



Leibniz Transactions on
Embedded Systems

Volume 5 | Issue 1 | October 2018

ISSN 2199-2002

Published online and open access by

the European Design and Automation Association (EDAA) / EMbedded Systems Special Interest Group (EMSIG) and Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Online available at

<http://www.dagstuhl.de/dagpub/2199-2002>.

Publication date

October 2018

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <http://dnb.d-nb.de>.

License

This work is licensed under a Creative Commons Attribution 3.0 Germany license (CC BY 3.0 DE): <http://creativecommons.org/licenses/by/3.0/de/deed.en>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier

10.4230/LITES-v005-i001

Aims and Scope

LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Editorial Board

- Alan Burns (Editor-in-Chief)
- Bashir Al Hashimi
- Karl-Erik Arzen
- Neil Audsley
- Sanjoy Baruah
- Samarjit Chakraborty
- Marco di Natale
- Martin Fränzle
- Steve Goddard
- Gernot Heiser
- Axel Jantsch
- Florence Maraninchi
- Sang Lyul Min
- Lothar Thiele
- Virginie Wiels

Editorial Office

Michael Wagner (*Managing Editor*)

Jutka Gasiorowski (*Editorial Assistance*)

Dagmar Glaser (*Editorial Assistance*)

Thomas Schillo (*Technical Assistance*)

Contact

Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office

Oktavie-Allee, 66687 Wadern, Germany

lites@dagstuhl.de

<http://www.dagstuhl.de/lites>


Contents

Risk-Aware Scheduling of Dual Criticality Job Systems Using Demand Distributions <i>Bader Naim Alahmad and Sathish Gopalakrishnan</i>	1:1–1:30
Errata for Three Papers (2004-05) on Fixed-Priority Scheduling with Self-Suspensions <i>Konstantinos Bletsas, Neil C. Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen</i>	2:1–2:20
The Semantic Foundations and a Landscape of Cache-Persistence Analyses <i>Jan Reineke</i>	3:1–3:52
A Static Analysis for the Minimization of Voters in Fault-Tolerant Circuits <i>Dmitry Burlyayev, Pascal Fradet, and Alain Girault</i>	4:1–4:26



Risk-Aware Scheduling of Dual Criticality Job Systems Using Demand Distributions

Bader Naim Alahmad

The University of British Columbia
2366 Main Mall, Vancouver, BC, Canada V6T 1Z4
bader@ece.ubc.ca
 <https://orcid.org/0000-0002-6409-1277>

Sathish Gopalakrishnan

The University of British Columbia
2332 Main Mall, Vancouver, BC, Canada V6T 1Z4
sathish@ece.ubc.ca

Abstract

We pose the problem of scheduling Mixed Criticality (MC) job systems when there are only two criticality levels, LO and HI—referred to as Dual Criticality job systems—on a single processing platform, when job demands are probabilistic and their distributions are known. The current MC models require that the scheduling policy allocate as little execution time as possible to LO-criticality jobs if the scenario of execution is of HI criticality, and drop LO-criticality jobs entirely as soon as the execution scenario’s criticality level can be inferred and is HI. The work incurred by “incorrectly” scheduling LO-criticality jobs in cases of HI realized scenarios might affect the feasibility of HI criticality jobs; we quantify this work and call it Work Threatening Feasibility (WTF). Our objective is to construct online scheduling policies that minimize the expected WTF for the given instance, and under which the instance is feasible in a probabilistic sense that is consistent with the traditional deterministic definition of MC feasibility. We develop a probabilistic framework for MC

scheduling, where feasibility is defined in terms of (chance) constraints on the probabilities that LO and HI jobs meet their deadlines. The probabilities are computed over the set of sample paths, or trajectories, induced by executing the policy, and those paths are dependent upon the set of execution scenarios and the given demand distributions. Our goal is to exploit the information provided by job distributions to compute the minimum expected WTF *below which the given instance is not feasible in probability*, and to compute a (randomized) “efficiently implementable” scheduling policy that realizes the latter quantity. We model the problem as a Constrained Markov Decision Process (CMDP) over a suitable state space and a finite planning horizon, and show that an optimal (non-stationary) Markov randomized scheduling policy exists. We derive an optimal policy by solving a Linear Program (LP). We also carry out quantitative evaluations on select probabilistic MC instances to demonstrate that our approach potentially outperforms current MC scheduling policies.

2012 ACM Subject Classification Mathematics of computing → Markov processes, Software and its engineering → Real-time systems software, Software and its engineering → Real-time schedulability

Keywords and Phrases Real-time scheduling; Mixed-criticality; Probability distribution; Chance-constrained Markov decision process; Linear programming

Digital Object Identifier 10.4230/LITES-v005-i001-a001

Received 2016-02-04 **Accepted** 2018-01-07 **Published** 2018-05-30

1 Introduction

We consider a system comprised of a *finite set of jobs* executing upon a *shared platform* (processor), and a *scheduling policy* that allocates processor time to jobs. A Mixed-Criticality (MC) real-time job system is one that carries out multiple jobs, with each job being of a specific criticality. For example, in an avionics/UAV system, some jobs relate to the flight stability or safety of the aircraft,



© Bader Naim Alahmad and Sathish Gopalakrishnan;
licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)

Leibniz Transactions on Embedded Systems, Vol. 5, Issue 1, Article No. 1, pp. 01:1–01:30



Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and these jobs have the highest criticality. Other jobs may relate to the mission of the aircraft (gather visual information of a particular region) and these jobs may be of a lower criticality. In the special and important case—that we consider in this article—where every job assumes one of exactly two criticality levels, LO or HI, we will refer to the MC system as Dual-Criticality.

From a *job scheduling* perspective, one would like to schedule jobs so that they meet their timing constraints or deadlines. To do so, one needs to know the execution time requirements of these jobs. Using worst-case execution time (WCET) estimates for execution time would lead to infeasibility of low criticality jobs (because the worst-case utilization could saturate the system capabilities) but, since worst-case execution times are rarely realized, one could use the same platform for jobs of all criticality levels provided the scheduler makes suitable choices when the execution duration of a job approaches the worst case or when it exceeds certain thresholds.

Vestal [38] was the first to offer an abstraction for scheduling MC job systems. In Vestal's model, there are $L \geq 2$ distinct criticality levels, and n jobs J_1, \dots, J_n . For notational convenience, we denote as $[n]$ the set $\{1, \dots, n\}$ for integers $n \geq 1$. Job J_i is characterized by the parameters (χ_i, c_i, d_i) , where

- $\chi_i \in [L]$ is job J_i 's **criticality**;
- $c_i = (c_i(1), \dots, c_i(L)) \in (0, \infty)^L$ is the vector of **WCET estimates** at all criticality levels;
- $d_i > 0$ is job J_i 's **deadline**.

