseteq (c \mapsto \{c\}) & q \in \mathsf{Irpt} \\
[\mathrm{F}1] & \mathbf{R}^\sharp[v] \sqsupseteq [\![(u, s, v)]\!]^\sharp \circ \mathbf{R}^\sharp[u] & (u, s, v) \in E \\
[\mathrm{F}2] & \mathbf{R}^\sharp[v] \sqsupseteq \mathbf{S}^\sharp[r_h] \circ \mathbf{R}^\sharp[u] & (u, h(), v) \in E \\
[\mathrm{F}3] & \mathbf{R}^\sharp[s_h] \sqsupseteq \mathbf{R}^\sharp[u] & (u, h(), v) \in E
\end{array}
$$

In practice all flags would be analyzed simultaneously. This analysis is linear in the size of the program and the number of flags. For each flag f it is quadratic in the number of possible values of f. Formally, we show that the analysis is sound — given Property 2 holds.

Lemma 1. *For a program point u, interrupt q, flag f and initial state σ we have for all paths π from s_q to u:*

$$(\llbracket \pi \rrbracket\ \sigma)\ f \in \mathbf{R}^\sharp[u]\ (\sigma\ f)$$

Proof. The proof is by induction on the lengths of paths obtained as solutions of the system [R]. For the initialization constraints [S0], [R0] and [F'0], [F0] we use that $\epsilon \in \mathit{\Pi}(\mathbf{R}[T], \sigma) \subseteq \mathit{\Pi}(\mathbf{R}[s_q], \sigma)$ and with $\llbracket \epsilon \rrbracket\ \sigma = \sigma$ we obtain:

$$(\llbracket \epsilon \rrbracket\ \sigma)\ f\ =\ \sigma\ f\ \in\ \{\sigma\ f\}\ =\ \mathbf{R}^\sharp[s_q]\ (\sigma\ f)$$

Assuming the condition holds for a given path π, we consider how adding an edge (or summary path) and applying the corresponding constraint in the [F/F'] system may change the result. Due to Property 2 interrupts do not contribute to the values of f. Therefore we need not consider constraints [S3] and [R3]. For the extension of π with a basic edge the result follows as a direct consequence of $\llbracket \cdot \rrbracket^\sharp$ soundly abstracting $\llbracket \cdot \rrbracket$. This works for both procedure summaries constructed by [S1] and [F'1] or reaching information constructed by [R1] and [F1]. For same-level paths Constructed by [S2] and [F'2] the result then follows by induction. Consequently we also obtain the result for [R2] and [F2]. Since we know by induction that every program point u that influences a procedure entry node s_h is a sound approximation the result for s_h follows by monotonicity. □

In order to show which programs the analysis can handle precisely we introduce an abstraction of $\llbracket \cdot \rrbracket$ which only evaluates guards on flag f and denote it by $\llbracket \cdot \rrbracket_f$. E.g., $\llbracket (u, x == y, v) \rrbracket_f\ \sigma\ =\ \sigma$ for any σ, as the expression $x == y$ does not use the flag f, however $\llbracket (u, f == 5, v) \rrbracket_f\ \sigma\ =\ \sigma$ only in the case where $\sigma\ f$ equals 5 — otherwise it is undefined. A path π is *f-executable* for an initial state σ if $\llbracket \pi \rrbracket_f\ \sigma$ is defined. Let M denote any set of paths. Then the set of paths in M which are f-executable for σ is given by $\mathit{\Pi}_f(M, \sigma) = \{\pi \in M \mid \pi \text{ is } f\text{-executable for } \sigma\}$.

Lemma 2. *For a program point u, priority i, flag f and initial state σ we have*

$$\mathit{\Pi}_f(\mathbf{R}[u], \sigma) \neq \emptyset\ \Longrightarrow\ \mathbf{R}^\sharp[u]\ (\sigma\ f) \subseteq \bigcup_{\pi \in \mathit{\Pi}_f(\mathbf{R}[u], \sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

Proof. In case $\mathbf{R}^\sharp[u](\sigma\ f) = \emptyset$ the subset relation holds trivially. Otherwise we do induction on Kleene-iteration steps solving the constraint system [F/F']. First consider the initialization constraints [F0] and [F'0]. For entry nodes s_g we know that $\epsilon \in \mathit{\Pi}_f(\mathbf{R}[s_g], \sigma)$. Since $\llbracket \epsilon \rrbracket$ is the identity function we obtain:

$$\mathbf{R}^\sharp[s_g]\ (\sigma\ f) = \{\sigma\ f\} \subseteq (\llbracket \epsilon \rrbracket\ \sigma)\ f \subseteq \bigcup_{\pi \in \mathit{\Pi}_f(\mathbf{R}[s_g], \sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

