Static Analysis of Interrupt-driven Programs Synchronized via the Priority Ceiling Protocol

Conference Paper *in* ACM SIGPLAN Notices · January 2011

DOI: 10.1145/1925844.1926398 · Source: DBLP

5 authors, including:

Martin D. Schwarz
Technische Universität München
3 PUBLICATIONS **37** CITATIONS
SEE PROFILE

Helmut Seidl
Technische Universität München
249 PUBLICATIONS **4,528** CITATIONS
SEE PROFILE

Peter Lammich
Technische Universität München
46 PUBLICATIONS **596** CITATIONS
SEE PROFILE

Markus Muller-olm
University of Münster
91 PUBLICATIONS **1,460** CITATIONS
SEE PROFILE

Static Analysis of Interrupt-driven Programs Synchronized via the Priority Ceiling Protocol

Martin D. Schwarz Helmut Seidl
Vesal Vojdani
Technische Universität München
Boltzmannstraße 3, D-85748 Garching, Germany
{schwmart,seidl,vojdanig}@in.tum.de

Peter Lammich Markus Müller-Olm
Westfälische Wilhelms-Universität Münster
Einsteinstraße 62, D-48149 Münster, Germany
{peter.lammich,markus.mueller-olm}@uni-muenster.de

Abstract

We consider programs for embedded real-time systems which use priority-driven preemptive scheduling with task priorities adjusted dynamically according to the immediate ceiling priority protocol. For these programs, we provide static analyses for detecting data races between tasks running at different priorities as well as methods to guarantee transactional execution of procedures. Beyond that, we demonstrate how general techniques for value analyses can be adapted to this setting by developing a precise analysis of affine equalities.

Categories and Subject Descriptors F.3.1 [*Logics and meaning of programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and meaning of programs*]: Semantics of Programming Languages—Program analysis; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Algorithms, Theory, Verification

Keywords inter-procedural analysis, abstract domains, interrupt-driven concurrency

1. Introduction

There is an inherent tension in concurrent real-time software between synchronization, needed to preserve data consistency, and prioritized execution, needed to meet hard deadlines. Retaining execution of high-priority tasks within a predictable time frame, yet allowing lower-priority tasks to complete critical sections, requires sophisticated synchronization primitives which limit the worst-case waiting time of high-priority tasks. Using common binary semaphores (mutexes) can result in a higher-priority task waiting an indefinite amount of time for a lower-priority task to complete, a situation known as *unbounded priority inversion*.

As an example of unbounded priority inversion, consider a low-priority task q_1 which acquires a lock needed by a high-priority task q_3. When q_3 is ready to execute, it will preempt q_1, but as soon as q_3 attempts to acquire the lock, it must wait for q_1 to complete its critical section. It is perfectly acceptable, and necessary for the sake of data consistency, that a high-priority task wait for a low-priority task to complete execution of a critical section; however, the problem is that an intermediate-priority task q_2 may preempt the low-priority task before it releases the lock. Then, q_2 is indirectly blocking the high-priority task q_3 for an arbitrary amount of time.

Since unbounded priority inversion defeats the possibility of meeting hard deadlines, operating systems for embedded systems provide more sophisticated synchronization primitives than common mutexes. Typically such primitives are based on priority inheritance: a lower-priority task which blocks a higher-priority task inherits the priority of that higher-priority task for the duration of the critical section which caused the blocking. This bounds the time a higher-priority task can be blocked. The *original ceiling priority protocol* [27] ensures that a higher-priority task can only be blocked for the duration of a single critical section.

In safety-critical systems, a simplification of this protocol, *the immediate ceiling priority protocol*, is often used. This variation is also known as the priority ceiling emulation protocol to distinguish it from the original inheritance-based ceiling protocol. Under this name, it is included in Safety Critical Java [13], a proposed subset of Real-Time Java. The immediate ceiling priority protocol is used by the OSEK/VDX operating system [22], which has been adopted by the automotive software architecture Autosar [3], an emerging global standard in the automotive domain. It is also present in the POSIX library, where it is called the Priority Protect Protocol. In this paper, we will follow the OSEK usage and simply call it the priority ceiling protocol (PCP). The PCP relies on the concept of resources. Each resource r obtains a ceiling priority, which is the maximal priority of tasks acquiring the resource r. The scheduling of tasks then follows the *dynamic priority* of tasks, i.e., the maximum of a task's static priority and the ceiling priorities of all resources it has acquired. In this way, a task acquiring a given resource will immediately inherit the priority of all tasks which could request that resource. As this blocks intermediate-priority tasks, unbounded priority inversion is avoided.

In this paper, we develop static analyses for programs synchronized via the PCP. We provide methods for uncovering subtle flaws due to the concurrency induced by interrupts. Specifically, we focus on data races and transactional behavior of procedures. Moreover, we explain how interprocedural value analyses can be enhanced to take priorities and interrupts into account. We exemplify this with an algorithm for inferring affine equalities. The PCP was engineered to run on uniprocessor systems, which are the *de facto* standard for embedded real-time systems.

The program in Figure 1 is used as a running example throughout the paper. It consists of one (main) task T and two interrupts I and I' with priorities 1, 2 and 3, respectively (higher numbers denote higher priorities). The program uses resources r and r'. Since

POPL'11 January 26–28, 2011, Austin, Texas, USA.

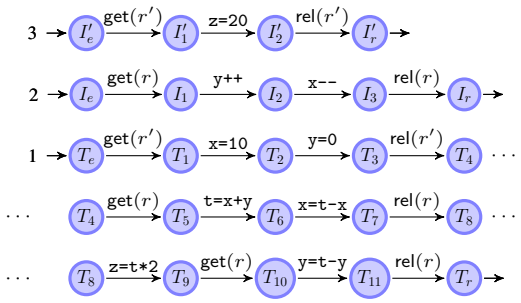

Figure 1. Example program.

r is used by T and I, its ceiling priority is 2, while r' is used by T and I', so its ceiling priority is 3. The interrupt I' of highest priority resets the variable z to the fixed value 20. The interrupt I increments the counter y and decrements the counter x. The main task T initializes the counters x and y; then, it attempts to swap their values by means of an auxiliary variable t, which receives the sum of x and y. Before completing the swap, the result $2 * t$ is stored in the variable z. This example is designed to exhibit both a data race and a nontransactional behavior.

The resource r' is held during the initialization of the main task; consequently, the dynamic priority of this part is 3 (the maximal priority), ensuring that no interrupts may occur. Then, the resource r with ceiling priority 2 is acquired, which will only protect against the interrupt I, while I' may still occur. Note that the interrupt I is unaware of the resource r', and yet acquiring resource r', protects a task from being interrupted by I because the static priority of I is less than the ceiling priority of r'. This effect is typical for priority based synchronization. An analysis which treats resources as locks, though, could not exclude the possibility of I interrupting the initialization code, resulting in false alarms.

The assignment $\mathtt{z} = \mathtt{t} * 2$ may overwrite the assignment in I' if I' occurs at T_8 (or earlier). Or it might itself be overwritten if I' occurs at T_9 (or later). This constitutes a *data race*. Moreover, at T_{10}, variable y is overwritten with a value that might have become outdated due to an occurrence of I. In the example, this will result in failing to correctly swap the variables x and y. Note that this occurs although all accesses to x and y are protected by the resource r. We call this *nontransactional* behavior.

The paper is organized as follows. In Section 2, we present a concrete semantics of PCP programs based on execution paths. We formally define the concepts of nontransactional behavior and data races for interrupt-driven programs. In Section 3, we show how to analyze resource sets to determine priorities and consequently possible interleavings. In Section 4, we use the resource framework to detect races, and in Section 5, nontransactional behavior. Section 6, shows how to extend the framework to allow data-flow analysis and we exemplify this by computing affine equations for the example program. Section 7 presents the experimental evaluation of our methods for race detection and transactionality. Finally, we discuss related work and conclude.

2. The Core PCP Model and its Semantics

Our model consists of one task $main$, with which program execution starts; a finite collection of interrupt routines Irpt, which we also call tasks, i.e., $\mathsf{Task} = \mathsf{Irpt} \cup \{main\}$; and auxiliary procedures Proc, which may be called by all tasks. The main task as well as the interrupt routines are distinguished procedures which may not be called otherwise. For the sake of the analysis, procedures are specified by means of control-flow graphs as in Figure 1.

Every procedure f has a designated entry node f_e and return node f_r. The collection of all control flow graphs of procedures in Proc is denoted by (N, E) where N is the set of nodes and E the set of edges. Let N_e and N_r denote the set of all entry and return nodes, respectively. Each edge is labeled either with a basic statement s, or with a call $f()$, $f \in \mathsf{Proc}$. For simplicity, we consider procedures without parameters only. Each basic statement s is either a basic command cmd, such as an assignment to a global variable, or a PCP statement, such as resource acquisition. Let Res denote the finite set of resources used by the program. For $r \in \mathsf{Res}$, the PCP statement $\mathsf{get}(r)$ acquires the resource r and $\mathsf{rel}(r)$ releases r. We assume that at the exit node of each task, all resources have been released. This can be enforced, e.g., by successively releasing all resources potentially used by the task.

Additionally, we assume that we are given functions $\mathcal{P} : \mathsf{Task} \to \mathbb{N}$ and $\mathcal{U} : \mathsf{Task} \to 2^{\mathsf{Res}}$ which map tasks to their *static* priorities from $\mathbb{N}$ and (super-)sets of the sets of resources possibly acquired during their execution. In particular, we assume that $\mathcal{P}(main) = 0 < \mathcal{P}(q)$ for each $q \in \mathsf{Irpt}$. To each resource $r \in \mathsf{Res}$, we then assign its *ceiling priority* $\mathcal{P}(r)$ which equals the maximal priority of a task acquiring r, i.e.,

$$\mathcal{P}(r) = \max\{\mathcal{P}(q) \mid r \in \mathcal{U}(q)\}$$

For a subset of resources $R \subseteq \mathsf{Res}$, we also write $\mathcal{P}(R)$ as a shorthand for $\max\{\mathcal{P}(r) \mid r \in R\}$ where $\mathcal{P}(\emptyset) = -\infty$.

2.1 Execution Paths

An execution path π of a PCP program is a sequence of control-flow edges labeled by basic statements into which subsequences corresponding to procedure calls or interrupts are nested. The nesting of a call to the procedure f is indicated by means of the start and end tags $\langle f \rangle$ and $\langle / f \rangle$, respectively. Interrupts are indicated analogously.

Same-level execution paths reaching a program point v (on the same level) are defined as follows:

- ϵ is an same-level execution path reaching the entry nodes q_e of procedures;
- $\pi(u, \mathsf{cmd}, v)$ is a same-level execution path reaching v if π is a same-level execution path reaching u;
- $\pi_1 \langle f \rangle \pi_2 \langle / f \rangle$ is a same-level execution path reaching v if π_1 is a same-level execution path reaching u, $(u, f(), v)$ is a call edge, and π_2 is a same-level execution path reaching the return node f_r of f;
- $\pi_1 \langle q \rangle \pi_2 \langle / q \rangle$ is a same-level execution path reaching v if π_1 is a same-level execution path reaching v, and π_2 is a same-level execution path reaching the return node q_r of an interrupt q.

Likewise, a (reaching) execution path reaching a program point v (not necessarily at the same level) is defined as follows.

- ϵ is an execution path reaching the entry point $main_e$ of the main task;
- $\bar{\pi}(u, \mathsf{cmd}, v)$ is an execution path reaching v if $\bar{\pi}$ is an execution path reaching u;
- $\bar{\pi}_1 \langle f \rangle \pi_2 \langle / f \rangle$ is an execution paths reaching v if $\bar{\pi}_1$ is a execution path reaching u, $(u, f(), v)$ is a call edge and π_2 is an execution path reaching the return node f_r of f;
- $\bar{\pi}_1 \langle q \rangle \pi_2 \langle / q \rangle$ is an execution path reaching v if $\bar{\pi}_1$ is an execution path reaching v, and π_2 is a same-level execution path reaching the return node q_r of an interrupt q;
- $\bar{\pi} \langle q \rangle$ is an execution path reaching the entry node q_e of q if $\bar{\pi}$ is an execution path reaching u and either there is a call edge $(u, q(), v)$ or q is an interrupt.