For example, consider a triple-criticality MC job system consisting of three jobs J_1, J_2 , and J_3 with criticalities $\chi_1 = 1$, $\chi_2 = 3$, and $\chi_3 = 2$, respectively, and with the following WCET estimates:

$$\begin{aligned} J_1 : \quad & c_1 = (c_1(1) = \mathbf{90}, c_1(2) = 90, c_1(3) = 90) \\ J_2 : \quad & c_2 = (c_2(1) = 10, c_2(2) = 12, c_2(3) = \mathbf{20}) \\ J_3 : \quad & c_3 = (c_3(1) = 1, c_3(2) = \mathbf{500}, c_3(3) = 500). \end{aligned}$$

We shall make the following common monotonicity assumption: $c_i(1) \leq \dots \leq c_i(L)$ for every $i \in [n]$. Moreover, we will assume that $c_i(\ell) = c_i(\chi_i)$ for all $\ell \geq \chi_i$, so that it is sufficient to specify job J_i 's WCET estimates by giving $c_i(1), \dots, c_i(\chi_i)$, $i \in [n]$. An execution **scenario**, or **behavior**, is a particular realization of job demands in a particular run of the system; i.e., it is a vector $b = (b_1, \dots, b_n)$ in $\prod_{i=1}^n (0, c_i(\chi_i)]$. In our example, $(10, 11, 450)$ is a possible execution scenario. In any particular run of the system, the scenario remains unknown until *all* jobs finish execution. The criticality level of behavior b is defined as

$$\text{critDemand}(b) = \min\{\ell \in [L] : b_i \leq c_i(\ell) \quad \forall i \in [n]\}.$$

For instance, $\text{critDemand}((10, 11, 450)) = 2$. During a schedule, at time t , say, job J_i is said to be **operating** at criticality level $\ell \in [L]$ if it has been given at least $c_i(\ell - 1)$ but less than $c_i(\ell)$ units of execution, and has not finished execution at time t . We call this time-dependent quantity the **job's operational criticality level** at t . With the monotonicity assumption, the range of execution times that job J_i might demand when operating at criticality level ℓ is the open interval $(c_i(\ell - 1), c_i(\ell)]$, with the convention that $c_i(0) = 0$. In our example, if we take a snapshot of a certain schedule at, say time 63, and observe that jobs J_1, J_2 and J_3 have executed for 50, 10 and 3 time units, respectively, but J_2 has not yet finished execution, then J_2 's operational criticality level at time 63 is 2. However, if J_2 finishes execution at time 63 with 10 time units of execution, then its operational criticality level for all $t \leq 63$ is 1. The operational criticality level remains the same from time t until J_i either signals that it has finished execution, or it executes for $c_i(\ell)$ time unit at some $t' > t$ and does not signal completion, at which point its operational criticality level jumps to $\ell + 1$. As such, a job's operational criticality level is an increasing piecewise-constant function of time, demand, and the scheduling policy, with a (random) set of jump points.

At time t , the maximum of all job operational criticality levels is the **system operational criticality level** at time t . We note that the system operational criticality level of an observed allocation snapshot, say b , at some time, is *not* the same as $\text{critDemand}(b)$; the system operational criticality level depends on additional information not encoded in b , namely whether or not jobs finished execution, whereas $\text{critDemand}(b)$ assumes that all jobs finished execution. Since the system operational criticality level is defined in terms of the job operational criticality levels, the former is also an increasing piecewise-constant function. In our example, the system operational criticality level at time 63 with the same execution snapshot $(50, 10, 3)$ is 1 if J_2 finishes execution at or before time 63, and is 2 otherwise. If the scheduler selects J_2 to execute from time 63 to time 67, then at time 65, the system operational critical level makes a jump from 2 to 3 (since then J_2 has executed for $12 = c_2(2)$ time units and has not finished execution), and remains 3 until the end of the schedule.

Once a job signals that it has finished execution, its **demand** is realized. A demand realization is a scenario of execution. Every job demand realization maps naturally to a unique **job criticality level realization**, and the maximum of which across all jobs is the **system criticality level realization**. Different runs, or executions, of the input job system might yield different criticality level realizations, since, generally, a job might demand anything in $(0, c_i(\chi_i)]$, and job demand realizations might differ across different executions.

The Job Dropping Model: Literature and Optimality

In addition to Vestal [38], there has been a substantial body of work that analyzes scheduling policies for deterministic MC systems, wherein low(er) criticality jobs are dropped when a high(er) criticality job demands more execution time. One such approach was studied by Baruah et al. [8, 10]. In this approach, low criticality jobs are dropped when it is deemed necessary to allocate more time to a high criticality job. This decision is based on deterministic thresholds and is conservative in the sense that worst-case assumptions are made about the execution time requirements of the low criticality jobs and other high criticality jobs. As a consequence, low criticality jobs may miss deadlines even when it may be possible to meet the deadlines for high and low criticality jobs. Feasibility of a given Dual-Criticality instance in this model is defined as follows: For every scenario of execution, if the scenario's criticality level is LO, all jobs should be given enough execution time to complete entirely and should meet their deadlines, but if the scenario's criticality level is HI, only HI-criticality jobs need to be given execution budget and must complete before their deadlines. In the latter case, giving any execution time to LO-criticality jobs is considered as an erroneous allocation, and doing so negatively affects the achievable processor utilization.

A **non-clairvoyant**, or **online**, scheduling policy does not know the scenario of execution in advance, and only an omniscient clairvoyant policy knows the realized scenario at time 0, and is therefore able to decide whether or not to drop LO-criticality jobs at the beginning of system operation and thus achieve the maximum processor utilization. An instance $I = (J_1, \dots, J_n; L)$ is said to be **correctly MC-schedulable** by scheduling policy π if for every scenario (behavior) $b \in \prod_{i=1}^n (0, c_i(\chi_i)]$, if b has criticality level ℓ , then every J_i with $\chi_i \geq \ell$ can be given b_i units of execution during $[0, d_i]$ under π . An instance I is said to be **MC-feasible** if there is an *online* scheduling policy under which I is correctly MC-schedulable.

Baruah et al. [10] showed that checking MC-feasibility can be reduced to checking its defining condition only for the scenarios that assume the WCET estimates; i.e., for $b \in \{(c_1(\ell_1), \dots, c_n(\ell_n)) : \ell_i \in [\chi_i]\}$. The MC-feasibility problem was shown to NP-Hard in the strong sense [7]; however, it is not yet clear whether or not MC-feasibility belongs to the class NP.