For the application of constraints [F1] and [F'1] the result follows since for a flag $\llbracket \cdot \rrbracket^\sharp$ is complete with respect to $\llbracket \cdot \rrbracket$. For constraint [F'2] the result then follows

by induction and consequently for [F2]. Regarding [F3] we know by induction, that the condition holds for all program points on the right hand side of the constraint. We have:

$$\mathbf{R}^\sharp[s_h]\ (\sigma\ f) = \bigcup_{(u,h(),v)\in E} (\mathbf{R}^\sharp[u]\ (\sigma\ f)) \subseteq$$
$$\bigcup_{(u,h(),v)\in E} \bigcup_{\pi\in\Pi_f(\mathbf{R}[u],\sigma)} \{([\![\pi]\!]\ \sigma)\ f\} \subseteq \bigcup_{\pi\in\Pi_f(\mathbf{R}[s_h],\sigma)} \{([\![\pi]\!]\ \sigma)\ f\}$$

□

Together Lemmas 1 and 2 show, that if Property 2 holds we lose precision with respect to the concrete semantics only when guards on other variables influence the value of f. Making use of the general properties of OSEK programs we can bootstrap a check of Property 2 from the results of system [F/F'].

Theorem 1. *For a flag f Property 2 holds if for all interrupts q and initial states σ the following holds:*

$$\mathbf{R}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$$

Proof. Consider an interrupt q of maximal priority, i.e. $\mathcal{P}(s_q) = p_{max}$. For a program point u occurring in a path $\pi \in \Pi(\mathbf{R}[r_q], \sigma)$ the condition of constraint [S3] is never satisfied since $\mathcal{P}(u) \geq \mathcal{P}(s_q)$. Thus constraint [S3] does not contribute to $\mathbf{R}[r_q]$. Consequently Lemma 1 holds for r_q even when Property 2 is not generally satisfied. Therefore for $\pi \in \Pi(\mathbf{S}[r_q], \sigma)$ and $\mathbf{S}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$ we have:

$$\{([\![\pi]\!]\ \sigma)\ f\} = \mathbf{S}^\sharp[r_q]\ (\sigma\ f) = \{\sigma\ f\}$$

I.e. Property 2 holds for q. If Property 2 holds for all interrupts of maximal priority we can repeat this argument for the next lower priority and so on. □

4 Simple Analysis of Flags

In this Section we provide a first analysis of flag based synchronization patterns where no value of f is excluded before an interrupt occurs. Accordingly, we start the analysis of each interrupt with $\mathbb{V}_f$. Then by Lemma 1, the set of possible values of f at any program point u is given by

$$\mathbf{R}^\flat[u] = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in \mathbb{V}_f\}$$

Now consider a global variable x. Simple protection mechanisms for x by means of the flag f assume that the value of f after the protected access to x is *reliable*. Here, we call f reliable at a program point u, if its value may not be changed by any higher priority interrupts occurring at u. More formally f is reliable at u if for all interrupts q that may occur at u, i.e. $\mathcal{P}(s_q) > \mathcal{P}(u)$, and every program point v reachable from s_q without entering another interrupt we have:

$$\mathbf{R}^\sharp[v]\ c \subseteq \{c\} \quad \text{for all } c \in \mathbf{R}^\flat[u]$$

This test can be done in time linear in the size of the program and the number of values of f. The following Theorem shows how reliable values of a flag f can be used to verify the absence of data races.

Theorem 2. *Assume that u and v are end points of accesses to x where v is reached without entering another interrupt by an interrupt q with $\mathcal{P}(u) < \mathcal{P}(s_q)$ and flag f is reliable at program point u. Then no data race between u and v occurs, if $\mathbf{R}^\flat[u] \cap \mathbf{R}^\flat[v] = \emptyset$.*

Proof. Assume there is a race path going through u and v. Since the flag f is reliable at u, the sub-path π from u to v may not contain assignments changing f. Let σ be a state reaching u where π is executable for σ and $\sigma\ f = c \in \mathbf{R}^\flat[u]$. Let $\pi = \pi_1(s_q, \_, \_)\pi_2$ where π_2 contains no interrupt entry edges. Then the program state $\sigma_1 = [\![\pi_1]\!]\ \sigma$ after execution of π_1 as well as $[\![\pi_2]\!]\ \sigma_1$ will map f to c. Therefore by Lemma 1, c also must be included in $\mathbf{R}^\flat[v]$. Accordingly, $\mathbf{R}^\flat[u] \cap \mathbf{R}^\flat[v] \neq \emptyset$ — in contradiction to the assumption. □

In the program in Fig. 1, the flag f is reliable at Ix where $\mathbf{R}^\flat[Ix] = \{1\}$ and $\mathbf{R}^\flat[Qx] = \{0\}$. Accordingly due to Theorem 2, the program does not contain a data race. However for the pattern in Fig. 2, the flag is *not* reliable in the sense of our definition here. Therefore, Theorem 2 cannot excluded a data race for variable x.