We write Π_S and Π_R for the set of same-level execution paths and reaching execution paths, respectively.

For same-level execution paths, we define the resource set after the path in terms of the resource set R before the path:

$$\begin{array}{lcll}
\mathcal{R}(\epsilon, R) & = & R & \\
\mathcal{R}(\pi(u, \mathsf{cmd}, v), R) & = & \mathcal{R}(\pi, R) & \\
\mathcal{R}(\pi(u, \mathsf{get}(r), v), R) & = & \mathcal{R}(\pi, R) \cup \{r\} & \\
\mathcal{R}(\pi(u, \mathsf{rel}(r), v), R) & = & \mathcal{R}(\pi, R) \setminus \{r\} & \\
\mathcal{R}(\pi_1\langle q\rangle\pi_2\langle/q\rangle, R) & = & \mathcal{R}(\pi_1, R) & \text{if } q \in \mathsf{Task} \\
\mathcal{R}(\pi_1\langle f\rangle\pi_2\langle/f\rangle, R) & = & \mathcal{R}(\pi_2, \mathcal{R}(\pi_1, R)) & \text{if } f \notin \mathsf{Task}
\end{array}$$

This definition is extended to (reaching) execution paths:

$$\begin{array}{lcll}
\mathcal{R}(\bar\pi\langle f\rangle\pi, R) & = & \mathcal{R}(\pi, \mathcal{R}(\bar\pi, R)) & \text{if } f \notin \mathsf{Task} \\
\mathcal{R}(\bar\pi\langle q\rangle\pi, R) & = & \mathcal{R}(\pi, \emptyset) & \text{if } q \in \mathsf{Task}
\end{array}$$

where π is a same-level execution path and $\bar\pi$ is a reaching execution path.

2.2 The Path Semantics

The concrete semantics collects, for each program point v, the set of execution paths reaching v taking static priorities and resource sets into account. Each task starts with an empty set of resources. We assume that procedures cannot be interrupted at their entry points. If a task is interrupted (while executing some procedure), its resource set before and after the interrupt is identical, while procedure calls may change the set of currently held resources.

Let p denote the maximal static priority of all tasks, and $[p]$ the interval $\{0, \ldots, p\}$. For a procedure f, let $[\![f]\!] : [p] \to (2^{\mathsf{Res}} \to 2^{\Pi_S})$ denote the function which assigns to each static priority i at which f can be called, a function which takes a resource set R before a call to f and returns the set of same-level execution paths of the procedure f including all execution paths of interrupts which possibly may have occurred. Likewise for $j > 0$, let $[\![j]\!] \subseteq \Pi_S$ denote the set of execution paths of all interrupts with static priority level j. For an edge e labeled with a basic statement, the concrete semantics is a function $[\![e]\!] : 2^{\mathsf{Res}} \to 2^{\Pi_S}$ which for each resource set R returns the singleton set $[\![e]\!](R) = \{e\}$.

In order to put up a constraint system to characterize the sets of same-level execution paths of procedures, we require composition operators which take resource sets and static priorities into account. The composition of mappings $M_1, M_2 : 2^{\mathsf{Res}} \to 2^{\Pi_S}$ must ensure that the sets of execution paths are concatenated according the attained sets of resources:

$$(M_2 \circ M_1)\ R = \{\pi_1\pi_2 \mid \pi_1 \in M_1(R), \pi_2 \in M_2(\mathcal{R}(\pi_1, R))\}$$

For interrupts, we need to filter out execution paths which cannot be interrupted due to high dynamic priorities. Let S_j denote the set of execution paths of interrupts of priority j. Then we define the application of S_j with a mapping $M : 2^{\mathsf{Res}} \to 2^{\Pi_S}$:

$$(S_j \bullet_j M)\ R = \{\pi_1\pi_2 \mid \pi_1 \in M(R), \pi_2 \in S_j, j > \mathcal{P}(\mathcal{R}(\pi_1, R))\}$$

where the priority condition checks that the acquired resources allow the interrupt to occur.

The functions $[\![f]\!] : [p] \to 2^{\mathsf{Res}} \to 2^{\Pi_S}$, $f \in \mathsf{Proc}$, and the sets $[\![j]\!]$ then can be characterized as the least solution of the following constraint system:

$$\begin{array}{lllll}
[\mathrm{S0}] & \mathbf{S}[f_e, i] & \supseteq I & & \\
[\mathrm{S1}] & \mathbf{S}[v, i] & \supseteq [\![(u, s, v)]\!] \circ \mathbf{S}[u, i] & (u, s, v) \in E & \\
[\mathrm{S2}] & \mathbf{S}[v, i] & \supseteq ([\![f]\!]\, i) \circ \mathbf{S}[u, i] & (u, f(), v) \in E & \\
 & [\![f]\!]\, i & \supseteq \mathcal{H}_f(\mathbf{S}[f_r, i]) & & \\
[\mathrm{S3}] & [\![j]\!] & \supseteq ([\![q]\!]\ j\ \emptyset) & q \in \mathsf{Irpt}, \mathcal{P}(q) = j & \\
 & \mathbf{S}[u, i] & \supseteq [\![j]\!] \bullet_j \mathbf{S}[u, i] & u \notin N_e, j > i &
\end{array}$$

Here, the function I is given by $I(R) = \{\epsilon\}$. The auxiliary variable $\mathbf{S}[v, i]$ for a node v of some procedure f, and a static priority i describes the function which for a given resource set R at procedure start, returns the set of all same-level execution paths reaching v within f when executed within a task of static priority i. The auxiliary operator $\mathcal{H}_f$ takes a description M of the same-level execution paths of the procedure f and wraps it into the opening and closing tags corresponding to f, i.e,

$$\mathcal{H}_f(M)(R) = \{\langle f\rangle\pi\langle/f\rangle \mid \pi \in M(R)\}$$

For real-time systems, it is reasonable to assume that every procedure has at least one same-level execution path, i.e., that $[\![f]\!]\, i\, R$ is non-empty for every i and R. If this is the case, the set of all program points which are definitely unreachable can be computed by standard means and then removed from the control-flow graphs — implying that property (S) is satisfied:

(S) Each program point v of a procedure f is same-level reachable, i.e., $S[v, i](R) \neq \emptyset$ for every i and R.

This property therefore will henceforth be generally assumed.

In order to put up a constraint system to characterize the sets of reaching execution paths, we require an operator to apply the effects $M : 2^{\mathsf{Res}} \to 2^{\Pi_S}$ of edges or procedures to sets of reaching execution paths S, which takes attained sets of resources into account. We define

$$M@S = \{\bar\pi_1\pi_2 \mid \bar\pi_1 \in S, \pi_2 \in M(\mathcal{R}(\bar\pi_1, \emptyset))\}$$

Likewise, the application of a set of same-level execution paths S_2 of interrupts of priority j to a set S_1 of reaching execution paths is defined by:

$$S_2 @_j S_1 = \{\bar\pi_1\pi_2 \mid \bar\pi_1 \in S_1, \pi_2 \in S_2, \mathcal{P}(\mathcal{R}(\bar\pi_1, \emptyset)) < j\}$$

The collecting semantics of sets of reaching execution paths then is given by the least solution of the following constraint system:

$$\begin{array}{lllll}
[\mathrm{R0}] & \mathbf{R}[main_e, 0] & \supseteq \{\epsilon\} & & \\
[\mathrm{R1}] & \mathbf{R}[v, i] & \supseteq [\![(u, s, v)]\!]@\mathbf{R}[u, i] & (u, s, v) \in E & \\
[\mathrm{R2}] & \mathbf{R}[v, i] & \supseteq ([\![f]\!]\, i)@\mathbf{R}[u, i] & (u, f(), v) \in E & \\
 & \mathbf{R}[f_e, i] & \supseteq \mathsf{enter}_f(\mathbf{R}[u, i]) & (u, f(), v) \in E & \\
[\mathrm{R3}] & \mathbf{R}[u, i] & \supseteq [\![j]\!]@_j\mathbf{R}[u, i] & u \notin N_e, j > i & \\
 & \mathbf{R}[j] & \supseteq \mathsf{proj}_j(\mathbf{R}[u, i]) & u \notin N_e, j > i & \\
 & \mathbf{R}[q_e, \mathcal{P}(q)] & \supseteq \mathsf{enter}_q(\mathbf{R}[j]) & q \in \mathsf{Irpt}, \mathcal{P}(q) = j &
\end{array}$$

Here, the operator enter_f for a procedure $f \notin \mathsf{Task}$ when applied to a set S of reaching execution paths, appends the opening tag $\langle f\rangle$ to each reaching execution path in S, i.e.,

$$\mathsf{enter}_f(S) = \{\bar\pi\langle f\rangle \mid \bar\pi \in S\}$$

The operator proj_j when applied to a set S, extracts the set of all reaching execution paths $\bar\pi$ from S where the priority of the resource set $\mathcal{R}(\bar\pi, \emptyset)$ is less than j, i.e,

$$\mathsf{proj}_j(S) = \{\bar\pi \mid \bar\pi \in S, \mathcal{P}(\mathcal{R}(\bar\pi, \emptyset)) < j\}$$

The constraints $S0$, $R0$ provide values for entry points of all procedures in Proc, and the entry node of procedure $main$, respectively. $S1$ and $R1$ take care of all non-call edges, by applying the semantics of the edge, i.e., appending the edge to the collected sets of paths. Procedure calls are handled by the constraints $S2$ and $R2$. In $S2$ as well as the first part of $R2$, the same-level executions of the called procedure are composed with the executions before the call. Additionally, the second part of $R2$ describes execution paths entering procedures. The constraints $S3$ and $R3$ deal with interrupts. They correspond to implicitly introducing extra loop edges with *interrupt calls* at every node where an interrupt may occur, given that the dynamic priority of the executing task is sufficiently

low. Interrupts are summarized by static priority levels. Similarly the entry sets of the reaching execution paths are summarized into one set for each static priority level. According to our assumption, entry nodes are excluded from constraints $S3$ and $R3$.

2.3 Data Races and Nontransactional Behavior

Let $Acc(\mathsf{cmd})$ and $Acc(f)$ denote the set of all variables accessed by the basic command cmd and same-level executions of procedure f, respectively. For clarity of presentation, we do not distinguish read and write accesses.

Then a *data race* at some global variable x occurs if program execution reaches an access to x at a dynamic priority j while x also might be accessed by some interrupt q of static priority exceeding j.

Definition 1. A PCP program contains a data race at variable x if there exists a reaching execution path

$$\bar{\pi}\ (u, \mathsf{cmd}, v)\ \langle q\rangle\ \pi\langle/q\rangle \quad \in \quad \mathbf{R}[v, i]$$

for a basic command cmd with $x \in Acc(\mathsf{cmd})$ and a same-level execution path π of the interrupt q with $x \in Acc(q)$.

In the example program of Figure 1, a data race occurs at the variable z. The interrupt I' has static priority 3 and accesses z while the main task T accesses z at a dynamic priority of 1 and can therefore be preempted by I' at T_8 or T_9. A possible run of the example program reaching the data race would be: $(T_e, \mathsf{get}(r'), T_1)\ \ldots\ (T_3, \mathsf{rel}(r'), T_4)\ \ldots\ (T_8, \mathtt{z} = \mathtt{t} * 2, T_9)$ $\langle I'\rangle (I'_e, \mathsf{get}(r'), I'_1)\ (I'_1, \mathtt{z} = 20, I'_2)\ (I'_2, \mathsf{rel}(r'), I'_r)\langle/I'\rangle$. There are no further data races in the example program since the variables x and y are only used by the interrupt I which has a static priority of 2 and all accesses to these variables occur at a dynamic priority of at least 2.