If an instance I is not MC-feasible, then there is no online scheduling policy under which it is correctly MC-schedulable. Conversely, if instance I is MC-feasible, then an online scheduling policy that correctly MC-schedules I may or may not exist.

A widely used measure of the performance of non-clairvoyant MC-scheduling policies is the processor **speed-up factor** (Baruah et al. [8], Kalyanasundaram and Pruhs [30]). It is defined as follows: If π is a non-clairvoyant scheduling policy, then its speed-up factor is the smallest real number $s > 1$ such that, for *every* MC instance I , if I is MC-feasible on a unit-speed processor, then policy π will correctly MC-schedule I on an s (or more)-speed processor. An **optimal** policy is one that minimizes s .

In the MC context, a non-unit speed-up factor arises because of the following: A non-clairvoyant algorithm has only WCET estimates available, and it does not know the scenario of execution in advance, so in high criticality scenarios, the algorithm might allocate execution time to jobs whose criticality is less than the realized system criticality level. Thus the earlier the time at which the scenario's criticality level is inferred under the scheduling policy *while preserving feasibility* (in the MC sense), the less the "processor time waste" the policy incurs for that scenario. Since the scheduler can drop all LO-criticality jobs as soon as the scenario's criticality is inferred as HI, predicting the earliest such time plays a central role in the MC-scheduling problem. Given an MC job instance, a scenario of execution and a non-clairvoyant scheduling policy, we call the earliest time instant at which the execution scenario's criticality level is inferred with certainty the **Time of Criticality Inference** (T_{CI}) associated with the scheduling policy for the given scenario. Giving any execution time to LO-criticality jobs early on in the schedule will only delay the T_{CI} , and thus delay the time instant at which we can decide whether or not to drop LO-criticality jobs. However, to preserve the schedulability of LO-criticality jobs in case the scenario is of LO criticality, the policy must judiciously give execution time to LO-criticality jobs early on in the schedule. Thus, we are facing conflicting objectives, and the optimal scheduling policy must strike the right allocation balance.

Here is an example to illustrate the situation.

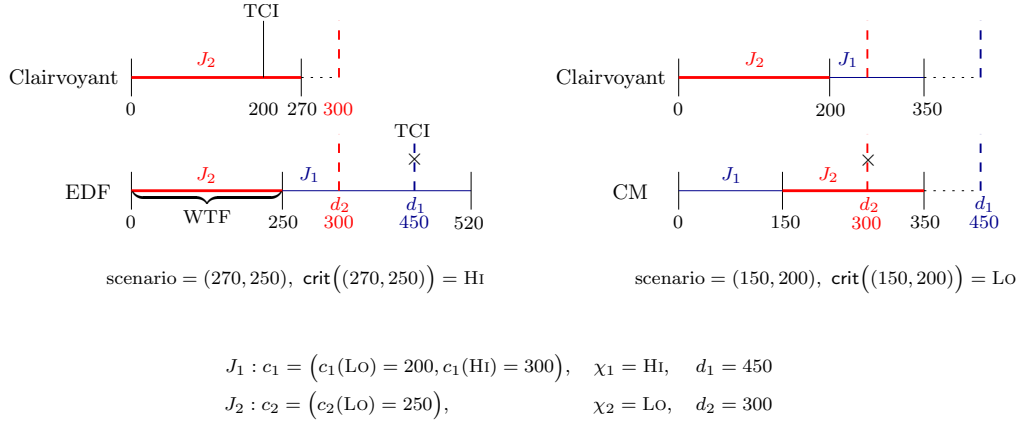
► **Example 1.** Consider a dual-criticality MC job system consisting of two jobs J_1 and J_2 with the following parameters:

$$\begin{aligned} J_1 : \quad & c_1 = (c_1(\text{LO}) = 200, c_1(\text{HI}) = 300), \quad \chi_1 = \text{HI}, \quad d_1 = 450 \\ J_2 : \quad & c_2 = (c_2(\text{LO}) = 250), \quad \chi_2 = \text{LO}, \quad d_2 = 300. \end{aligned}$$

First let us examine how the clairvoyant algorithm would MC-schedule this job instance. If the scenario is of HI criticality, then the clairvoyant policy knows this at time 0 and drops J_2 entirely and schedules J_1 , which would then meet its deadline of 450. If the scenario is of LO criticality, then the clairvoyant policy schedules both J_1 and J_2 using the Earliest Deadline First (EDF) policy, and they both meet their deadlines: The worst-case LO-criticality scenario is (200, 250), and under EDF, J_2 is scheduled first and finishes at time $250 < d_2 = 300$, and then J_1 occupies the processor till time 450 ($= d_1$). Now we consider two non-clairvoyant scheduling policies (see Figure 1):

- **EDF.** Suppose that the scenario of execution is (270, 250), which has HI criticality. EDF first schedules J_2 up to time 250, and then selects J_1 to occupy the processor until time 520. At time 450, however, J_1 misses its deadline.
- **Criticality Monotonic (CM),** which is a fixed-priority scheduling policy that at each instant schedules, among the jobs that have not finished execution, the job with the highest criticality. Suppose that the scenario of execution is the LO-criticality (150, 200). J_1 occupies the processor from time 0 to time 150, then J_2 executes until time 350. J_2 , however, misses its deadline at time 300.

The problem with EDF is that it does not consider criticalities, and consequently, in our example, it scheduled J_2 when it should have dropped it altogether. The work done by J_2 affected



■ **Figure 1** Optimal clairvoyant vs. non-clairvoyant EDF (left) and clairvoyant vs. CM (right) schedules for the job set of Example 1. EDF incurs WTF of 250, causing J_1 to miss its deadline at 450. CM does not allocate the LO-criticality J_2 enough execution time earlier in the schedule so as to guarantee its feasibility if the realized scenario is Lo, which is the case in this example. This causes J_2 to miss its deadline at time 300.

the feasibility of J_1 (which is the only job whose feasibility matters given that the scenario is of Hi criticality). We call this processor time waste—caused by lack of knowledge of the scenario—**Work Threatening Feasibility (WTF)**. In the example, EDF incurred WTF of 250 for the given scenario, caused by scheduling J_2 . *The speed-up factor of a given scheduling policy measures its worst case (maximum) incurred WTF across all MC instances that are MC-feasible (on a unit-speed processor).* We note that WTF is only incurred for scenarios that have Hi criticality and, in this case, by giving execution time to LO-criticality jobs; it is zero for LO-criticality behaviors. The problem with CM, on the other hand, is that it does not care about the feasibility of LO-criticality jobs in light of a LO behavior, although it causes the system criticality level to be realized the soonest possible.