5 Precise Analysis of Flags

The flag analysis from the last section is imprecise regarding two main points. First, all flag values are assumed to possibly occur at all entry points of interrupts. Second, only *reliable* flag values could be exploited for discarding potential races. In this section we therefore refine our approach by precisely tracking how flag values may change along *inter*-interrupt paths. Due to Property 2 the value analysis from Section 3 still provides valid intra-interrupt results for a given set of flag values at the beginning of the corresponding interrupt. In order to use these results to determine whether a value c is possible for a flag f at the start of an interrupt they are refined by additionally recording the minimal priorities at which f equals c. The minimal priority at which a flag f obtains a certain value is crucial since this is when the value is propagated to the largest number of other interrupts.

Accordingly, we now consider abstract values of the form $\mathbb{V}_f \to \mathbb{P}_\infty$, where $\mathbb{P}_\infty = \mathbb{P} \cup \{\infty\}$ and ∞ denotes, that a value is not obtained at all. The ordering is point-wise and reversed, i.e., $\bot$ maps all values to ∞ (no values are obtained) whereas $\top$ maps all values to 0 (all values can occur at all interrupt entry nodes). The following constraint system [C] accumulates for every interrupt and procedure the occurring values of flag f and the corresponding minimal priorities. In that constraint system, priorities are added to value summaries $\mathbf{R}[u]^\sharp$ of type $\mathbb{V}_f \to 2^{\mathbb{V}_f}$ by means of the functions pry_i which returns a function of type $\mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ and is defined by:

$$\mathsf{pry}_i\ \phi\ c\ c' = \begin{cases} i & \text{if } c' \in \phi\ c \\ \infty & \text{otherwise} \end{cases}$$

First we characterize functions $\mathbf{S}^*[r_h]$ (h a procedure) where $\mathbf{S}^*[r_h]\ c\ c'$ denotes the minimal priority at which the flag f could take the value c' along any f-executable path in h when the initial value of f is c. For this, we accumulate the values computed by $\mathbf{S}^\sharp[\cdot]$.

$$\begin{array}{lllll} [\text{C0}] & \mathbf{S}^*[s_h] & \sqsupseteq \mathsf{pry}_{\mathcal{P}(s_h)}\ \mathbf{S}^\sharp[s_h] & & h \in \mathsf{Proc} \\ [\text{C1}] & \mathbf{S}^*[v] & \sqsupseteq (\mathsf{pry}_{\mathcal{P}(v)}\ \mathbf{S}^\sharp[v]) \sqcup \mathbf{S}^*[u] & & (u, \_, v) \in E \\ [\text{C2}] & \mathbf{S}^*[v] & \sqsupseteq \mathbf{S}^*[r_h] \sqcup \mathbf{S}^*[u] & & (u, h(), v) \in E \end{array}$$

By means of these procedure summaries, we characterize mappings $\mathbf{R}^*[u]$, where $\mathbf{R}^*[u]\ c\ c'$ indicates the minimal priority at which flag f may have value c' along paths reaching u when the value of f equals c at the program point where the last interrupt before reaching u occurs. These are given by the least solution to the following constraint system:

$$\begin{array}{lllll} [\text{C3}] & \mathbf{R}^*[s_q] & \sqsupseteq \mathsf{pry}_{\mathcal{P}(s_q)}\ \mathbf{R}^\sharp[s_q] & & q \in \mathsf{Irpt} \\ [\text{C4}] & \mathbf{R}^*[v] & \sqsupseteq (\mathsf{pry}_{\mathcal{P}(v)}\ \mathbf{R}^\sharp[v]) \sqcup \mathbf{R}^*[u] & & (u, s, v) \in E \\ [\text{C5}] & \mathbf{R}^*[v] & \sqsupseteq \mathbf{S}^*[r_h] \sqcup \mathbf{R}^*[u] & & (u, h(), v) \in E \end{array}$$

Since the constraint system [C] is a pure join system (the only operator in right-hand sides is $\sqcup$), the least solution can be computed in time linear in the size of the program and polynomial in the number of flag values. The formal correctness of our construction is given by the following lemma.

Lemma 3. *For a program point u, flag f and initial state σ with $\sigma\ f = c$ the following holds:*

(1) For all paths $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ that do not contain any interrupt edges $(s_q, \_, \_)$ and program points v such that $\pi = \pi_1(\_, \_, v)\pi_2$ and $(\llbracket \pi_1(\_, \_, v) \rrbracket\ \sigma)\ f = c'$ we have $\mathbf{R}^*[u]\ c\ c' \leq \mathcal{P}(v)$.

(2) If $\mathbf{R}^*[u]\ c\ c' = p$ then there exists a path $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ that does not contain any interrupt edges $(s_q, \_, \_)$ and there exists a program point v such that $\pi = \pi_1(\_, \_, v)\pi_2$ and $(\llbracket \pi_1(\_, \_, v) \rrbracket\ \sigma)\ f = c'$ and $\mathcal{P}(v) = p$.