A procedure f is considered as *transactional* (or atomic) if during every execution of f, no interrupt may occur between the first and last access to global variables which accesses any of the globals accessed by f.

Definition 2. Formally, f is *nontransactional* at static priority j for a resource set R, if there exists a same-level execution path

$$\langle f\rangle\ \pi_1\ \langle q\rangle\ \pi\ \langle/q\rangle\ \pi_2\langle/f\rangle \quad \in \quad [\![f]\!](j)(R)$$

where the following holds:

- $\pi_1\pi_2 \in \mathbf{S}[f_r, j](R)$ is a same-level execution path which contains no interrupts;
- π is a same-level execution path of the interrupt q without further interrupts;
- π_1 and π_2 contain edges $(u_1, \mathsf{cmd}_1, v_1)$,$(u_2, \mathsf{cmd}_2, v_2)$ with $Acc(\mathsf{cmd}_1) \neq \emptyset \neq Acc(\mathsf{cmd}_2)$; and
- π contains an edge (u, cmd, v) with $Acc(\mathsf{cmd}) \cap Acc(f) \neq \emptyset$.

To exemplify this situation we use again the program of Figure 1. The first access of the main task T to a global occurs before program point T_2 while the last access is after T_{10}. In between, e.g., at program point T_8, the dynamic priority is 1. At this node, T can be preempted by the interrupt I which changes the variables x and y also used by T. Therefore T is *not* transactional. A possible run of the example program exhibiting this behavior would be: $\langle T\rangle\ (T_e, \mathsf{get}(r'), T_1)\ \ldots\ (T_7, \mathsf{rel}(r), T_8)\ \langle I\rangle\ (I_e, \mathsf{get}(r), I_1)\ \ldots$ $(I_3, \mathsf{rel}(r), I_r)\ \langle/I\rangle\ (T_8, \mathtt{z} = \mathtt{t} * 2, T_9) \ldots (T_{11}, \mathsf{rel}(r), T_r)\ \langle/T\rangle$.

3. Analyzing Resources

In this section, we present an analysis of sets of resources possibly held at a given program point. The results of this analysis are fundamental for determining the minimal dynamic priority guaranteed to hold at this program point, as well as all subsequent analyses of the program.

The analysis determines for each program point a set of possible resource sets. The complete lattice thus is given by $2^{2^{\mathsf{Res}}}$. Sets of sets are ordered by subset inclusion. At join points we therefore take the union of reaching sets.

For analyzing same-level executions, we associate to each procedure f an abstract semantics $[\![f]\!]^\sharp : 2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$. Unlike the collecting semantics, $[\![f]\!]^\sharp$ does not depend on the static priority in which a call to f is made. The static priority determines the interrupts which may occur during the call of f, but interrupts do not modify the sets of currently held resources. In order to set up the corresponding abstract constraint system, we define an abstract semantics from $2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$ for edges (u, s, v) labeled with basic statements. We define:

$$\begin{array}{lcl} [\![(u, \mathsf{cmd}, v)]\!]^\sharp(R) & = & \{R\} \\ [\![(u, \mathsf{get}(r), v)]\!]^\sharp(R) & = & \{R \cup \{r\}\} \\ [\![(u, \mathsf{rel}(r), v)]\!]^\sharp(R) & = & \{R \setminus \{r\}\} \end{array}$$

Additionally, we require an abstract composition operator $\circ^\sharp$ which, for abstract mappings $M_1, M_2 : 2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$, returns the function defined by the following equation:

$$(M_2 \circ^\sharp M_1)(R) = \bigcup\{M_2(R') \mid R' \in M_1(R)\}$$

As with the concrete semantics we compute the effect of procedures depending on resource sets they are called with. The functions $[\![f]\!]^\sharp : 2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$, $f \in \mathsf{Proc}$, then are given by the least solution of the following constraint system:

$$\begin{array}{lll} [\mathrm{S}^\sharp 0] & \mathbf{S}[f_e]^\sharp \sqsupseteq I^\sharp & f \in \mathsf{Proc} \\ [\mathrm{S}^\sharp 1] & \mathbf{S}[v]^\sharp \sqsupseteq [\![(u, s, v)]\!]^\sharp \circ^\sharp \mathbf{S}[u]^\sharp & (u, s, v) \in E \\ [\mathrm{S}^\sharp 2] & \mathbf{S}[v]^\sharp \sqsupseteq [\![f]\!]^\sharp \circ^\sharp \mathbf{S}[u]^\sharp & (u, f(), v) \in E \\ [\mathrm{S}^\sharp 3] & [\![f]\!]^\sharp \sqsupseteq \mathbf{S}[f_r]^\sharp & \end{array}$$

where the function $I^\sharp$ is given by $I^\sharp(R) = \{R\}$. No constraints have been included to deal with interrupts: the reason is that interrupts do not change sets of held resources.

For determining the sets of reaching resource sets, we require an abstract application operator $@^\sharp$ which applies a mapping $M : 2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$ to a set of resource sets $S \in 2^{2^{\mathsf{Res}}}$ as follows:

$$M@^\sharp S = \bigcup\{M(R) \mid R \in S\}$$

For approximating the sets of resource sets reaching a node v at static priority level i, we consider the following constraint system:

$$\begin{array}{lll} [\mathrm{R}^\sharp 0] & \mathbf{R}[q_e, j]^\sharp \supseteq \{\emptyset\} & q \in \mathsf{Task}, j = \mathcal{P}(q) \\ [\mathrm{R}^\sharp 1] & \mathbf{R}[v, j]^\sharp \supseteq [\![(u, s, v)]\!]^\sharp @^\sharp \mathbf{R}[u, j]^\sharp & (u, s, v) \in E \\ [\mathrm{R}^\sharp 2] & \mathbf{R}[v, j]^\sharp \supseteq [\![f]\!]^\sharp @^\sharp \mathbf{R}[u, j]^\sharp & (u, f(), v) \in E \\ & \mathbf{R}[f_e, i]^\sharp \supseteq \mathbf{R}[u, i]^\sharp & (u, f(), v) \in E \end{array}$$

Note that the constraints $R^\sharp 0$ not only provides an abstract start value for the entry node of $main$, but also for the entry nodes of all interrupts $q \in \mathsf{Irpt}$. This is because every interrupt may occur, for example at the exit node of $main$, and will always start with the empty resource set. Note further that for reachability, we have kept the information i about the static priority at which a node is reached, in order to be able to determine its possible dynamic priorities.

In order to relate the concrete and abstract semantics of programs, we introduce appropriate abstraction functions $\alpha : (2^{\mathsf{Res}} \to 2^{\Pi_S}) \to (2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}})$ and $\bar{\alpha} : 2^{\Pi_R} \to 2^{2^{\mathsf{Res}}}$. These are given by

$$\begin{array}{lcl} \alpha(M)(R) & = & \{\mathcal{R}(\pi, R) \mid \pi \in M(R)\} \\ \bar{\alpha}(X) & = & \{\mathcal{R}(\bar{\pi}, \emptyset) \mid \bar{\pi} \in X\} \end{array}$$

The following theorem states that the resource sets computed by the above constraint systems not only safely over-approximate the set of resources attained by the collecting semantics, but the computations are precise w.r.t. the concrete semantics.

Theorem 1. *1. Let $[\![f]\!]\ j, \mathbf{S}[v,j]$ and $[\![f]\!]^\sharp, \mathbf{S}[v]^\sharp$ denote the least solutions of the constraint systems S and $S^\sharp$, respectively. Then for every procedure f, static priority i, and program point v,*

$$\alpha([\![f]\!]\ j) = [\![f]\!]^\sharp \qquad \alpha(\mathbf{S}[v,j]) = \mathbf{S}[v]^\sharp$$

2. Let $\mathbf{R}[v,j]$ and $\mathbf{R}[v,j]^\sharp$ denote the least solutions of the constraint systems R and $R^\sharp$, respectively. Then for every program point v and possible static priority j,

$$\bar{\alpha}(\mathbf{R}[v,j]) = \mathbf{R}[v,j]^\sharp$$

Proof. For the proof, we observe that for every edge $e = (u,s,v)$,

$$\alpha([\![e]\!]) = [\![e]\!]^\sharp$$

Also for composition of functions $M_1, M_2 : 2^{\mathsf{Res}} \to 2^{\Pi_S}$, we have:

$$\alpha(M_2 \circ M_1) = \alpha(M_2) \circ^\sharp \alpha(M_1)$$

Since furthermore, the abstract functions $[\![e]\!]^\sharp$, as well as the operation $\circ^\sharp$ are completely distributive, i.e., commute with arbitrary least upper bounds, the first assertion of the theorem follows by fixpoint induction. A similar argument applies to the second statement of the theorem. □

Let n denote the sum of the number of program points and control flow edges, and o the number of resources used by the program. Recall that p is the number of static priority levels. Then the number of constraints in the systems $S^\sharp$ and $R^\sharp$ are bounded by $\mathcal{O}(p \cdot n)$. Since the height of the lattice $2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}$ is exponential in the number of resources, the least solution of these systems can be computed by a standard work-list algorithm in time $\mathcal{O}(p \cdot n \cdot 2^{co})$ for some constant c.

Typically, the number of resources used by embedded controllers are quite small. A practical implementation still may tabulate functions $[\![f]\!]^\sharp$ only for those (i,R) of static task priorities i and resource sets R for which the procedure f may be called at runtime. Also, elements in $2^{2^{\mathsf{Res}}}$ can be naturally modeled by Boolean functions which in turn can be efficiently operated on, if these are represented through ordered binary decision diagrams.

The example program in Figure 1 does not use any procedures, which means we have to look at the constraint $R^\sharp 0$ and $R^\sharp 1$ only. Furthermore, each node is only reached with the static priority of its task. Therefore we directly obtain the following results:

Node	$[I'_e,3]$	$[I'_1,3]$	$[I'_2,3]$	$[I'_r,3]$
Result	$\{\emptyset\}$	$\{\{r'\}\}$	$\{\{r'\}\}$	$\{\emptyset\}$

Node	$[I_e,2]$	$[I_1,2]$	$[I_2,2]$	$[I_3,2]$	$[I_r,2]$
Result	$\{\emptyset\}$	$\{\{r\}\}$	$\{\{r\}\}$	$\{\{r\}\}$	$\{\emptyset\}$

Node	$[T_e,1]$	$[T_1,1]$	$[T_2,1]$	$[T_3,1]$	$[T_4,1]$
Result	$\{\emptyset\}$	$\{\{r'\}\}$	$\{\{r'\}\}$	$\{\{r'\}\}$	$\{\emptyset\}$
Node	$[T_5,1]$	$[T_6,1]$	$[T_7,1]$	$[T_8,1]$	$[T_9,1]$
Result	$\{\{r\}\}$	$\{\{r\}\}$	$\{\{r\}\}$	$\{\emptyset\}$	$\{\emptyset\}$
Node	$[T_{10},1]$	$[T_{11},1]$	$[T_r,1]$		
Result	$\{\{r\}\}$	$\{\{r\}\}$	$\{\emptyset\}$		

3.1 Flow-independent Must-resource Analysis

One interesting class of PCP programs consists of all programs where the set of held resources at a program point v does only depend on the set of resources held at the entry point of a procedure f but not on the concrete execution path reaching v. More precisely, we demand that for every procedure f and set of resources R, the set $\mathbf{S}[v,j]^\sharp(R)$ is a singleton set $\{R'\}$. These programs are said to have *flow-independent* resource sets. Such programs can be analyzed more efficiently, while still being precise w.r.t our model.