Our example suggests that to both minimize the WTF and guarantee feasibility, the scheduling policy must strive to achieve a balance between the following conflicting objectives:

- O1.** It should allocate LO-criticality jobs sufficiently enough execution times early on; in particular, prior to the T_{CI} , so as to guarantee their schedulability in the case where the realized behavior is of LO criticality, and
- O2.** It should minimize any WTF, by
 - a. driving the revelation of the system criticality level sufficiently quickly by scheduling HI-criticality jobs, so as to decide whether to drop LO-criticality jobs as soon as possible, and
 - b. minimizing the allocation in O1 if the scenario is HI-criticality (it is here where the objectives are conflicting).

Probabilistic MC-Model: Justification

The MC model we consider in this article is a probabilistic variant of the MC model thus described. But *why use a probabilistic MC model?* First, the current MC standards and accreditations express the required performance guarantees of MC software components as failure probabilities. For

■ **Table 1** DO-178B Criticality Specifications (AdaCore [1]).

Level	Failure Condition	Failure Rate Limit (failures/hour)	Example
A	Catastrophic	10^{-9}	Fly-by-wire
B	Hazardous	10^{-7}	Fuel management
C	Major	10^{-5}	Pilot/ATC communication
D	Minor	10^{-3}	Flight data recorder
E	No effect	n/a	Entertainment system

instance, the DO-178B avionics standard¹ lists 5 levels of criticality, and specifies for each criticality level an upper bound on the *failure rate* of software components having that criticality (Table 1). From the scheduling perspective, job failures are deadline misses. Then given how MC systems are specified in practice, we believe that a probabilistic framework is the natural setting in which MC systems ought to be framed and reasoned about.

Note. Failure rate estimation is a research problem in its own right, but is outside the scope of this article. We refer the reader to Shooman [36] for an in-depth account of failure rate estimation in avionics software systems, along with feasibility studies and the associated analysis.

Second, without any additional information about job demands other than WCET estimates, the scheduler is oblivious to job demand realizations prior to job completion, until the realizations present themselves online at one of the system operational criticality level jump instants. As a consequence, working with WCET estimates solely will lead to underutilization of the processor when the WCETs are not realized.

Contribution

Whereas the contribution in this specific article relates to a specific restriction of the MC job model, the overall thrust of our work is to develop a framework for reasoning about workload of different criticality levels and providing probabilistic guarantees about the successful execution of jobs. The one-shot *job* model that we consider—as opposed to the more complex *recurrent task model*—was, as we shall see below, studied extensively in the context of MC scheduling, and it remains highly relevant due to the complexity of MC scheduling problems. We have chosen this particular model as a first step towards reasoning about recurring tasks. One can interpret our work as providing the boundaries for synthesizing feasible policies.

This article is an attempt to reconcile the widely used job-dropping model and the mixed-criticality specifications as instituted by the current standards and the industry requirements. Baruah’s work and ours have following in common: We both regard allocating execution times to LO-criticality jobs in cases of HI-criticality execution scenarios as undesirable behavior that the scheduling algorithm should avoid. In our model, however, feasibility is defined more generally, and our definition includes Baruah’s definition as a special case: We are given upper bounds on the probabilities that jobs at each criticality level miss their deadlines, and one of our goals is to determine a policy under which the probabilities of deadline misses respect the user-supplied failure tolerance parameters. Toward this goal, we introduce the notion of **probably feasible** MC instances in the job dropping model (for the precise definitions, see Definition 4).

¹ Titled *Software Considerations in Airborne Systems and Equipment Certification*, and developed jointly by RTCA SC-167 and EUROCAE WG-12.

We propose an approach for Dual-Criticality job systems that is not deterministic, and uses the probability distribution of job execution times. Our contribution is a model of MC job systems as a **chance-Constrained Markov Decision Process (CMDP)** that then allows us to provide guarantees around jobs meeting their timing constraints with high probability. The chance constraints are sample path constraints on the trajectories of the MDP induced by executing a policy, and they represent the *risk* of missing deadlines at the various criticality levels. We show how to derive a randomized non-stationary Markov scheduling policy that is expected WTF-optimal, by solving a linear program.

This approach can be computationally expensive, but we envisage this as a first step in enabling such probabilistic analysis. Nevertheless, the problem is amenable to approximation, and we briefly outline one method that can be used to obtain approximately optimal and approximately feasible scheduling policies.

More Literature

Before concluding this section, we mention some prior work related to MC-scheduling and to probabilistic analysis of real-time systems.

Baruah and Vestal [11] showed that for recurrent MC task systems, Earliest Deadline First (EDF) does not dominate Rate-Monotonic (RM), and neither are optimal for scheduling MC tasks in the job dropping model. The Own Criticality-Based Priority (OCBP) algorithm was among the first algorithms designed specifically for the scheduling of (deterministic) MC job systems within the job dropping model [10]. OCBP is a fixed-priority scheduling policy, and it utilizes Audsley's priority assignment scheme [6]. OCBP was shown to be optimal in the class of fixed-priority MC-scheduling algorithms in the *speed-up factor*, with a speed-up factor of $(\sqrt{5} + 1)/2$ for dual-criticality job system. It was shown that if an instance I is OCBP-schedulable, then it is MC-feasible; thus, correct schedulability by OCBP is sufficient for MC-feasibility, and the correct MC-scheduling policy is given by the OCBP priorities. Conversely, if I is MC-feasible, then OCBP might or might not correctly MC-schedule I ; however, if I is MC-feasible, then OCBP can correctly MC-schedule I on a speed $(\sqrt{5} + 1)/2$ processor, or, in other words, OCBP is capable of correctly MC-scheduling the (smaller) instance where every given WCET is divided by the speed-up factor $(\sqrt{5} + 1)/2$. This quantifies how inexact OCBP is.

The MC-EDF algorithm [37] was shown to dominate OCBP, in the sense that there are (deterministic) MC-feasible instances that are deemed MC-schedulable by MC-EDF but not by OCBP.

Guo and Baruah [22] studied the scheduling of MC jobs (with job dropping) on a single processor with varying speeds. The authors of the latter extended their work to the sporadic task model with implicit deadlines [9]. Chen et al. [15] devised a deadline-tightening technique for scheduling MC *sporadic task* systems on a unit-speed single processor, wherein virtual deadlines that are shorter than the actual deadlines are assigned to the higher criticality jobs. Again, low(er) criticality tasks may be rejected in order to satisfy the demands of high(er) criticality tasks. We refer the reader to the manuscript by Burns and Davis [14] for the most current and comprehensive overview of MC systems and related problems.