Proof. The proof is done by induction on the Kleene-iteration steps used to compute the fix-point. Since we use the same priority information in the systems [C] and [R] Lemmas 1 and 2 imply the result for the initialization constraints [C0] and [C3]. Again by Lemmas 1 and 2 and induction we know the result holds for both operands of a join. Soundness then follows directly. For the completeness statement (2) we note, that $\mathbb{P}_\infty$ is totally ordered. Therefore the join operator element-wise selects the result of one of its operands. □

Consider, e.g., the program in Fig. 1. For the return nodes of r_I and r_Q we have $\mathbf{R}^*[r_I]\ 0 = \{0 \mapsto 1, 1 \mapsto 1\}$ and $\mathbf{R}^*[r_Q]\ 0 = \{0 \mapsto 3, 1 \mapsto \infty\}$, respectively.

The results of the intra-interrupt accumulation of flag values and priorities are now combined to summarize the effects of all interrupts of a given priority. For each priority i, let I_i denote the least upper bound of $\mathbf{R}^*[r_q]$ for all interrupts q of priority i. Thus, $I_i\ c\ c'$ returns the least priority for which the flag f may obtain value c' inside an interrupt of priority i if the value of f is c before execution is interrupted (by an interrupt of priority i). Formally, we have:

Lemma 4. *For a program point u, flag f, initial state σ with $\sigma\ f = c$ and a path $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ with $\pi = (s_q, \_, \_)\pi'$ where π' contains no interrupt edges $(s_{q'}, \_, \_)$ and $([\![\pi]\!]\ \sigma)\ f = c'$, we have $I_{\mathcal{P}(s_q)}\ c\ c' \leq \mathcal{P}(u)$.*

Proof. By definition of I_i we have $I_i\ c\ c' \leq \mathbf{R}^*[r_q]\ c\ c'$ and from Lemma 3 we obtain $\mathbf{R}^*[r_q]\ c\ c' \leq \mathcal{P}(u)$. □

Conversely we can show, that the functions I_i are complete with respect to f-executable paths.

Lemma 5. *For a flag f and initial state σ with $\sigma\ f = c$ if $I_i\ c\ c' = j$ then there exists an interrupt q with $\mathcal{P}(s_q) = i$ and an path $\pi \in \Pi_f(\mathbf{R}[r_q], \sigma)$ such that $\pi = (s_q, \_, \_)\pi_1(\_, \_, v)\pi_2$, π_1 contains no interrupt edges $(s_{q'}, \_, \_)$, $([\![(s_q, \_, \_)\pi_1(\_, \_, v)]\!]\ \sigma)\ f =' c$ and $\mathcal{P}(v) = j$.*

Proof. Since $\mathbb{P}_\infty$ is totally ordered, the join on $\mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ can always select one of the joined values. Therefore we do not introduce finite priorities for values, which are not actually obtained during some interrupt. Let q denote the interrupt, such that $\mathbf{R}^*[r_q]\ c\ c' = j$. Lemma 3 then yields a path realizing the value c. □

In order to account for *inter*-interrupt effects, we first lift the mappings $I_i : \mathbb{V}_f \to \mathbb{V}_f \to \mathbb{P}_\infty$ to mappings $(\mathbb{V}_f \to \mathbb{P}_\infty) \to (\mathbb{V}_f \to \mathbb{P}_\infty)$ by

$$(\mathsf{lift}\ I_i)\ \delta = \delta \sqcup \bigsqcup \{I_i\ c \mid c \in \mathbb{V}_f,\ \delta\ c < i\}$$

Thus, the transformation $(\mathsf{lift}\ I_i)$ takes a mapping of flag values to priorities and returns the mapping, which maps each value c to the minimal priority at which flag f can attain value c if additionally interrupts at priority i are taken into account. Note that the application of $(\mathsf{lift}\ I_i)$ to a mapping δ can be represented as matrix operations and therefore computed in time polynomial in the number of possible values. In the final step, the inter-interrupt summaries $I_{(i,j)}$ comprising the effects of interrupt at priority levels between i and j are obtained from the lifted intra-interrupt summaries by composition:

$$I_{(i,j)} = (\mathsf{lift}\ I_j) \circ \ldots \circ (\mathsf{lift}\ I_i)$$

where for $i > j$, we assume $I_{(i,j)}$ to equal the identity transformation. The level summaries do not need to be composed in every possible order, since a higher priority interrupt must terminate before another interrupt of lower priority can execute. But on termination the higher priority interrupt must have restored