The analysis presented above can be further abstracted to compute not all possible resource sets, but the set of definitely held resource. The domain is then simplified to 2^{Res} ordered by superset inclusion ($\supseteq$) and intersection as the least upper bound. Since every program point is reachable by some same-level execution path, no dedicated bottom element is required to denote unreachability. We consider the abstraction functions $\alpha_m : (2^{\mathsf{Res}} \to 2^{2^{\mathsf{Res}}}) \to (2^{\mathsf{Res}} \to 2^{\mathsf{Res}})$ and $\bar{\alpha}_m : 2^{2^{\mathsf{Res}}} \to 2^{\mathsf{Res}}$ given by:

$$\alpha_m(M)(R) = \bigcap M(R) \qquad \bar{\alpha}_m(S) = \bigcap S$$

where $\bigcap \emptyset$ is defined as Res. The resulting analysis is called *must resource* analysis. The abstract functions $[\![e]\!]^\sharp_m : 2^{\mathsf{Res}} \to 2^{\mathsf{Res}}$ for edges $e = (u,s,v)$ corresponding to this analysis are given by:

$$\begin{array}{lcl} [\![(u,\mathsf{cmd},v)]\!]^\sharp_m(R) & = & R \\ [\![(u,\mathsf{get}(r),v)]\!]^\sharp_m(R) & = & R \cup \{r\} \\ [\![(u,\mathsf{rel}(r),v)]\!]^\sharp_m(R) & = & R \setminus \{r\} \end{array}$$

Note that these now are functions of the general format $g(R) = R \setminus K \cup G$ for suitable constant sets K, G. Let $\mathcal{F}$ denote the set of all these functions. This lattice is well known for *gen-kill* bit-vector analyses [12]. Since also the abstract composition of functions as used by the constraint system $S^\sharp$, for must resource analysis becomes ordinary composition of functions, must resource analysis for PCP programs can be performed by means of an interprocedural *gen-kill* approach. Similarly the abstract function application, becomes ordinary function application. Let $S^\sharp_m$ denote the constraint system over the complete lattice $\mathcal{F}$ corresponding to $S^\sharp$.

Theorem 2. *Assume that $[\![f]\!]^\sharp, \mathbf{S}[v]^\sharp$ and $[\![f]\!]^\sharp_m, \mathbf{S}[v]^\sharp_m$ are the least solutions of $S^\sharp$ and $S^\sharp_m$, respectively, where $\mathbf{S}[v]^\sharp(R) \neq \emptyset$ for all program points v, and resource sets R. Then for every procedure f, and program point v,*

$$\alpha_m([\![f]\!]^\sharp) = [\![f]\!]^\sharp_m \qquad \alpha_m(\mathbf{S}[v]^\sharp) = \mathbf{S}[v]^\sharp_m$$

The proof is analogous to the proof of Theorem 1. The non-emptiness assumptions are required as the functions from $\mathcal{F}$ only commute with *non-empty* least upper bounds (since $\bigcap \emptyset = \mathsf{Res}$).

In the analysis results of the example program of Figure 1 presented above every set of resource sets, has only one element. This is the case because the example program is flow-independent. Applying α_m in this case just removes the extra pair of set brackets. For the interrupt I' we obtain e.g.:

Node	$[I'_e,3]$	$[I'_1,3]$	$[I'_2,3]$	$[I'_r,3]$
Result	$\emptyset$	$\{r'\}$	$\{r'\}$	$\emptyset$

For programs with flow-independent resource sets satisfying assumption (S), we obtain:

Corollary 1. *Assume that all resource sets are flow-independent. Assume that $[\![f]\!], \mathbf{S}[v,i]$ and $[\![f]\!]^\sharp_m, \mathbf{S}[v]^\sharp_m$ are the least solutions of the constraint systems S and $S^\sharp_m$, respectively. Then for every program point v, possible static priority i and resource set R,*

$$\alpha([\![f]\!]\ i)(R) = \{[\![f]\!]^\sharp_m(R)\} \qquad \alpha(\mathbf{S}[v,i])(R) = \{\mathbf{S}[v]^\sharp_m(R)\}$$

The least solution to the system $S^\sharp_m$ can be computed in $\mathcal{O}(n \cdot o)$ if operations on resource sets (bit vectors) are counted for $\mathcal{O}(1)$. In case a program with flow-independent resource sets consisting solely of tasks, each program node needs to be analyzed for only a single context, the static priority of its task and the empty resource set. However, a node inside a procedure can be reached not only with different static priorities but also with several distinct resource sets. In order to deal with this situation, we propose to combine the

summary functions $[\![f]\!]^\sharp_m$ (which can be computed in polynomial time) with the constraint system $R^\sharp$. This is possible since, by Corollary 1, the functions $[\![f]\!]^\sharp$ can be recovered from the functions $[\![f]\!]^\sharp_m$ by: $[\![f]\!]^\sharp(R) = \{[\![f]\!]^\sharp_m(R)\}$.

4. Data Races

In this section, we apply the results from the last section to detect data races in PCP programs. One can think of a task using its dynamic priority to defend against interfering interrupts. Interrupts use their static priority to attack other tasks. Note that it is the static priority of an interrupt q which decides whether q may interfere with the computation of another task executing at a given dynamic priority level. We there refer to the dynamic priority level protecting an access as *defensive priority* and the static priority level of an access as its *offensive priority*.

Definition 3. Assume that $\mathbf{R}[v,i]^\sharp$ denotes the least solution to the constraint system $R^\sharp$. We define:

$$\mathcal{P}_o(x) = \bigvee\{\mathcal{P}(q) \mid x \in Acc(q), q \in \mathsf{Irpt}\}$$

$$\mathcal{P}_d(x) = \bigwedge\{\mathcal{P}(R) \vee i \mid (u,\mathsf{cmd},v) \in E, x \in Acc(\mathsf{cmd}), R \in \mathbf{R}[v,i]^\sharp \neq \emptyset\}$$

Where $\wedge$ and $\vee$ denote minimum and maximum, respectively.

These functions map global variables to their offensive and defensive priorities, respectively. We have:

Theorem 3. *If the program satisfies assumption (S), a data race occurs at x if and only if $\mathcal{P}_o(x) > \mathcal{P}_d(x)$.*

Proof. First we assume that we are given an execution path $\bar\pi(u,s,v)\langle q\rangle\pi\langle/q\rangle \in \mathbf{R}[v,i]$ for some static priority i where $\bar\pi$ is an execution path reaching program point u, π is a same-level execution path of the interrupt q, (u,s,v) accesses the global x and π also contains an access to x. Let $R = \mathcal{R}(\bar\pi,\emptyset)$ denote the resource set held when reaching v along $\bar\pi(u,s,v)$, and $j = i \vee \mathcal{P}(R)$ denote the corresponding dynamic priority. Since the interrupt q may occur after the execution of $\bar\pi$, we have $j < \mathcal{P}(q)$. By Theorem 1, $R \in \mathbf{R}[v,i]^\sharp$. Therefore, $\mathcal{P}_d(x) \le j < \mathcal{P}(q) \le \mathcal{P}_o(x)$. Conversely assume that $\mathcal{P}_o(x) > \mathcal{P}_d(x)$. This means that there is an interrupt q of priority $\mathcal{P}_o(x)$ which has a same-level execution path π containing an access to the global x. Furthermore, there is an edge (u,cmd,v) accessing x together with a static priority i and resource set $R \in \mathbf{R}[v,i]^\sharp$ such that $\mathcal{P}(R) \vee i = \mathcal{P}_d(x) < \mathcal{P}(q)$. Since $\mathbf{R}[u,i]^\sharp$ gathers all resource sets possibly reaching u and cmd may not change resource sets, R is also contained in $\mathbf{R}[u,i]^\sharp$. Therefore by Theorem 1, there exists an execution path $\bar\pi$ reaching u at static priority i with $\mathcal{R}(\bar\pi,\emptyset) = R$. It follows that $\bar\pi(u,\mathsf{cmd},v)\langle q\rangle\pi\langle/q\rangle$ is a reaching execution path from $\mathbf{R}[v,i]$, and we have a data race at x. □

If the program has flow-independent resource sets, and all accesses are reachable, an equivalent result can be achieved using the cheaper must-resource analysis $R^\sharp_m$. In a general setting this would still yield a safe over-approximation of data races. For the example program in Figure 1, $\mathcal{P}_o(x)$ and $\mathcal{P}_d(x)$ are as follows:

$$\mathcal{P}_o(x) = \mathcal{P}_o(y) = 2 \quad \mathcal{P}_o(z) = 3$$
$$\mathcal{P}_d(x) = \mathcal{P}_d(y) = 2 \quad \mathcal{P}_d(z) = 1$$

This means that x and y are safe, but there is a data race at z. Which is due to I' being possible at T_9. More generally we can say, that an access to x or y is safe at a defensive priority of 2 and higher, while z requires priority 3 or more.

5. Analyzing Transactionality

Nontransactional behavior occurs when a fragment of a program which is meant to be executed atomically is interrupted by a task which accesses data manipulated by this fragment. A write access of the interrupting task may result in inconsistent data for the program fragment, while a read access may supply the interrupt with inconsistent data. In the example in Figure 1 the main task switches the value of x and y and whenever accessing either one it holds the resource r guaranteeing exclusive access. However, it releases the resource between the two operations, which allows the interrupt I to modify x and y. However the old value of x is still stored in the local variable t which is used later to overwrite y. This is an instance of the nontransactional behavior described by Definition 2.

These problems are avoided if the defensive priority is sufficiently large not only for a single access, but for the whole program fragment in question. There are several subtle points to be taken into account. One point is, that a procedure may have leading and trailing parts which do not access globals. These parts should not influence the defensive priority of the procedure. Another one point is, that there can be calls to other procedures which might release a held resource and then, perhaps, acquire it again. This influences the defensive priority of the caller, no matter where in the callee the temporary decrease in priority occurs.

5.1 Tasks Without Procedures

Let us first consider PCP programs with tasks, but no procedure calls. Thus, transactional behavior may refer to tasks only. Assume that q is such a task with static priority j. Assume further that for every program point v of q, we are given a set $\mathbf{S}[v]^\sharp_m(\emptyset) \subseteq \mathsf{Res}$ of (definitely) held resources when reaching program point v. In particular, $\mathbf{S}[q_e]^\sharp_m(\emptyset) = \emptyset$. These sets allow to compute a (lower bound to) the dynamic priority of q when reaching program point v. This value is given by:

$$\mathcal{P}[v] = \mathcal{P}(q) \vee \mathcal{P}(\mathbf{S}[v]^\sharp_m(\emptyset))$$

Let $[p]_* = \{0,\dots,p\} \cup \{\infty\}$ equipped with the reverse natural ordering $\ge$. For convenience, we denote the componentwise ordering on pairs from $[p]^2_*$ by $\ge$ as well.

For each program point v, we determine values $\mathbf{S}[v]^\sharp_t \in [p]^2_*$ where the first component of $\mathbf{S}[v]^\sharp_t$ is the minimal dynamic priority attained on execution paths reaching v between the first and the last access to a global. It equals ∞ if and only if no global has been accessed so far. The second component is the minimal dynamic priority attained after the first access. The pairs $\mathbf{S}[v]^\sharp_t$ are characterized as the least solution (least w.r.t. to the ordering $\ge$) of the following constraint system:

$$\begin{array}{lcl}
\mathbf{S}[q_e]^\sharp_t & \le & (\infty,\infty) \\
\mathbf{S}[v]^\sharp_t & \le & \textbf{let } (a_1,a_2) = \mathbf{S}[u]^\sharp_t \\
 & & \textbf{in let } a = \mathcal{P}[v] \wedge a_2 \\
 & & \textbf{in } (a_2,a) \\
 & & (u,s,v) \text{ control-flow edge with } Acc(s) \neq \emptyset \\
\mathbf{S}[v]^\sharp_t & \le & \textbf{let } (a_1,a_2) = \mathbf{S}[u]^\sharp_t \\
 & & \textbf{in let } a = \ \textbf{if } a_2 < \infty \textbf{ then } \mathcal{P}[v] \wedge a_2 \\
 & & \qquad\qquad \textbf{else } \infty \\
 & & \textbf{in } (a_1,a) \\
 & & (u,s,v) \text{ control-flow edge with } Acc(s) = \emptyset
\end{array}$$

Let $\mathcal{P}_d(q)$ be the first component of $\mathbf{S}[q_r]^\sharp_t$. Then the task q is transactional, if and only if $\mathcal{P}_d(q) \ge \mathcal{P}_o(x)$ holds for all globals x accessed by q. Where $\mathcal{P}_o(x)$ is the offensive priority of the global x as defined in Definition 3. In case that the PCP program has flow-independent resource sets, also the reverse implication holds. We will not prove these statements here, since they follow from

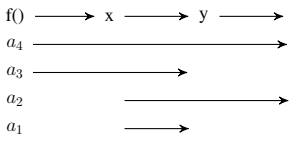

Figure 2. Priority ranges.

the corresponding properties of the more general interprocedural setting presented in the next subsection. Instead, we consider our introductory example.