The probabilistic analysis of (non-MC) real-time systems is not new. Díaz et al. [16] analyzed the behavior of fixed-priority (e.g., RM) and dynamic-priority (e.g., EDF) scheduling algorithms for recurrent, stochastically independent tasks when execution times are random variables. The goal of their work is to compute the probability of deadline miss as well as the (random) response-time of every task. See also [17, 18, 19, 31]. Maxim and Cucu-Grosjean [33] extended the probabilistic analysis framework of Díaz et al. [16] for fixed-priority scheduling schemes to task systems where also the minimum inter-arrival times between job invocations as well as task deadlines may be

random variables. Their focus was to efficiently compute the response time of each task under the assumption that tasks are stochastically independent. They do so by using convolution of probability distributions as the key underlying mathematical operation.

To the best of our knowledge, there is no work that aims at identifying feasible scheduling policies for MC job systems where job execution times are random. Alahmad et al. [2] were the first to propose the consideration of probabilistic execution times for MC systems. Guo et al. [23] carried out schedulability analysis of EDF applied to recurrent MC task systems, wherein lower priority tasks are given guarantees against failure. The latter is the closest work we are aware of to our efforts in this article. However, our problem is substantially harder, because it is concerned with *synthesizing* MC scheduling policies, as opposed to *analyzing* existing (fixed) scheduling policies.

2 System Model

We will adopt Vestal's model described above, but we will frame it in a probabilistic setting. The system we consider is that of n jobs executing upon a single processor, and all jobs are ready to execute at time 0. We will make use of the sets $\mathbb{N} = \{1, 2, \dots\} \subset \{0, 1, 2, \dots\} = \mathbb{Z}_+$. For ease of reference, we give in Table 2 a listing of most of the notation used in this article.

Note: The purpose of this section is to present as general a probabilistic framework for MC systems. As such, the exposition to follow will be in terms of general probability spaces, arbitrary number of criticality levels, with no assumptions about the random demands except boundedness. *This setting is, however, much more general than the actual problem that we consider, which is a specialization of the framework to be presented to two criticality levels and discrete demands.*

In addition to the parameters (χ_i, c_i, d_i) described earlier, the execution **demand** of job J_i is described by a random variable

$$\zeta_i : \Omega_i \rightarrow (0, c_i(1)] \cup \dots \cup (c_i(\chi_i - 1), c_i(\chi_i)] = (0, c_i(\chi_i)]$$

on a probability space $(\Omega_i, \mathcal{M}_i, \mathbb{P}_i)$, where Ω_i is the scenario space associated with job J_i consisting of all possible execution scenarios, \mathcal{M}_i is the set of possible (observable, measurable) events, and \mathbb{P}_i is a probability measure on Ω_i .

We will assume that the jobs are *independent*; that is, the demand random variables ζ_1, \dots, ζ_n are independent. The distribution of ζ_i is the probability measure $\mathbb{P}_{\zeta_i} \equiv \mathbb{P}_i \circ \zeta_i^{-1}$ on $(0, c_i(\chi_i)]$, and \mathbb{P}_{ζ_i} is known. Accordingly, job J_i is characterized by the tuple $((\Omega_i, \mathcal{M}_i, \mathbb{P}_i), \zeta_i, \chi_i, c_i, d_i)$, $i \in [n]$. The actual execution time that a job consumes at run-time (upon completion) is a **job demand realization**. The demand realization of a job is not known prior to its completion. A job **completes** execution when it announces, or signals, that it has finished execution; i.e., when the demand realization has presented itself. The latter happens when the job has been allocated enough execution time to produce its output entirely.

To this end, let

$$\Omega = \prod_{i=1}^n \Omega_i, \quad \mathcal{M} = \bigotimes_{i=1}^n \mathcal{M}_i,$$

where $\bigotimes_{i=1}^n \mathcal{M}_i$ is the product σ -algebra; that is, the σ -algebra with respect to which all the projection (coordinate) maps $\text{proj}_i : \Omega \rightarrow \Omega_i$ are measurable. Let \mathbb{P} be the product measure on (Ω, \mathcal{M}) ; i.e., \mathbb{P} is such that for every rectangle $A \in \mathcal{M}$, where $A = A_1 \times \dots \times A_n$ and $A_i \in \mathcal{M}_i$,

$$\mathbb{P}(A) = \mathbb{P}(A_1 \times \dots \times A_n) = \prod_{i=1}^n \mathbb{P}_i(A_i). \quad (1)$$