the values of flags, i.e. for the lower priority interrupt it might as well not have occurred. Therefore only chains of consecutive interruptions add new entries to the set of possible values of a flag f. Such chains, however, must have strictly increasing priorities. Let $\mathsf{init}_i : 2^{\mathbb{V}_f} \to (\mathbb{V}_f \to \mathbb{P}_\infty)$ denote the functions that given a set of values C returns the mapping which maps each value in C to i, i.e $\mathsf{init}_i\, C\, c = i$ for $c \in C$ and ∞ otherwise. Applying init_0 to the set of initial values of flag f reflects the fact, that initial values can occur at all interrupt entry nodes. Moreover, let $\mathsf{filter}_j : (\mathbb{V}_f \to \mathbb{P}_\infty) \to 2^{\mathbb{V}_f}$ denote the function which takes a mapping of flag values to priorities and returns the set of values of priorities less than j, i.e., $\mathsf{filter}_j\ \delta = \{c \mid \delta\ c < j\}$. Applying filter_j returns the set of flag values possible at entry nodes of interrupts of priority j according to the given mapping. With these definitions, we obtain the set of all possible values of flag f at entry nodes of interrupts with priority j by:

$$C_j = (\mathsf{filter}_j \circ I_{(1,j-1)} \circ \mathsf{init}_0)\ C_0$$

where C_0 is the set of possible values of f at program start. Accordingly, the set of all values of f possibly occurring at program point u when reached by an interrupt with priority j, is given by:

$$\mathbf{R}^\flat[u, j] = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_j\}$$

The complexity of applying $I_{(i,j)}$ to some initial mapping of values to priorities, requires to apply $(j - i)$ times a lifted transformation ($\mathsf{lift}\ I_k$). Therefore, this can be done in time polynomial in the number of possible flag values where the exponent linearly depends on $j - i$.

Theorem 3. *For a program point u reached by an interrupt with priority j, flag f and initial state σ with $\sigma\ f \in C_0$, we have:*

$$\mathbf{R}^\flat[u, j] \supseteq \bigcup_{\pi \in \Pi_f(\mathbf{R}[u], \sigma)} \{(\llbracket \pi \rrbracket\ \sigma)\ f\}$$

Proof. For a path $\pi \in \Pi_f(\mathbf{R}[u], \sigma)$ with $(\llbracket \pi \rrbracket\ \sigma)\ f = c''$ we show that $c'' \in \mathbf{R}^\flat[u, j]$. In case that $j = 1$, we have the following:

$$\mathbf{R}^\flat[u, j] = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_1\} = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in (\mathsf{filter}_1 \circ \mathsf{init}_0)\ C_0\} = \bigcup\{\mathbf{R}^\sharp[u]\ c \mid c \in C_0\} \supseteq \mathbf{R}^\sharp[u]\ (\sigma\ f)$$

With Lemma 1 the result follows. For higher context priorities we decompose π into the two parts. The part inside the final interrupt q and the part before q. Let $\pi = \pi'(\_, \_, v)(s_q, \_, \_)\pi''$ where $\mathcal{P}(s_q) = j$ and π'' does not contain any interrupt edges $(s_{q'}, \_, \_)$. Since Lemma 1 applies $(s_{q'}, \_, \_)\pi''$ again what's left to show is, that the values observable by $s_{q'}$ are contained in C_j. Let $(\llbracket \pi'(\_, \_, v) \rrbracket\ \sigma)\ f = c'$. Then Lemma 4 yields, that there exists a priority $i \leq \mathcal{P}(v)$ such that $I_i\ (\sigma\ f)\ c' = i$. Since $j = \mathcal{P}(s_q) > \mathcal{P}(v)$ we obtain $c' \in C_j$ concluding the proof. □

Note, that Property 2 ensures, that the interrupt-free tails exist. Otherwise higher priority interrupts changing f might be necessary to reach u with c''. For f-executable paths the sets of reaching values are also complete.

Theorem 4. *For a program point u reached by an interrupt with priority j, flag f and initial set C_0 we have:*

$$\mathbf{R}^\flat[u,j] \subseteq \bigcup_{\sigma\ f \in C_0} \bigcup_{\pi \in \Pi_f(\mathbf{R}[u],\sigma)} \{([\![\pi]\!]\ \sigma)\ f\}$$

Proof. In case $\mathbf{R}^\flat[u,j] = \emptyset$ the result follows immediately. Otherwise Lemma 2 yields, that for every $c' \in \mathbf{R}^\flat[u,j]$ there exists a $c'' \in C_j$ such that for a state σ_2 with $\sigma_2\ f = c''$ there exists a path $\pi_2 \in \Pi_f(\mathbf{R}[u],\sigma_2)$ with $([\![\pi_2]\!]\ \sigma_2)\ f = c'$ and $\pi_2 = (s_q, \_, \_)\pi_2'$ with $\mathcal{P}(s_q) = j$. If there exists a program point v with $\mathcal{P}(v) < j$ such that there exists state σ with $\sigma\ f \in C_0$ and a path $\pi_1 in \Pi_f(\mathbf{R}[v],\sigma)$ with $([\![\pi_1]\!]\ \sigma)\ f = c''$ then $\pi_1\pi_2 \in \Pi_f(\mathbf{R}[u],\sigma)$ concluding the proof. If however there is no such state or program point then by Lemma 5 we have $((\mathsf{lift}\ I_{i-1}) \circ \cdots \circ (\mathsf{lift}\ I_1) \circ (\mathsf{init}_0 C_0))\ c'' \geq j$. Therefore filter_j would remove c'' in contradiction with Lemma 2. □

Combining Theorems 3 and 4 shows, that if $\mathbf{R}^\flat[u,j] = \emptyset$ then there is no path which is f-executable for σ_0 and reaches u by an interrupt with priority j.