Applying the analysis to the example program from Figure 1, we obtain the following results (showing only selected nodes):

Node	T_2	T_5	T_8	T_r	I_r	I'_r
Priorities	$(3,3)$	$(1,1)$	$(1,1)$	$(1,1)$	$(2,2)$	$(3,3)$

In each task, the return point is reachable from every program point of the task. Therefore, the defensive priorities are given by:

$$\mathcal{P}_d(T) = 1 \quad \mathcal{P}_d(I) = 2 \quad \mathcal{P}_d(I') = 3$$

We conclude that I' and I are transactional, since the sets of variables accessed by I and I' are disjoint, but T is not, since the offensive priority, e.g., of z exceeds 1. Note that if z is removed from the program, our analysis still detects that the task T does not swap x and y transactionally.

5.2 Tasks With Procedures

In presence of procedure calls, the dynamic priority when reaching a program point v of some procedure f, may depend on the static priority j of the task which executes f as well as the set R of resources held when f has been called. Let $\mathbf{S}[v]^\sharp_m(R)$ denote the set of resources which are (definitely) held when reaching program point v. Then (a lower bound to) the dynamic priority of v is given by $j \vee \mathcal{P}(\mathbf{S}[v]^\sharp_m(R))$.

Moreover, two values per program point do no longer suffice for analyzing transactionality, since we cannot exclude procedure parts before the first or after the last access to a global when a procedure is called. Therefore, we determine four values a_1, a_2, a_3, a_4. The first two components correspond to the two components which have been used in absence of procedures. Thus, the priority a_1 is the lowest priority attained between the first and the last access to some global variable, and a_2 is the lowest priority attained after the first access to a global. As before, a_1 receives the value of a_2 at every access to a global. At the return node of a procedure, the value of a_1 denotes the procedure's defensive priority.

Additionally, we compute the lowest priority obtained before the last access to a global variable in component a_3. For that, the component a_4 tracks the lowest priority encountered altogether. The component a_3 then receives the value of component a_4 at every access to a global. This is illustrated in Figure 2.

Let $\mathbb{D}$ denote the set of all quadruples $(a_1, a_2, a_3, a_4) \in [p]^4_*$ where $a_1 \geq a_2 \vee a_3$ and $a_4 \leq a_2 \wedge a_3$. Let the ordering on $\mathbb{D}$ again be the componentwise extension of the reverse natural ordering $\geq$ on quadruples. The minimal element w.r.t. this ordering thus is given by $(\infty, \infty, \infty, \infty)$ which signifies the empty set of same-level execution paths. The abstract effect $[\![(u,s,v)]\!]^\sharp_t : [p] \times 2^R \to \mathbb{D}$ of a control-flow edge (u,s,v) is defined by:

$$[\![(u,s,v)]\!]^\sharp_t(j,R) = \begin{cases} (\infty, j \vee \mathcal{P}(R), j \vee \mathcal{P}(R), j \vee \mathcal{P}(R)) & Acc(s) \neq \emptyset \\ (\infty, \infty, \infty, j \vee \mathcal{P}(R \backslash \{r\})) & s \equiv \mathsf{rel}(r)) \\ (\infty, \infty, \infty, j \vee \mathcal{P}(R)) & \text{otherwise} \end{cases}$$

The tuple $[\![(u,s,v)]\!]^\sharp_t(j,R)$ collects the defensive priorities of the execution of s with resource set R at static priority j.

The following constraint system characterizes the defensive priorities for triples (u,j,R) of nodes u, static priority level j and resource sets R. The resource set R denotes the set of resources held, when the current procedure has been called. Similarly for non-task procedures, j denotes the static priority of the calling task.

$$\begin{array}{lll} [\mathrm{S}^\sharp_\mathrm{t}0] & \mathbf{S}[q_e,j,\emptyset]^\sharp_t \leq (\infty,\infty,\infty,j) & q \in \mathsf{Task}, j = \mathcal{P}(q) \\ & \mathbf{S}[f_e,j,R]^\sharp_t \leq (\infty,\infty,\infty,j \vee \mathcal{P}(R)) & f \notin \mathsf{Task}, j \in [p] \\ [\mathrm{S}^\sharp_\mathrm{t}1] & \mathbf{S}[v,j,R]^\sharp_t \leq ([\![(u,s,v)]\!]^\sharp_t(j, \mathbf{S}[u]^\sharp_m(R))) \circ^\sharp_t \mathbf{S}[u,j,R]^\sharp_t & \\ & & (u,s,v) \in E \\ [\mathrm{S}^\sharp_\mathrm{t}2] & \mathbf{S}[v,j,R]^\sharp_t \leq ([\![f]\!]^\sharp_t(j, \mathbf{S}[u]^\sharp_m(R))) \circ^\sharp_t \mathbf{S}[u,j,R]^\sharp_t & \\ & & (u,f(),v) \in E \\ [\mathrm{S}^\sharp_\mathrm{t}3] & [\![f]\!]^\sharp_t\, j\, R \leq \mathbf{S}[f_r,j,R]^\sharp_t & \end{array}$$

where abstract composition $\circ^\sharp_t : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is defined by:

$$(b_1,b_2,b_3,b_4) \circ^\sharp_t (a_1,a_2,a_3,a_4) = \begin{cases} (\infty,\infty,\infty,\infty) & a_4 \vee b_4 = \infty \\ (a_1,a_2,a_3,a_4 \wedge b_4) & a_2 = \infty = b_2 \\ (b_1,b_2,a_4 \wedge b_3, a_4 \wedge b_4) & a_2 = \infty \neq b_2 \\ (a_1,a_2 \wedge b_4, a_3, a_4 \wedge b_4) & a_2 \neq \infty = b_2 \\ (a_2 \wedge b_3, a_2 \wedge b_4, a_4 \wedge b_3, a_4 \wedge b_4) & a_2 \neq \infty \neq b_2 \end{cases}$$

Note that a quadruple represents an execution path containing an access to some global if and only if the second component is less than ∞. Let $(c_1,c_2,c_3,c_4) = (b_1,b_2,b_3,b_4) \circ^\sharp_t (a_1,a_2,a_3,a_4)$. The abstract composition then is defined by case distinction on whether the represented execution paths contain accesses to globals or not. Assume for example the fourth case, i.e., that $a_2 \neq \infty$ and $b_2 = \infty$. Thus, the right quadruple represents an execution path π containing an access, while the left quadruple represents execution paths π' which do not access globals. In this case, $c_1 = a_1$ because first and last accesses in $\pi\,\pi'$ both are contained in π. Furthermore, $c_2 = a_2 \wedge b_4$ as the dynamic priorities encountered during the entire path π' occur after the first access to a global. The third component is $c_3 = a_3$, since the last access to a global occurs in π. Finally, the fourth component is $c_4 = a_4 \wedge b_4$ since this component provides the minimal dynamic priority encountered during the whole path $\pi\,\pi'$. The other cases are analogous with the exception of the first case, which takes care of bottom values. We have:

Proposition 1. *The abstract composition $\circ^\sharp_t : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is distributive in each argument, i.e., for all $a,b,c \in \mathbb{D}$,*

$$\begin{array}{rcl} c \circ^\sharp_t (a \wedge b) & = & (c \circ^\sharp_t a) \wedge (c \circ^\sharp_t b) \\ (a \wedge b) \circ^\sharp_t c & = & (a \circ^\sharp_t c) \wedge (b \circ^\sharp_t c) \end{array}$$

Proof. Let $a = (a_1,a_2,a_3,a_4)$, $b = (b_1,b_2,b_3,b_4)$ and $c = (c_1,c_2,c_3,c_4)$. Consider the first assertion of the proposition If $a_2 = b_2$, then the same case of the composition applies to $c \circ^\sharp_t (a \wedge b)$ as well as to $c \circ^\sharp_t a$ and $c \circ^\sharp_t b$, and the assertion follows by idempotency, commutativity and associativity of the minimum $\wedge$. Accordingly, assume that w.l.o.g. $a_2 = \infty$ and $b_2 \neq \infty$. If $c_2 = \infty$, we obtain:

$$\begin{array}{lll} c \circ^\sharp_t (a \wedge b) & = (a_1 \wedge b1, & a_2 \wedge b_2 \wedge c_2, \; a_3 \wedge b_3, \; a_4 \wedge b_4 \wedge c_4) \\ c \circ^\sharp_t a & = (\quad a_1 \quad , & \quad a_2 \quad , \quad a_3 \quad , \quad a_4 \wedge c_4 \quad) \\ c \circ^\sharp_t b & = (\quad b_1 \quad , & b_2 \wedge c_2 \quad , \quad b_3 \quad , \quad b_4 \wedge c_4 \quad) \end{array}$$

By inspection of each component, the assertion can be verified. If on the other hand, $c_2 \neq \infty$, we obtain:

$$\begin{array}{lll} c \circ^\sharp_t (a \wedge b) & = (a_2 \wedge b_2 \wedge c_3, & a_2 \wedge b_2 \wedge c_4, \; a_4 \wedge b_4 \wedge c_3, \; a_4 \wedge b_4 \wedge c_4) \\ c \circ^\sharp_t a & = (\quad c_1 \quad , & \quad c_2 \quad , \quad a_4 \wedge c_3 \quad , \quad a_4 \wedge c_4 \quad) \\ c \circ^\sharp_t b & = (\quad b_2 \wedge c_3 \quad , & b_2 \wedge c_4 \quad , \quad b_4 \wedge c_3 \quad , \quad b_4 \wedge c_4 \quad) \end{array}$$

In order to prove the assertion, it only must be proven for the first two columns that:

$$a_2 \wedge b_2 \wedge c_3 = c_1 \wedge b_2 \wedge c_3 \qquad a_2 \wedge b_2 \wedge c_4 = c_2 \wedge b_2 \wedge c_4$$

Recall that $a_2 = \infty$. Since by assumption on the quadruples in $\mathbb{D}$, $c_1 \geq c_3$ and $c_2 \geq c_4$, these equalities hold. This completes the proof of the first assertion. The proof of the second assertion is analogous. □

Assume that $\mathbf{S}[v, j, R]^\sharp_t$ and $[\![f]\!]^\sharp_t\ j\ R$ is the least solution of the constraint system $S^\sharp_t$. Assume further that $\mathcal{P}_o(x)$ is the offensive priority of the global x as defined in Definition 3. Then we have:

Theorem 4. *Assume that assumption (S) is satisfied, i.e., all program points are reachable by some same-level execution path. For a procedure f of a program with control-flow independent resource sets, a static priority j and a resource set R, let $(a_1, a_2, a_3, a_4) = [\![f]\!]^\sharp_t(j, R)$. Then the procedure f called with resource set R at static priority level j is nontransactional, if and only if there exists a global variable $x \in Acc(f)$ with $\mathcal{P}_o(x) > a_1$.*