■ Table 2 Notation

Notation	Meaning
\mathbb{Z}_+	: $\{0, 1, 2, \dots\}$
\mathbb{N}	: $\{1, 2, \dots\}$
$[m]$, where $m \in \mathbb{N}$: $\{1, \dots, m\}$
\mathbb{R}	: The real numbers
$n \in \mathbb{N}$: Number of input jobs
$c_i(\ell) > 0$: WCET estimate of job J_i at criticality level ℓ
χ_i	: Criticality level of job J_i
$d_i > 0$: Deadline of job J_i
Ω_i	: Scenario space of job J_i
\mathcal{M}_i	: Set of events; subsets of Ω_i (σ -algebra)
\mathbb{P}_i	: Probability measure on the scenario space Ω_i of job J_i
$\bigotimes_{i=1}^n \mathcal{M}_i$: n -fold product σ -algebra
\mathbb{P}	: Probability measure on the product scenario space
ζ_i, Z_i	: Demand random variables
G_{Z_i}	: Distribution function of random variable Z_i
\mathbb{E}	: Expectation operator with respect to product scenario space
$\mathbb{1}_E(x)$: Indicator function of a set E
$\delta_x(E)$, E is a set, x a point	: Dirac measure
proj_i	: Projection (coordinate) map, returns i th component of a given vector
$\mathbf{0}, \mathbf{1}$: All zeros and all ones vectors
e_i	: Unit vector whose i th coordinate is 1
\mathbf{A}	: $\{e_1, \dots, e_n\} \cup \mathbf{0}$, Action space of the MDP
\mathbf{S}	: State space of the MDP
a_t	: n -component vector, action taken at time t
y_t	: n -component binary vector of job finish signals at time t
x_t	: n -component vector, cumulative execution time allocations up to time t
r_t	: scalar error flag
$s_t = (t, y_t, x_t, r_t)$: State of the MDP at time t
$\mathbf{A}(s_t)$: Admissible actions in state s_t
$\pi(ds_t \mid s_{t-1}, a_{t-1})$: Markov policy
$Q(ds_t \mid s_{t-1}, a_{t-1})$: State transition kernel of MDP
H_∞	: Canonical trajectory space of the MDP induced by executing policy π
$\{A_t\}, \{S_t\}, \{Y_t\}, \{R_t\}$, $t \in \mathbb{Z}_+$: Action, state, finish signal, and error stochastic processes on H_∞
$\text{crit} : \Omega = \prod_{i=1}^n \Omega_i \rightarrow \mathbb{N}$: Scenario criticality level
$\text{critDemand} : B \equiv \prod_{i=1}^n (0, c_i] \rightarrow \mathbb{N}$: Demand realization criticality level
$\text{critPath} : H_\infty \rightarrow \mathbb{N}$: System criticality level of trajectory (path)
$\text{critState} : \mathbf{S} \rightarrow \{\text{Lo}, \text{Hi}, \text{UNKNOWN}\}$: Operational system criticality level given a state
\mathbb{P}^π	: The (unique) probability measure on H_∞
\mathbb{E}^π	: Expectation with respect to H_∞
$F_i \equiv F_i^\pi : H_\infty \rightarrow \mathbb{N}$: (Random) Finish time of job J_i with respect to policy π
$T_{\text{Lo}} \equiv T_{\text{Lo}}^\pi : H_\infty \rightarrow \mathbb{N}$: Earliest time at which Lo system criticality level is inferred by policy π
$T_{\text{Hi}} \equiv T_{\text{Hi}}^\pi : H_\infty \rightarrow \mathbb{N}$: Earliest time at which Hi system criticality level is inferred by policy π
$T_{\text{CI}} \equiv T_{\text{CI}}^\pi$: $\min(T_{\text{Lo}}, T_{\text{Hi}})$, T_{CI} of a trajectory in H_∞
$w : \mathbf{S} \times \mathbf{S} \rightarrow \mathbb{Z}_+$: Local (immediate, per stage) objective cost function of MDP
$W \equiv W^\pi : H_\infty \rightarrow \mathbb{Z}_+$: WTF random variable on trajectory space
$\kappa : \mathbf{S} \rightarrow \{0, 1\}$: Immediate constraint cost function of MDP
$C \equiv C^\pi : H_\infty \rightarrow \mathbb{Z}_+$: Constraint cost random variable on trajectory space

01:10 Risk-Aware Scheduling of Dual Criticality Job Systems Using Demand Distributions

We shall denote vectors $\omega \in \Omega$ as $\omega_1, \dots, \omega_n$, where $\omega_i \in \Omega_i$ is the i th coordinate of ω . We extend every ζ_i to be defined on Ω as follows. Let $Z_i = \zeta_i \circ \text{proj}_i$. Then $Z_i : \Omega \rightarrow (0, c_i(L)]$ depends only on the i th coordinate of a given $\omega \in \Omega$; that is,

$$Z_i(\omega) = \zeta_i(\text{proj}_i(\omega)) = \zeta_i(\omega_i) \quad (\omega \in \Omega).$$

We define the demand vector $Z = (Z_1, \dots, Z_n) : \Omega \rightarrow \prod_{i=1}^n (0, c_i(L)]$. Then

$$\begin{aligned} Z^{-1}(C_1 \times \dots \times C_n) &= (Z_1, \dots, Z_n)^{-1}(C_1 \times \dots \times C_n) \\ &= \{\omega \in \Omega : Z_i(\omega) \in C_i \quad \forall i \in [n]\} & [= \bigcap_{i=1}^n Z_i^{-1}(C_i)] \\ &= \{\omega \in \Omega : \text{proj}_i^{-1}(\omega) \equiv \omega_i \in \zeta_i^{-1}(C_i) \quad \forall i \in [n]\} \\ &= \prod_{i=1}^n \zeta_i^{-1}(C_i). \end{aligned} \tag{2}$$

Then the definition of \mathbb{P} implies that the distribution of Z , \mathbb{P}_Z , is such that

$$\begin{aligned} \mathbb{P}_Z(C_1 \dots C_n) &= \mathbb{P}(Z^{-1}(C_1 \dots C_n)) \\ &= \mathbb{P}\left(\bigcap_{i=1}^n Z_i^{-1}(C_i)\right) = \mathbb{P}\left(\prod_{i=1}^n \zeta_i^{-1}(C_i)\right) =_{(*)} \prod_{i=1}^n \mathbb{P}_i(\zeta_i^{-1}(C_i)) = \prod_{i=1}^n \mathbb{P}_{\zeta_i}(C_i), \end{aligned}$$

where equality $(*)$ follows by (1).

We will let $G_{\zeta_i}(t) = \mathbb{P}_{\zeta_i}((-\infty, t])$ denote the **distribution function** of ζ_i . In the probabilistic setting, every $\omega \in \Omega$ is a **scenario** of execution, and $Z(\omega)$ is the corresponding **system demand realization** (contrast these definitions with their counterparts in the deterministic setting described above). Every execution scenario maps to a unique **system criticality level realization** through the function $\text{crit} : \Omega \rightarrow [L]$, where

$$\text{crit}(\omega) = \min\left\{\ell \in [L] : Z_i(\omega) \in (0, c_i(\ell)] \text{ for all } i \in [n]\right\}. \tag{3}$$

That crit is defined for all scenarios $\omega \in \Omega$ follows by monotonicity of $c_i(\ell)$ with respect to ℓ .

Fix $\ell \in [L]$. For a scenario $\omega \in \Omega$, by (3), $\text{crit}(\omega) = \ell$ if there is *at least* one job, say J_i , such that $c_i(\ell - 1) < Z_i(\omega) \leq c_i(\ell)$, while the remaining jobs are such that $Z_j(\omega) \leq c_j(\ell)$. For $\ell \in [L]$, by independence of job demands,

$$\mathbb{P}(\text{crit} \leq \ell) = \mathbb{P}\left(\bigcap_{i=1}^n \{Z_i \leq c_i(\ell)\}\right) = \prod_{i=1}^n G_{\zeta_i}(c_i(\ell)).$$

Since every scenario has a unique criticality level,

$$\mathbb{P}(\text{crit} \leq \ell) = \sum_{k=1}^{\ell} \mathbb{P}(\text{crit} = k).$$

Therefore,

$$\mathbb{P}(\text{crit} = \ell) = \mathbb{P}(\text{crit} \leq \ell) - \mathbb{P}(\text{crit} \leq \ell - 1) = \prod_{i=1}^n G_{\zeta_i}(c_i(\ell)) - \prod_{i=1}^n G_{\zeta_i}(c_i(\ell - 1)),$$

with the convention that $c_i(0) = 0$. Specializing to the dual criticality case, where $\text{Lo} \equiv 1$ and $\text{Hi} \equiv 2$, we have

$$\mathbb{P}(\text{crit} = \text{Lo}) = \prod_{i=1}^n G_{\zeta_i}(c_i(\text{Lo})), \quad \mathbb{P}(\text{crit} = \text{Hi}) = 1 - \prod_{i=1}^n G_{\zeta_i}(c_i(\text{Lo})). \tag{4}$$

Now we recast the definitions made earlier in terms of scenarios spaces, random variables, and the functions that we have just defined.