Corollary 1. *For a program point u, a priority j and a flag f with initial value set $C_0 = \{\sigma_0\ f\}$ we have:*

$$\mathbf{R}^\flat[u,j] = \emptyset \iff \Pi_f(\mathbf{R}[u],\sigma_0) = \emptyset$$

Consider again the program in Fig. 1. For (program points in) the critical sections Ix and Qx we have $\mathbf{R}^\flat[Ix,1]\ \{0\} = \{1\}$ and $\mathbf{R}^\flat[Qx,3]\ \{0\} = \{0\}$.

5.1 Data Race Analysis

In the previous section we have shown, that the values of a flag f can be tracked precisely along all f-executable paths reaching a given program point. A data race, however, consists of two program points u and v at which shared data is accessed and a path from one to the other. Therefore we extend the results for reaching values to start from any given priority level (instead of 1). The set of possible values of flag f at entry nodes of interrupts with priority j from a set of values C at priority i, is given by:

$$C_{i,j} = (\mathsf{filter}_j \circ I_{(i+1,j-1)} \circ \mathsf{init}_i)\ C$$

Accordingly, the set of all values of f possibly occurring at program point u when reached by an interrupt with priority j, is given by:

$$\mathbf{R}^\flat[u,i,j] = \bigcup \{\mathbf{R}^\sharp[u]\ c \mid c \in C_{(i,j)}\}$$

Consider a flag f with initial value set C_0 and program points u and v which are reached by interrupts with priorities j and k, respectively. To decide if there is a race at accesses to a variable x occurring at u and v we first compute by $\mathbf{R}^\flat[u, j] = C$ the values of f reaching u. Then starting with C and the priority $\mathcal{P}(u)$ of u we compute $\mathbf{R}^\flat[v, \mathcal{P}(u), k]$ the set of values of f reaching v is execution is interrupted immediately after the access at u. If this set is non-empty, there exists a path satisfying the conditions for a data race.

In contrast to Theorem 2 the construction and use of interrupt summaries allows us to determine the set of possible values reaching the start of a potentially racing interrupt and avoid the imprecision incurred by starting the analysis with V_f. Furthermore since intermediate changes of flags are tracked it is no longer necessary for flags to be *reliable*. Formally we have:

Theorem 5. *For accesses to a variable x at program points u and v where v is reached by an interrupt with priority $j > \mathcal{P}(u)$ and the set of values of flag f reaching u given by $C \neq \emptyset$ we have:*

(1) There is no race between the two accesses, if $\mathbf{R}^\flat[v, \mathcal{P}(u), j] = \emptyset$.
(2) If $\mathbf{R}^\flat[v, \mathcal{P}(u), j] \neq \emptyset$ there is a race path connecting u and v which is f-executable for some σ_0 with $\sigma_0\ f \in C_0$.

Proof (Theorem 5). If $\mathbf{R}^\flat[v, \mathcal{P}(u), j] = \emptyset$ we have $\mathbf{R}^\sharp[v]\ c = \emptyset$ for all $c \in C_{(\mathcal{P}(u),j)}$. This corresponds to $\mathbf{R}^\flat[v, j] = \emptyset$ in the (sub-)program consisting only of interrupts with priority greater than $\mathcal{P}(u)$. Therefore Corollary 1 yields, that there is no f-executable path reaching v from u with the given value sets. Consequently, there can be no race path.

If however, $\mathbf{R}^\flat[v, \mathcal{P}(u), j] \neq \emptyset$ there exists a path π from u to v which contains only interrupts of priority greater than $\mathcal{P}(u)$ and which is f-executable for initial states σ_1 with $\sigma_1\ f \in C$. Therefore every program point v_1 occurring in π_1 has a higher priority than u, i.e. $\mathcal{P}(v) > \mathcal{P}(u)$. Since $C \neq \emptyset$ there also exists an initial states σ_2 and a path π_2 reaching u that is f-executable for σ_2. With Corollary 1 we can assume, that $(\llbracket \pi_2 \rrbracket\ \sigma_2\ f = \sigma_1\ f$. Then $\pi_2\pi_1$ is a f-executable race path since the lowest priority obtained in π_1 is greater than $\mathcal{P}(u)$ and the states match. □