Proof. For a given static priority level j and an initial resource set R, we assign to each same-level execution path π not containing interrupts, a priority tuple $(b_1, b_2, b_3, b_4) = \alpha_t(\pi, j, R) \in \mathbb{D}$ where b_1 is the minimal dynamic priority between the first and the last access to a global in π; b_2 is the minimal dynamic priority after the first access to the end of π; b_3 is the minimal dynamic priority from the beginning of π to the last access to a global; and b_4 is the minimal dynamic priority through π. By the definition of $[\![(u, s, v)]\!]^\sharp_t$ and $\circ^\sharp_t$, the tuple $\alpha_t(\pi, j, R)$ is inductively defined by:

$$\alpha_t(\epsilon, j, R) = (\infty, \infty, \infty, j \vee \mathcal{P}(R))$$

$$\alpha_t(\pi_1(u, s, v), j, R) = [\![(u, s, v)]\!]^\sharp_t(j, \mathcal{R}(\pi_1, R)) \circ^\sharp_t \alpha_t(\pi_1, j, R)$$

$$\alpha_t(\pi_1\langle f\rangle\pi_2\langle/f\rangle, j, R) = \alpha_t(\pi_2, j, \mathcal{R}(\pi_1, R)) \circ^\sharp_t \alpha_t(\pi_1, j, R)$$

We extend the mapping α_t to sets $S \subseteq \Pi_S$ of same-level execution paths by:

$$\alpha_t(S, j, R) = \bigwedge\{\alpha_t(\pi, j, R) \mid \pi \in S\}$$

Note that according to this definition, $\alpha_t(S, j, R) = (\infty,\infty,\infty,\infty)$ if and only if $S = \emptyset$. Let $\mathbf{S}[v, j]'(R)$ denote the set of same-level execution paths at static priority j and initial resource set R reaching v which do not contain interrupts. These sets can be characterized by a constraint system S' which is obtained from constraint system S characterizing all same-level execution paths by removing the constraints $S3$. By definition, the operator $\circ^\sharp_t$ preserves the least element. Since by Proposition 1, $\circ^\sharp_t$ is also distributive in each argument, it follows that $\alpha_t(\mathbf{S}[v, j]'(R), j, R) = \mathbf{S}[v, j, R]^\sharp_t$ for all program points v.

Now assume that the procedure f is nontransactional at static priority level j and initial resource set R, i.e., there is an execution path $\langle f\rangle\ \pi_1\ \langle q\rangle\ \pi\ \langle/q\rangle\ \pi_2\langle/f\rangle \in [\![f]\!](j)(R)$ such that $\pi_1\pi_2 \in \mathbf{S}[f_r, j](R)$ is a same-level execution path containing no interrupts; π is a same-level execution path of the interrupt q; both π_1 and π_2 contain an edge accessing a global; and π contains an edge which accesses a global variable $x \in Acc(f)$. Thus in particular, $\mathcal{P}_o(x) \geq \mathcal{P}(q)$. Let $\alpha_t(\pi_1\pi_2, j, R) = (b_1, b_2, b_3, b_4)$. Since the interrupt q may occur between the two global accesses in π_1 and π_2, we have $\mathcal{P}(q) > b_1$. From the definition of α_t we obtain that:

$$(b_1, b_2, b_3, b_4) \geq (a_1, a_2, a_3, a_4) = [\![f]\!]^\sharp_t(j, R)$$

Therefore,

$$\mathcal{P}_o(x) \geq \mathcal{P}(q) > b_1 \geq a_1$$

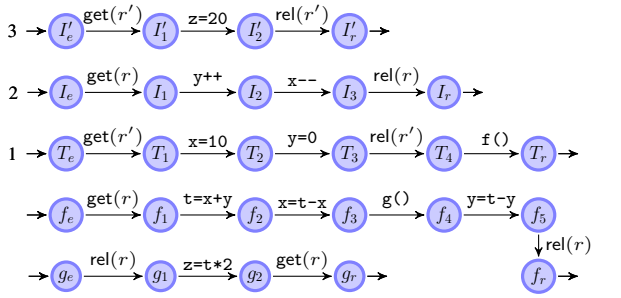

Figure 3. Example program with procedures.

Conversely assume that $[\![f]\!]^\sharp_t(j, R) = (a_1, a_2, a_3, a_4)$ and there is a global $x \in Acc(f)$ such that $\mathcal{P}_o(x) > a_1$. This means that $a_1 < \infty$, and that there is an interrupt q with $\mathcal{P}(q) = \mathcal{P}_o(x) > 0 = \mathcal{P}(main)$ which has a same-level execution path π which accesses x. Since $a_4 \leq a_1 < \infty$, Recall that

$$\begin{aligned}\mathbf{S}[f_r, j, R]^\sharp_t &= \alpha_t(\mathbf{S}[f_r, j]'(R), j, R)\\ &= \bigwedge\{\alpha_t(\pi', j, R) \mid \pi' \in \mathbf{S}[f_r, j]'(R)\}\end{aligned}$$

Therefore, there exists an interrupt-free same-level execution path $\pi' \in \mathbf{S}[f_r, j]'(R)$ such that a_1 equals the first component of $\alpha_t(\pi', j, R)$. Since $a_1 < \infty$, there exist at least two accesses to globals in π'. Moreover, π' can be factored into $\pi' = \pi_1\pi_2$ where π_1 and π_2 both contain at least one access to a global and the dynamic priority after π_1 equals a_1. Since $\mathcal{P}(q) > a_1$ interrupt q may occur between π_1 and π_2. Thus by Definition 2, f is not transactional when called at static priority j with resource set R. □

For programs with flow-dependent resource acquisition, this analysis still yields a safe over-approximation, i.e., transactional procedures may be considered as possibly nontransactional, but not the other way round. Alternatively, the flow-precise resource analysis $S^\sharp$ could be used to obtain more precise results.

Consider the example program of Figure 3 where the part of T after the initialization has been wrapped into the procedure f and the nodes T_9 to T_{10} of the original example program have into procedure g. The procedure g is transactional since it only contains one access to a global variable. The procedure f holds the resource r at all nodes between its first and last access to a global variable. However the call to g compromises the transactionality of f since it temporarily releases r. The analysis presented above captures this behavior. For the procedure g called at static priority 1 with resource set $\{r\}$ we obtain the following summary: $[\![g]\!]^\sharp_t(j, \{r\}) = (\infty, 1, 1, 1)$. When applied to the tuple reaching f_3 we have: $(\infty, 1, 1, 1) \circ^\sharp_t (2, 2, 1, 1) = (1, 1, 1, 1)$. Since tuple entries may not increase, this tuple is then carried over to f_r and with $\mathcal{P}_d(x) = 2$ we see that f is not transactional.

6. Linear Equalities For PCP Programs

The resource analysis presented in Section 3 need not explicitly deal with interrupts. The reason is that interrupts neither change sets of held resources nor affect the current priorities of tasks. Values of globals on the other hand, can be affected by interrupts.

Fortunately, summary-based interprocedural analyses [28] can be adapted to resource aware value analyses of PCP programs rather easily. As a prototypic example we extend the approach presented in [21] for analyzing *linear equalities* to take resources and interrupts into account. The goal of this analysis is to compute the linear closure of extended states of the collecting semantics.

For simplicity, we restrict ourselves to PCP programs with flow-independent resource sets.

Let $X = \{x_1, \ldots, x_k\}$ denote the set of global variables, where $k = |X|$ is the number of global variables. We consider affine assignments of the form $x_j := t_0 + \Sigma_{i=1}^{k} t_i \cdot x_i$ with $t_i \in \mathbb{Q}$. We write V for the complete lattice of linear subspaces of $(k+1)\times(k+1)$ matrices over the rationals $\mathbb{Q}$. Each matrix A describes the effect of an execution onto the global state. Thereby, each global state is represented by a vector $v = [v_0, \ldots, v_k]^\mathsf{T} \in \mathbb{Q}^{k+1}$ where $v_0 = 1$ and for $i > 0$, v_i records the value of the variable x_i. We write V' for the complete lattice of linear subspaces of $(k+1)$ vectors over the rationals $\mathbb{Q}$. The extra component v_0 allows to linearize affine transformations. A set $S \subseteq \mathbb{Q}^{(k+1)\times(k+1)}$ of transformation matrices is abstracted by the linear space $\mathbf{Span}\, S \in V$ of all linear combinations of matrices in S.

The concrete resource sensitive semantics $[\![(u,s,v)]\!]_l : 2^{\mathsf{Res}} \to \mathbb{Q}^{(k+1)\times(k+1)}$ of an edge is then given by:

$$[\![(u,s,v)]\!]_l(R) = \begin{cases} \left[\begin{array}{c|c|c} 1 & 0 & 0 \\ \hline 0 & I_{j-1} & 0 \\ \hline t_0 & t_1 \;\ldots\; t_k & \\ \hline 0 & 0 & I_{k-j} \end{array}\right] & s = x_j := t_0 + \Sigma_{i=1}^{k} t_i \cdot x_i \\ I_{k+1} & \text{otherwise} \end{cases}$$

Where I_m denotes the m-dimensional identity matrix. Note, that the resource parameter is not used here, but might be in another instance of the framework and is therefore given for completeness.

The resource sensitive semantics of an execution path $\pi \in \Pi$ is defined as follows:

$$\begin{array}{rcl} [\![\epsilon]\!]_l & = & I_{k+1} \\ [\![e\pi']\!]_l & = & [\![\pi']\!]_l \cdot [\![e]\!]_l \\ [\![\langle f\rangle\pi_1\langle/f\rangle\pi_2]\!]_l & = & [\![\pi_2]\!] \cdot [\![\pi_1]\!]_l \\ [\![\langle f_1\rangle\pi_1 \ldots \langle f_s\rangle\pi_s]\!]_l & = & [\![\pi_s]\!]_l \cdot \ldots \cdot [\![\pi_1]\!]_L \end{array}$$

The concrete resource sensitive collecting semantics $\mathbf{R}[u,i]_l : 2^{\mathsf{Res}} \to \mathbb{Q}^{k+1}$ of a program point u and static priority level i is then given by:

$$\mathbf{R}[u,i]_l(R) = \{[\![\bar\pi]\!](x) \mid R = \mathcal{R}(\bar\pi, \emptyset), x \in \mathbb{Q}^{k+1}, \bar\pi \in \mathbf{R}[u,i]\}$$

As in [21] we approximate this by its span, i.e.:

$$\mathbf{R}[u,i]^\sharp_l(R) = \mathbf{Span}(\mathbf{R}[u,i]_l(R))$$

We define the abstract semantics $[\![e]\!]^\sharp_l : 2^{\mathsf{Res}} \to V$ by $[\![e]\!]^\sharp_l(R) = \mathbf{Span}\{[\![e]\!]_l(R)\}$. Since in the end we are interested in equalities holding at a program point, we can model non-affine assignments, $x_j = ?$, by assignments of all constant values. For the abstract semantics we therefore have:

$$[\![x_j :=?]\!]^\sharp_l(R) = \mathbf{Span}\left\{ \left[\begin{array}{c|c|c} 1 & 0 & 0 \\ \hline 0 & I_{j-1} & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & I_{k-j} \end{array}\right], \left[\begin{array}{c|c|c} 1 & 0 & 0 \\ \hline 0 & I_{j-1} & 0 \\ \hline 1 & 0 & 0 \\ \hline 0 & 0 & I_{k-j} \end{array}\right] \right\}$$

To each procedure f, we assign an abstract function $[\![f]\!]^\sharp_l : [p] \to 2^{\mathsf{Res}} \to V$ which summarizes the abstract effect of f. These effects are characterized by the least solution of the constraint system:

$$\begin{array}{lll} [\mathrm{S}^\sharp_1 0] & \mathbf{S}[f_e,i,R]^\sharp_l \supseteq I & \\ [\mathrm{S}^\sharp_1 1] & \mathbf{S}[v,i,R]^\sharp_l \supseteq ([\![(u,s,v)]\!]^\sharp_l R) \circ^\sharp_l \mathbf{S}^\sharp_l[u,i,R]^\sharp_l & \\ & & (u,s,v) \in E \\ [\mathrm{S}^\sharp_1 2] & \mathbf{S}[v,i,R]^\sharp_l \supseteq ([\![f]\!]^\sharp_l\, i\, (\mathbf{S}[u]^\sharp_m(R))) \circ^\sharp_l \mathbf{S}[u,i,R]^\sharp_l & \\ & & (u,f(),v) \in E \\ [\mathrm{S}^\sharp_1 3] & \mathbf{S}[u,i,R]^\sharp_l \supseteq [\![j]\!]^\sharp_l \circ^\sharp_l \mathbf{S}[u,i,R]^\sharp_l & \\ & & u \notin N_e, j > i \vee \mathcal{P}(\mathbf{S}^\sharp_m[u](R)) \\ [\mathrm{S}^\sharp_1 4] & [\![f]\!]^\sharp_l\, i\, R \supseteq \mathcal{H}^\sharp_f(\mathbf{S}[f_r,i,R]^\sharp_l) & \\ & [\![j]\!]^\sharp_l \supseteq [\![q]\!]^\sharp_l\, j\, \emptyset & q \in \mathsf{Irpt}, \mathcal{P}(q=j) \end{array}$$

For two sets of matrices $M_1, M_2 \in V$, the operator $\circ^\sharp_l : V \times V \to V$ is defined as follows:

$$M_1 \circ^\sharp_l M_2 = \mathbf{Span}\{A_1 A_2 \mid A_1 \in M_1, A_2 \in M_2\}$$

By using the precomputed transformers for resource sets, we can statically check the priority condition including the resource set information and therefore, for this instance, we do not need a dedicated $\bullet^\sharp_l$ operator performing this check. Instead, we use $\circ^\sharp_l$ for interrupts as well. Since we consider global variables only, the operator $\mathcal{H}^\sharp_f$ which transforms the abstract semantics of a procedure into the abstract semantics of its call, is simply the identity. For tasks the resource set parameter R is always the empty set and the static priority i is the static priority of that task. Thus, if the program does not contain procedures, the summaries can be simplified to vector spaces of matrices.