For our running example from Fig. 1 we have the initial value set $\{0\}$ and $\mathbf{R}^\flat[Ix, 1] = \{1\}$ as well as $\mathbf{R}^\flat[Qx, 1, 3] = \emptyset$ since the only value that can pass through the guard edge to Qx is 0. The analysis presented in this section is able to handle more complex patterns as well. For example consider the extension in Fig. 2. The assignment f=2; in the interrupt R adds 2 to the set of values Q can observe at Ix. We have $\mathbf{R}^\flat[I_1, 1] = \{1, 2\}$. But still $\mathbf{R}^\flat[Qx, 1, 3] = \emptyset$ certifying, that there is no race between Ix and Qx. If however Q checked for f != 1 instead, which yields the same result for the program without R, we would obtain $\mathbf{R}^\flat[Qx, 1, 3] = \{2\}$. The corresponding race path is given by first interrupting Ix with R and then after f has been set to 2 interrupting R with Q.

```
int f = 0;                    /* Priority 2 */          /* Priority 1 */
int x = 0;                    isr_R() {                 isr_I () {
/* Priority 3 */                if (f==1){                f = 1;
isr_Q() {                         f = 2;                  x++; /* Ix */
  if (f==0)                       f = 1;                  f = 0;
    x--; /* Qx */               }                       }
}                             }
```

Fig. 2. Extended example code of program with a flag f

Table 1. Benchmark results

program	loc	time (s)	warnings w/o flags	warnings w flags
`example_flag`	18	0,012	1	0
`resource_flag`	26	0,020	1	0
`inverse_flag`	21	0,012	1	1
`weak_flag`	21	0,016	1	1
`arbiter_flag`	26	0,018	1	0
`linecar`	2586	0,072	5	0
`bipedrobot`	2684	0,028	1	1
`ballsort`	2786	0,424	7	0
`controller`	∼400k	3171	929	265

6 Experimental Results

The analysis from Section 4 has been implemented in the Goblint tool [18]. For that, we extended the analysis with tracking of priorities and acquired resources as described in [15], as well as an inter-procedural treatment of local variables and parameters and global invariants to deal approximately with global data different from flag variables. We have tested our implementation on a number of valid and invalid uses of flags. The results are summarized in Table 1. Execution time (in seconds) on an Intel(R) Core(TM) i5 650 running Ubuntu is listed in column 3. The last two columns indicate the numbers of warnings by the analyzer without and with taking flag information into account.

Benchmark `example_flag` is the programs of Fig. 1 and the accesses to x are proven safe. Benchmark `inverse_flag` is benchmark `example_flag` with flipped priorities. While syntactically the flag pattern is intact the associated priorities render it moot. The analysis captures this and issues a data race warning for x. Benchmark `resource_flag` has an additional task of intermediate priority which temporarily (re)sets the flag to 0, but employs resources to increase the priority of I to the intermediate priority before using the flag. Due to the raised priority the value of f is reliable and the analysis verifies the accesses to x as safe. Benchmark `weak_flags` employs the flag pattern correctly, but not thoroughly. There is an additional high priority interrupt, which also accesses x. The analysis recognizes the flag pattern where it is used, but issue a data race warning due to the unprotected access. Benchmark `arbiter_flag` uses an extended synchronization scheme where the lowest priority interrupt uses the value of the flag variable to signal which higher-priority interrupt is allowed to enter its critical section.

Additionally, the analysis has been evaluated on four larger programs Benchmarks `linecar`, `bipedrobot`, and `ballsort` consist of about 300 to 500 lines of application code plus about 2300 lines of headers taken from the NxtOSEK framework [17]. While benchmark `bipedrobot` is taken from the NxtOSEK samples [17], benchmarks `linecar` and `ballsort` have been produced in students' projects. Interestingly, the students made use of flag variables without having been told to do so. Benchmark `linecar` is the control program for a line following car which picks up items placed on the track. Two variables are used to synchronize between scanning for the line and forward movement. Benchmark `bipedrobot` is a two-wheel self balancing robot which uses a resource, i.e. dynamic priorities, to synchronize between balancing and movement. The warning is due to an initialization task, which omits acquiring the resource. It assumes that the timers have not yet triggered the remaining parts of the program. The code of benchmark `ballsort` resembles a state machine controlling a manipulator arm which first locates colored balls in reach and then sorts them by placing red balls to the left and blue balls to the right. Additionally it has a pause button, which stops all movement until it is pressed again. A race-analysis based on priorities alone would warn on all shared variables. On the contrary, treating the state variable as a flag, allows to verify all accesses as safe.

Finally, `controller` is an industrial benchmark. It consists of about 400,000 lines of code including headers. During the analysis 7232 global variables are found. The flag analysis reduces the number of race warnings from 929 to 265. This corresponds to a reduction by over 60%. The analysis on this real-world example takes about 3171s.

Overall, the analysis times, in particular for the larger programs, are acceptable. Also the number of spurious data race warnings is dramatically reduced in the industrial benchmark `controller`. The remaining data race warnings could possibly be reduced further, if additional information such as, e.g., worst-case execution time and cycle duration of time-triggered tasks is taken into account.

7 Related Work

In general terms, we are attempting to exclude invalid paths from diluting the analysis result. Taking into account the control flow of procedure call and return, makes the analysis *context-sensitive*. Static analysis of concurrency errors requires context-sensitive alias analysis, and there are various solutions to obtain context-sensitive alias analysis for race detection, including approaches based on procedure summaries of relative locksets [19], bootstrapping of alias analyses [3], type based label-flow [12], and conditional must-not aliasing [10].

Distinguishing valid paths w.r.t. the sequential semantics of program execution is known as *path-sensitivity*. Path-sensitivity is required to deal, e.g., with conditional locking schemes. Voung et al. [19] identify this as one important source of false alarms in their analyzer. A generic approach to path-sensitivity is *property simulation* [2]. We have previously applied this technique to achieve path-sensitive data race analysis [18]. What we require here, however, is path-sensitivity w.r.t. the interleaving semantics of the program. We need to exclude semantically

invalid interleavings, not merely invalid paths in the flow-graphs. None of the static race detection tools we are aware of [3, 9, 13, 19] attempt to exclude semantically invalid interleavings based on boolean guards. Also, our tests with state-of-the-art dynamic race detection tool, Intel Inspector XE 2013 (formerly Intel Thread Checker [14]), revealed that boolean flags confuse the analysis such that no warnings are generated at all even when negating the flag to make it invalid.

We realized the seriousness of this issue when attempted to adapt conventional race detection techniques to priority-driven single-processor concurrency systems [15]. In this setting, locking can be avoided by higher priority threads because lower-priority threads cannot preempt them. We have explored how boolean program variables can be used instead of explicit locking. Another way to avoid acquiring a resources by higher priority threads is to merely check if the lock is free. If no other thread holds the lock, the highest priority thread can simply execute the critical section. Miné [7] presents a scheduled interference semantics of ARINC 653 systems that takes this into account by tracking a separate set of resources known to be free. A mutex is only known to be free when it has been explicitly probed by the highest priority thread that uses it and there has been no blocking operations (e.g., attempting to lock another mutex) since the mutex was last probed.

8 Conclusion

Application developers tend to rely not only on the synchronization mechanisms provided by embedded operating systems such as Autosar/OSEK to secure their programs. Therefore, we provided analysis methods for programs executed by Autosar/OSEK which can deal with hand-crafted synchronization mechanisms based on flag variables.

The analysis is based on an off-the-shelf constant propagation analysis, together with post-processing to take care of the intricate inter-interrupt dependencies arising from the priority based scheduling of OSEK programs. Our characterization of flags allowed us to precisely determine whether two accesses comprise a data race or not. The required fix-point computations are all linear in the size of the program and polynomial in the number of possible flag values, which in our experience is small.

The construction can be enhanced by using more sophisticated value analyses of flag variables, e.g., by allowing to store and restore flag values or by tracking dependences between different flags. Our preliminary experiments, however, showed that already the simple version of our analysis presented in Section 4 drastically reduces the number of false alarms in real-world programs. It remains for future work to evaluate in how far stronger analyses will have a significant impact on the practical precision of the analysis.

Acknowledgements. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement № 269335 (MBAT) and from the German Research Foundation (DFG). The third author was supported by the research theme IUT2-1 and ERDF funded Estonian ICT national programme project “Coinduction”.

References

[1] Autosar consortium: Autosar Architecture Specification, Release 4.0 (2009), http://www.autosar.org/

[2] Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI 2002, pp. 57–68. ACM Press (2002)

[3] Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 226–239. Springer, Heidelberg (2007)

[4] Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all — reducing concurrent analysis to sequential analysis under priority scheduling. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010)

[5] Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206. ACM Press (1973)

[6] Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 125–140. Springer, Heidelberg (1992)

[7] Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 398–418. Springer, Heidelberg (2011)

[8] Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL 2004, pp. 330–341. ACM Press (2004)

[9] Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI 2006, pp. 308–319. ACM Press (2006)

[10] Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL 2007, pp. 327–338. ACM Press (2007)

[11] OSEK/VDX Group: OSEK/VDX Operating System Specification, Version 2.2.3 (2005), http://www.osek-vdx.org

[12] Pratikakis, P., Foster, J.S., Hicks, M.W.: Existential label flow inference via CFL reachability. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 88–106. Springer, Heidelberg (2006)

[13] Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for detecting races. In: PLDI 2006, pp. 320–331. ACM Press (2006)

[14] Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: ASID 2006, pp. 34–41. ACM Press (2006)

[15] Schwarz, M.D., Seidl, H., Vojdani, V., Lammich, P., Müller-Olm, M.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: POPL 2011. ACM Press (2011)

[16] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)

[17] Chikamasa, T., et al.: OSEK platform for lego® mindstorms® (2010), http://lejos-osek.sourceforge.net/

[18] Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)

[19] Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: ESEC/FSE 2007, pp. 205–214. ACM Press (2007)

[20] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 181–210 (1991)