In the second phase, we compute for every program point v and static priority i a mapping $\mathbf{R}^\sharp_l[v,i] : 2^{\mathsf{Res}} \to V'$. For any given resource set R, this mapping is meant to return the linear hull of all concrete states $x \in \mathbb{Q}^{k+1}$ possibly reaching program point v within a task with static priority i given the current resource set equals R. The set of all functions $2^{\mathsf{Res}} \to V'$ forms a complete lattice w.r.t. the partial ordering on functions induced by the partial ordering on V'. From a basis B of $R^\sharp_l[u,i](R)$ for a program point u with static priority i and resource set R, we obtain the set of valid equalities $t_0 + \sum_{i=1}^{k} t_i \cdot x_i = 0$ as the set of solutions of the system

$$t_0 \cdot b_0 + \ldots + t_k \cdot b_k = 0\ , \quad (b_0, \ldots, b_k) \in B$$

In order to describe the effect of a procedure f for this second phase, we rely on two ingredients:

- the summary $[\![f]\!]^\sharp_m$ computed by same-level must resource analysis, which records how the execution of f may change the sets of held resources, and
- the summary $[\![f]\!]^\sharp_l$ computed by the same-level analysis of linear transformations in the first phase, which for each static priority i and resource set before the call returns the vector space of possible linear transformations.

This yields the following constraint system:

$$\begin{array}{lll} [\mathrm{R}^\sharp_1 0] & \mathbf{R}[main_e\ , 0\ ]^\sharp_l \supseteq M_0 & \\ [\mathrm{R}^\sharp_1 1] & \mathbf{R}[v,i]^\sharp_l \supseteq ([\![e]\!]^\sharp_l, [\![e]\!]^\sharp_m) @^\sharp_l \mathbf{R}[u,i]^\sharp_l & \\ & & e = (u,s,v) \in E \\ [\mathrm{R}^\sharp_1 2] & \mathbf{R}[v,i]^\sharp_l \supseteq (([\![f]\!]^\sharp_l\, i), [\![f]\!]^\sharp_m) @^\sharp_l \mathbf{R}[u,i]^\sharp_l & \\ & & (u,f(),v) \in E \\ & \mathbf{R}[f_e,i]^\sharp_l \supseteq \mathsf{enter}^\sharp_{f,l}(\mathbf{R}[u,i]^\sharp_l) & (u,f(),v) \in E \\ [\mathrm{R}^\sharp_1 3] & \mathbf{R}[u,i]^\sharp_l \supseteq [\![j]\!]^\sharp_l @^\sharp_{j,l} \mathbf{R}[u,i]^\sharp_l & u \notin N_e, j > i \\ & \mathbf{R}[j]^\sharp_l \supseteq \mathsf{proj}^\sharp_{j,l}(\mathbf{R}_l[u,i]) & u \notin N_e, j > i \\ & \mathbf{R}[q_e,j]^\sharp_l \supseteq \mathsf{enter}^\sharp_{q,l}(\mathbf{R}[j]^\sharp_l) & q \in \mathsf{Irpt}, \mathcal{P}(q) = j \end{array}$$

Here, M_0 is the mapping which assigns the full vector space $\mathbb{Q}^{k+1}$ to the empty resource set $R = \emptyset$ and the zero space $\{\mathbf{0}\}$ to all resource sets $R \neq \emptyset$. The operator $@^\sharp_l : ((2^{\mathsf{Res}} \to V') \times (2^{\mathsf{Res}} \to 2^{\mathsf{Res}})) \times (2^{\mathsf{Res}} \to V') \to (2^{\mathsf{Res}} \to V')$ is defined by:

$$((M,h)@^\sharp_l \phi)(R') = \mathbf{Span}\{Ax \mid R' = h(R), A \in M(R), x \in \phi(R)\}$$

Since we consider global variables only, the function $\mathsf{enter}^\sharp_{f,l}$ is the identity function, i.e., $\mathsf{enter}^\sharp_{f,l}\, \phi = \phi$.

In the constraint $R^\sharp_l 3$, a modified version of the application and the enter operator are required which take into account that an interrupt q can only be enabled if the static priority of q exceeds the

dynamic priority at a given program point. Therefore, we define:

$$(M@^\sharp_{j,l}\phi)\, R = \mathbf{Span}\{Av \mid A \in M(\emptyset), v \in \phi(R), j > \mathcal{P}(R)\}$$

$$(\mathsf{proj}^\sharp_{j,l}\phi)\, R = \begin{cases} \mathbf{Span}\{v \in \phi(R') \mid j > \mathcal{P}(R')\} & \text{if } R = \emptyset \\ \{\mathbf{0}\} & \text{otherwise} \end{cases}$$

In any case we have the following theorem:

Theorem 5. *Assume that the PCP program has flow-independent resource sets and satisfies* (S) *Let furthermore* $\mathbf{S}[v,i]$, $[\![f]\!]$ *and* $\mathbf{S}[v,i,R]^\sharp_l$, $[\![f]\!]^\sharp_l$, $\mathbf{S}[v]^\sharp_m$, $[\![f]\!]^\sharp_m$ *as well as* $\mathbf{R}[v,i]^\sharp_l$ *denote the least solutions of the constraint systems* S, $S^\sharp_l$, $S^\sharp_m$, *and* $R^\sharp_l$, *respectively. Then the following holds:*

1. *For every program point* v, *static priority* i, *resource set* R *and procedure* f,

$$\mathbf{Span}\{[\![\pi]\!]_l \mid \pi \in \mathbf{S}[v,i](R)\} = \mathbf{S}[v,i,R]^\sharp_l$$
$$\mathbf{Span}\{[\![\pi]\!]_l \mid \pi \in [\![f]\!]\, i\, R\} = [\![f]\!]^\sharp_l\, i\, R$$

2. *For every program point* v, *static priority* i *and resource set* R

$$\mathbf{Span}\{[\![\bar\pi]\!]_l(x) \mid \bar\pi \in \mathbf{R}[v,i], x \in \mathbb{Q}^{k+1}, \mathcal{R}(\bar\pi,\emptyset) = R\} = \mathbf{R}[v,i]^\sharp_l(R)$$

For the constraint system $S^\sharp_l$ we have at most $n \cdot p^2 \cdot 2^o$ constraints. Using the techniques from [21], this implies that the least solution can be computed in time $\mathcal{O}(n \cdot p^2 \cdot 2^o \cdot k^8)$. This is a smooth generalization of the results obtained in [21] to the case of programs with interrupts and resources. Note that a practical implementation may explore the given constraint systems in a demand-driven fashion such that only those resource sets are considered which actually are necessary for computing the reachability information as provided by the least solution of $R^\sharp_l$. For the special case of tasks only without auxiliary procedures, this means that the factor 2^o can be dropped completely.

We exemplify our analysis for the example program in Figure 1. Due to their non-deterministic nature, interrupts may occur arbitrarily often, creating an infinite number of program executions for a given program point, each corresponding to linear transformations of the state vectors. Since we are only interested in the linear hull of these transformations, it suffices to maintain a basis of the generated sub-space of matrices. Accordingly, we obtain the following summaries for the interrupts I and I':

$$[\![I']\!]^\sharp_l(3,\emptyset) = \mathbf{Span}\left\{\begin{bmatrix} 1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\20&0&0&0&0\\0&0&0&0&1\end{bmatrix}\right\}$$

$$[\![I]\!]^\sharp_l(2,\emptyset) = \mathbf{Span}\left\{\begin{bmatrix} 1&0&0&0&0\\-1&1&0&0&0\\1&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{bmatrix}, \begin{bmatrix} 1&0&0&0&0\\-1&1&0&0&0\\1&0&1&0&0\\20&0&0&0&0\\0&0&0&0&1\end{bmatrix}\right\}$$

These transformations can be applied to compute the subspace of possible values for each program point of the main task. For T_4, we obtain:

$$\mathbf{R}[T_4,1]^\sharp_l\emptyset = \mathbf{Span}\left\{\begin{bmatrix}1\\10\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\-1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix}\right\}$$

The second vector is due to the potential occurrence of the interrupt I. The interrupt I' has no impact, since I' is not enabled and there are no restrictions on the value of z, at T_4. From the basis we obtain the system of equalities:

$$\begin{array}{rcrcrcrcl} 1\cdot t_0 & + & 10\cdot t_1 & & & & & & = 0\\ & & 1\cdot t_1 & + & (-1)\cdot t_2 & & & & = 0\\ & & & & & & 1\cdot t_3 & & = 0\\ & & & & & & & 1\cdot t_4 & = 0 \end{array}$$

Solving this yields the equality $x + y = 10$. Overall we have the following equalities:

Node	Example Equalities
T_3	$x = 10,\ y = 0,\ x + y = 10$
T_4, T_5	$x + y = 10$
T_6	$t = x + y = 10$
T_7	$x - y + t = 0$
T_8	$z = 10$
$T_9, \ldots, T_r$	$z = 20,\ t = 10$
I'_2, I'_r	$z = 20$

This means that at the end of the main task T and the interrupt I', the equality $z = 20$ holds. Moreover, the equality $10 = x+y$ holds although interrupts may freely occur at nodes T_4 and T_5. Finally, the local variable t equals 10, once it has been written. Note that the equalities for z and t cannot be guaranteed in I, since I may occur at node T_3 where both variables may still be uninitialized.

Note that while this analysis directly fits into the framework, it is also possible to use analyses, which for example use infinite lattices, evaluate branching conditions, or even are unable to provide closed function summaries. For infinite lattices one would use a demand-driven local solver ([9]) and possibly widenings ([6]. The branching conditions could be added to the flow-graph and the framework semantics extended to treat them as nops while allowing the concrete analysis instance to fully evaluate them. Finally summary functions can be handled by tabulating them. This technique is used in the Goblint analyzer and proved to be very useful.

7. Implementation

The data race and transactionality analyses from Sections 4 and 5, respectively, have been implemented in the analyzer *Goblint* for multi-threaded C [31]. This analyzer framework is based on a local fixpoint engine and provides basic analyses such as constant propagation and alias analysis, which then can be enhanced with additional specific domains and transfer functions. The analyzer differentiates between read and write accesses in order to avoid read-read warnings. Beyond the path-based approach presented in sections 4 and 5, *Goblint* takes conditions into account whenever possible. By that, the analysis may exclude some unrealizable execution paths and therefore may raise less false alarms.

The test suite consists of sample programs from the *nxtOSEK* implementation [29] together with our own examples. Program `biped_robot` is part of the control software of a self-balancing two wheeled robot, which uses resources to synchronize the balancing with remote control commands. The programs `xxx_test` are examples for preemptive scheduling (`pe`), resource synchronization (`res`), time-triggered tasks (`tt`) and usb communication (`usb`). Each of these tests uses two tasks and one resource. Programs `example` and `example_fun` are from Figure 1 and Figure 3, respectively. Program `pingpong` consists of two tasks which alternately set a variable to "ping" and "pong" synchronizing via a single resource. Program `counter` consists of an interrupt which increases two fields of a struct if an integer flag is set and does nothing otherwise as well as a task which unsets the flag, then prints the struct and re-sets the flag. The integer flag itself is protected by the resource.

The results of running the analyzer on these examples are summarized in Table 1. We ran these experiments on a Intel(R) Core(TM)2 Quad CPU machine with 3.00GHz under Ubuntu 10.04. The analyzer verifies that the two programs `pingpong` and `usb_test` are free of data races. The data races in both versions of the example program are discovered, as well as the unsafe access to counters in `pe_test` and `tt_test`. The race warning in `biped_robot` occurs since re-running the initialization task is not ruled out. For `counter`, race warnings are produced for the fields

Program	Size	Time	Race	Trans.
`biped_robot`	151 lines	0,02 s	1	0
`pe_test`	97 lines	0,06 s	1	1
`res_test`	74 lines	0,03 s	0	1
`tt_test`	101 lines	0,07 s	1	1
`usb_test`	140 lines	0,04 s	0	0
`example`	38 lines	0,01 s	1	1
`example_fun`	51 lines	0,01 s	1	2
`pingpong`	53 lines	0,03 s	0	0
`counter`	58 lines	0,02 s	2	1

Table 1. Result of analyzing example programs

of the struct since they are accessed both by the task and the interrupt without protection with a resource. While we verify that the integer flag accesses are safe, the analysis presented here approximates conditional branching with non-deterministic branching and therefore does not relate the flag value with the accesses to the struct. Other analyses provided by the *Goblint* analyzer, however, may split the execution path based on the value of the flag and thus may separately analyze the cases for a set and an un-set flag.

Regarding transactionality, the analyzer verifies that the tasks of `usb_test` and `pingpong` are transactional. It discovers the violation of transactionality of the running example and also produces warnings for `pe_test` and `tt_test` where the race occurs between accesses to the counter. In `res_test`, a variable whose accesses are otherwise protected by a resource, is *read* after the release of the resource — thus violating our definition of transactionality. In `biped_robot`, on the other hand, the accesses to globals may be involved in data races, but are the only accesses to globals in their respective procedures. Accordingly, transactionality is not violated. For `counter`, a transactionality warning is produced, since the resource is released between un-setting and re-setting the flag.

All examples are small and therefore analyzed in negligible time. In order to get an intuition how the analyses scales, we evaluated the analyzer on the synthetic benchmarks `chain_`n. While the estimates for the asymptotic complexity of our analyses grow exponentially with the number of resources, they depend only linearly on the program size. Therefore, we do not expect the program size to be the major bottleneck for scalability but the number of resourced and interrupt levels. Thus, we vary these latter parameters in the benchmarks. For $n \geq 1$, program `chain_`n has globals $x_0, \ldots, x_n$, n interrupt levels and n resources which are used to successively copy the value of the variable x_i into the variable x_{i-1}. The running times of the analyzer for the instances $n = 100, 200, \ldots, 1000$ are shown in Figure 4. For each of them, the analyzer verified absence of data races and transactionality of all tasks. The increase in run time for these instances is slightly worse than linear. Even for 1000 interrupt levels and resources, the runtime is still quite acceptable. The source code of our analyses, all benchmarks, and a script to run them are available at `http://goblint.in.tum.de/popl11.html`.

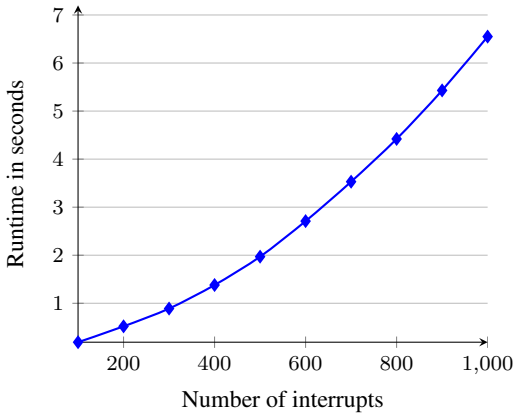

Figure 4. Runtimes of the analyzer on `chain_`n.

8. Related Work

While the ceiling and inheritance protocols [4, 27] have been formally studied [7, 23], these papers focus on schedulability rather than data consistency; that is, one assumes the program is correctly synchronized and characterizes the impact of synchronization primitives on meeting hard deadlines as a function of resource usage. In contrast, we are interested in detecting erroneous use of these synchronization primitives.

Already simple analysis problems for concurrent programs with recursion and synchronization are undecidable [24]. Practical approaches therefore either over-approximate the interaction between threads, as in thread-modular model checking [10], or ignore synchronization altogether [5, 8]. Some also place restrictions on concurrency. In a number of papers, Kahlon et al. [14–16] discuss model checking of pushdown systems synchronized via locks where the usage of locks must be well nested. This approach has been generalized to pushdown systems with dynamic thread creation [19, 20]. With the exception, perhaps, of dynamically changing priorities, PCP programs extended with dynamic task creation can also be cast within the more general model of multi-set pushdown systems. For these, Atig et al. [2] show that control point reachability is decidable. Their approach is based on Petri net reachability and does not support the inference of more complicated invariants such as linear equalities.

Kidd et al. [18] observe that every priority preemptive system can be transformed into a pushdown system. Beyond that, they additionally represent the schedulers corresponding to synchronization protocols as exponentially large pushdown systems. In case of PCP as well as for priority inheritance with well-nested resource usage, these two systems can be combined into one pushdown system. From that, they conclude that reachability is decidable for PCP as well as for priority inheritance with well-nested resource usage. Our approach cannot be applied to priority inheritance directly, while for PCP, our analysis is exponential only in the number of resources (not in the number of interrupts).

Summarizing abstract effects of interrupts has also been considered by Regehr et al. [26] for analyzing stack overflow in assembly code. Their summaries describe the stack consumption of an interrupt together with the set of interrupts which become enabled/disabled by the execution. Their model does not deal with specific protocols such as PCP. Additionally, Regehr and Cooprider [25] present a transformation technique to turn interrupt-driven embedded code into thread-based code and apply off-the-shelf race detection tools to the transformed code. They introduce artificial interrupt locks to make interrupt disabling/(re-)enabling visible to the analysis. They assume fixed static priorities and suffer from the imprecision possibly incurred by the thread analyzer. In contrast, our approach directly exploits the properties of the PCP protocol for interrupt-driven concurrency and explicitly deals with dynamically changing priorities. This allows us to handle the set of possible interleavings precisely.

Flanagan et al. [11] present a type system for atomicity in concurrent Java programs. A transactional procedure as defined in this paper is atomic in their sense since every execution of a transactional procedure that is possibly interrupted has an equivalent serial execution, i.e., no interrupts occur during its execution. This

atomicity condition is relaxed by Vaziri et al. [30], who provide a set of problematic access patterns and show that they are complete, i.e., the absence of all these patterns guarantee that the execution is serializable w.r.t. the critical variables and demarcated critical sections. Our notion of transactionality for a procedure f considers all, potentially accessed, global variables as critical for f where the critical section of f is the section between the first and last access to global variables. Kidd et al. [17] provide a static analysis for Java programs to verify the absence of such patterns in a given program. Artho et al. [1] give a static analysis to detect *stale-value* atomicity violations, i.e., accesses to outdated values of globals stored in a local variable that has escaped the critical section.

9. Conclusions

We have provided practical methods to analyze data races and transactionality in PCP programs. Moreover, our analysis of linear equalities can be considered as one instance of an analysis framework which generalizes the functional approach of [28] from programs with procedures to programs with procedures, interrupts, priorities and resources following the PCP protocol. Other instances of this framework can be obtained by providing specific domains V and V' for summary functions and abstract states, respectively, together with transfer functions for the basic statements and specific versions of the operators $\circ$, $\bullet_j$, $\mathcal{H}_f$, $@$, enter_f, $@_j$ and proj_j.

We have implemented the analyses of potential data races and transactionality within the static analyzer *Goblint* [31]. Preliminary experiments with typical examples as well as a scalable synthetic benchmark, are encouraging. Still, experiments with larger and more complicated real-world examples are desirable. We would also like to analyze further kinds of concurrency flaws in PCP programs and explore in how far the given approach can be generalized to more general programming models, e.g., PCP programs with task creation.

Acknowledgements

We thank Gordon Haak and Philipp Legrum for interesting us in PCP programs as well as providing example programs and valuable insights from a user's perspective. This work is supported by the joint DFG project OpIAT (MU 1508/1-1 and SE 551/13-1). The third author is partially supported by EstSF grant 8421.

References

[1] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In *ATVA'04*, vol. 3299 of *LNCS*, pp. 150–164. Springer, 2004.

[2] M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS'08*, vol. 2 of *LIPIcs*, pp. 37–48. Schloss Dagstuhl, 2008.

[3] Autosar consortium. *Autosar Architecture Specification, Release 4.0*, 2009. URL `http://www.autosar.org/`.

[4] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[5] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR'05*, vol. 3653 of *LNCS*, pp. 473–487. Springer, 2005.

[6] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *PLILP'92*, pp. 269–295. Springer, 1992.

[7] B. Dutertre. Formal analysis of the priority ceiling protocol. In *RTSS'00*, pp. 151–160. IEEE Press, 2000.

[8] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL'00*, pp. 1–11. ACM Press, 2000.

[9] C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Sci. Comput. Programming*, 35(2):137–161, 1999.

[10] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Comput. Sci.*, 338 (1-3):153–183, 2005.

[11] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Prog. Lang. Syst.*, 30(4):1–53, 2008.

[12] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[13] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *SafeCert'09*, ENTCS. Elsevier, 2010.

[14] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL'07*, pp. 303–314. ACM Press, 2007.

[15] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV'05*, vol. 3576 of *LNCS*, pp. 505–518. Springer, 2005.

[16] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07*, vol. 4590 of *LNCS*, pp. 226–239. Springer, 2007.

[17] N. Kidd, P. Lammich, T. Touili, and T. Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. In *SPIN'09*, vol. 5578 of *LNCS*, pp. 125–142. Springer, 2009.

[18] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all — reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN'10*, vol. 6349 of *LNCS*, pp. 245–261. Springer, 2010.

[19] P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS'08*, vol. 5079 of *LNCS*, pp. 205–220. Springer, 2008.

[20] P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with Tree-Regular constraints. In *CAV'09*, vol. 5643 of *LNCS*, pp. 525–539. Springer, 2009.

[21] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL'04*, pp. 330–341. ACM Press, 2004.

[22] OSEK/VDX Group. *OSEK/VDX Operating System Specification, Version 2.2.3*, 2005. URL `http://www.osek-vdx.org`.

[23] M. Pilling, A. Burns, and K. Raymond. Formal specifications and proofs of inheritance protocols for real-time scheduling. *Softw. Eng. J.*, 5(5):263–279, 1990.

[24] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.

[25] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *ENTCS*, 174(9):139–150, 2007.

[26] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4): 751–778, 2005.

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

[28] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pp. 189–234, 1981.

[29] Takashi Chikamasa et al. *OSEK platform for* LEGO® MINDSTORMS ®, 2010. URL `http://lejos-osek.sourceforge.net/`.

[30] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pp. 334–345. ACM Press, 2006.

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