A **scheduling policy** is a rule that at every time instant decides which job, from the set of available jobs (those that have not finished execution), is assigned the processor. At every time instant, a scheduling policy may use the characterizing parameters of all jobs, as well as its previous decisions, in making its next job allocation decision.

► **Definition 2** (Correct MC-Schedulability). A policy π is said to **correctly MC-schedule** an instance $I = (J_1, \dots, J_n; L)$ if for every scenario $\omega \in \Omega$, every J_i with $\chi_i \geq \text{crit}(\omega)$ receives $Z_i(\omega)$ units of execution during $[0, d_i]$ under π .

We stress again that this definition does not require that jobs whose criticality is less than that of the realized system criticality level be given any execution; in fact, we will consider doing so as an undesired allocation scheme that is wasting the processor utilization.

► **Definition 3** (MC-Feasibility, Classical). An instance $I = (J_1, \dots, J_n; L)$ is MC-feasible if there is an online (non-clairvoyant) scheduling policy π under which I is correctly MC-schedulable.

Since our setting is probabilistic, we will be concerned with the notions of probabilistic feasibility and *expected* WTF-optimality. We defer the formal definitions of these notions until we have precisely defined the stochastic process induced by a policy, and the underlying probability space over which the expectation is taken (Definitions 4 and 5 in section 3.4).

3 Problem Definition: Integer Demands and Dual Criticalities

We consider a specialization of the setting discussed in the previous section, in which all demand random variables are integer-valued², and the system is dual-criticality. We are given two error parameters: One is a lower bound on the probability that all n jobs finish at or before their deadlines if the system criticality level is realized as LO, and the other is a lower bound on the probability that HI-criticality jobs meet their deadlines if the system criticality level is realized as HI. We are required to compute a scheduling policy that minimizes, in expectation, the time wasted scheduling LO-criticality jobs if the system criticality level turns out to be HI, while respecting the deadline miss constraints. That is, we want to compute a policy that minimizes the WTF for the given instance, while respecting the timeliness constraints given by the error parameters. We will make the definition of deadline miss probability precise in sections to follow. Formally, the demand becomes the random variable

$$\zeta_i : \Omega_i \rightarrow \{1, 2, \dots, c_i(\text{LO}), c_i(\text{LO}) + 1, \dots, c_i(\chi_i)\}.$$

Accordingly, all demand realizations are integers, and we will therefore consider scheduling at integer boundaries. We shall assume that a job system is MC if not all the input jobs have the same criticality.

3.1 MDP Setup

Let $Y = \{0 : \text{finished}, 1 : \text{not finished}\}^n$, and let $y_t \in Y$ be the following variable (n -component vector): $y_t^i = 1$ iff job J_i still requires execution at time t , and $y_t^i = 0$ iff J_i has finished execution. At time 0, all jobs require execution, so we shall assume that $y_0 = \mathbf{1}$, the vector of all 1s. The

² One may equally well work with rational times by regarding time as being divided into integer multiples of some fixed rational quantum $q > 0$, and using scaling arguments to convert to integers.

evolution of the system state depends on the policy, so will specify precisely how the system evolves after formally defining policies.

Let \mathbf{A} be the set of control actions (here jobs) available to the scheduler. We will let $\mathbf{A} = \{e_1, \dots, e_n\} \cup \{\mathbf{0}\}$, where $\mathbf{0}$ is the vector of all 0s, and $\{e_1, \dots, e_n\}$ is the standard basis for \mathbb{R}^n ; e_i is the unit vector that is 1 at the i th coordinate and 0 elsewhere. If the action taken at time $t \in \mathbb{Z}_+$ is $a_t \in \mathbf{A}$, then $a_t = e_i$ means that job J_i occupies the processor during $[t, t+1]$. If $a_t = \mathbf{0}$, then no job is scheduled and the processor is kept idle. Let $x_t = (x_t^1, \dots, x_t^n)$ encode the amount of execution time that every job has been allocated up to the beginning of the t th epoch (before acting at time t); that is, $x_0 = \mathbf{0}$ and $x_t = \sum_{m=0}^{t-1} a_m$ for $t \in \mathbb{N}$. Then for every $t \in \mathbb{Z}_+$ and $i \in [n]$, $x_t^i \in \mathbf{X}_i = \{0, 1, \dots, c_i(\chi_i)\}$. We will let $\mathbf{X} = \prod_{i=1}^n \mathbf{X}_i$. We will utilize a variable r_t to “mark” the state as “error”. An error flag stamped on a state signifies a deadline miss. r_t assumes values in

$$\mathbf{R} = \{\text{not error, potential error, error, error'}\}.$$

The **state** of the scheduling system at time $t \in \mathbb{Z}_+$ is $s_t = (t, y_t, x_t, r_t) \in \mathbf{S}$, where $\mathbf{S} \subset \{0, \dots, N\} \times \mathbf{Y} \times \mathbf{X} \times \mathbf{R}$. The rationale behind our choice of this particular design of system state will become clear during the derivation of the state process below. For now, we mention how each element comprising our state representation achieves a desirable merit we seek in the system state:

- t : The main reason we include time is that we want to encode job finish times in the state, because we will identify “error” states as those where some job’s finish time exceeds its deadline. As a byproduct, augmenting the state with time will result in time-homogeneous (stationary) state transition dynamics (Hernández-Lerma [25], p. 13);
- y_t : Implements the idea that a job *signals* that it has finished execution; this is the only state element that we *observe*, the others we *set* according to y_t ;
- x_t : Summarizes all we need to know about the decisions we have made (allocations) up to time t , thus eliminating the need to include all actions up to time t . This is the key to ensure that the state process, which we derive below, is a *Markov chain*;
- r_t : One case where a state s_t becomes **error** is if some HI-criticality job $i \in \mathcal{I}_{\text{HI}}$ has just missed its deadline, which happens when $t = d_i$ and J_i still requires execution ($y_t^i = 1$). In this case, we will set $r_t = \text{error}$. When $r_t = \text{error}$, we will set $r_{t'}$ to **error'** for all $t' > t$; we do so to avoid charging the trajectory of execution more than once if more than one job miss their deadlines (see (17) and the discussion thereafter). Another possible error scenario is that when some LO-criticality job $i \in \mathcal{I}_{\text{LO}}$ has just missed its deadline ($t = d_i$) and still demands execution ($y_t^i = 1$), and the system criticality level realization is inferred as LO at or before t ; that is, all HI-criticality jobs have already finished execution with LO demand realizations ($y_t^j = 0$ and $x_t^j \leq c_j(\text{LO})$ for all $j \in \mathcal{I}_{\text{HI}}$). However, those are not the only cases where the state becomes error. Consider the more subtle situation where no job has missed its deadline prior to time t , and s_t is such that there is $i \in \mathcal{I}_{\text{LO}}$ that just missed its deadline ($t = d_i$ and $y_t^i = 1$), but the scenario’s criticality level realization is not yet determinable; in terms of our control variables, there is a non-empty $F \subset \mathcal{I}_{\text{HI}}$, possibly all of \mathcal{I}_{HI} , such that every job j in F has executed for at most $c_j(\text{LO}) - 1$, and none of the jobs in F have finished execution ($y_t^j = 1$ and $x_t^j < c_j(\text{LO})$ for all $j \in F$), while the other $k \in \mathcal{I}_{\text{HI}} \setminus F$, if any, have finished already with LO demand realizations ($y_t^k = 0$ and $x_t^k \leq c_k(\text{LO})$ for all $k \in \mathcal{I}_{\text{HI}} \setminus F$). In this case, the LO-criticality job J_i that just missed its deadline does not drive the system into an error state at time t since, by our definition of MC feasibility, this cannot be decided until we know the execution scenario’s criticality level realization with certainty, which here depends on the (yet unknown) demand realizations of the jobs in F . In such case, we will say that the system is *potentially* in error state at time t , and we set $r_t = \text{potential error}$ to “remember” that a LO-criticality job has

missed its deadline at t . Doing so gives us the facility to decide later whether or not the system is in error state—as soon as the scenario’s criticality level realization is inferred—and, as a consequence, deduct the penalties correctly in the MDP.

For $t \in \mathbb{N}$, let $(S \times A)^t$ be the Cartesian product of $S \times A$ with itself t times. Define the set of **admissible histories** up to time t as $H_0 = S$, and

$$H_t = (S \times A)^t \times S \quad (t \in \mathbb{N}).$$

Every element of H_t is called a **t -history**, and has the form

$$h_t = (s_0, a_0, \dots, s_{t-1}, a_{t-1}, s_t).$$

t -histories are the information available to the scheduler before making its job selection decision at time t .

Let $A(s_t) \subset A$ be the set of actions that the scheduler is allowed to apply at time t when the scheduling system is in state s_t . We shall call $A(s_t)$ the set of **admissible actions** in state s_t . A **scheduling policy** is a sequence $\pi = \{\pi_t : t \in \mathbb{Z}_+\}$, where π_t is a stochastic kernel on $A(s_t)$ given H_t . That is, if we denote the power set of a set X as 2^X , then $\pi_t \equiv \pi_t(da_t | h_t)$, where $\pi_t : 2^{A(s_t)} \times H_t \rightarrow [0, 1]$ is such that

- (i) for every $B \in 2^{A(s_t)}$, $\pi_t(B | \cdot)$ is a function from H_t to $[0, 1]$, and
- (ii) for every $h_t \in H_t$, $\pi_t(\cdot | h_t) : 2^{A(s_t)} \rightarrow [0, 1]$ is a probability measure on $A(s_t)$.

The state s_t summarizes all allocation decisions and remaining demands up to time t . We will restrict our attention to *Markov* policies, where $\pi_t(a_t | h_t) = \pi_t(a_t | s_t)$ for every h_t ([26] Definition 2.3.2 a).

A Note on Terminology: Since our state and action spaces are finite, all the stochastic (transition) kernels here can be represented by *transition matrices*. In this article, however, we will not use any of the matrix algebra machinery used to analyze Markov chains, so we will present our framework in the language of stochastic kernels.

A **work-conserving** scheduling policy always schedules a job that still demands execution; i.e., it never keeps the processor idle whenever there is a job that has not finished execution. Thus a policy is non-work-conserving iff there is $t \in \mathbb{Z}_+$ such that $a_t = \mathbf{0}$ (no job is selected) and there is $i \in [n]$ such that J_i has not finished execution; i.e., $y_t^i = 1$. The epoch $N = \sum_{i=1}^n c_i(\chi_i)$ is an upper bound on our planning horizon. With N fixed, any trajectory induced by executing a work conserving policy satisfies 1) $\sum_{i=1}^n x_t^i = t$ for every $t \in \{0, \dots, N\}$ for which $y_t^i = 1$ for some $i \in [n]$, and 2) $x_t = x_T$ for all $t \in \{T, \dots, N\}$, where T is the first time instant at which all jobs finish execution. A non-work-conserving schedule will only delay job completions and the time at which the criticality level realization can be inferred, so we restrict ourselves to work-conserving policies.

We implement the requirement that the scheduling policy be work-conserving by specifying that $A(s_t)$ includes only vectors e_i for which $y_t^i = 1$, if any. We will *drop* LO-criticality jobs (temporarily) as soon as the state s_t indicates that the operational system criticality level is HI, and we will enforce this by placing further restrictions on $A(s_t)$. Namely, given state s_t , if there is $i \in \mathcal{I}_{\text{HI}}$ such that both $x_t^i \geq c_i(\text{LO})$ and $y_t^i = 1$, then the operational system criticality level at time t is HI and there are HI-criticality jobs still requiring execution, so we exclude from $A(s_t)$ all LO-criticality jobs. Otherwise, we include all LO-criticality jobs that have not finished yet. If s_t does not satisfy the latter condition, then either the system criticality level realization cannot be inferred at t , or all HI-criticality jobs finished with LO demand realizations before or at t , or the

system criticality level was known before or at t as HI, but all HI-criticality jobs have finished execution at t . In the last case, we might have dropped LO-criticality jobs earlier, and, since scheduling LO-criticality jobs at time t in this case is not considered WTF (and does not affect feasibility), we may bring back any LO-criticality jobs that still need to execute. In summary,

$$A(s_t) = \begin{cases} \{e_i : y_t^i = 1, \chi_i = \text{HI}\} & (*) \text{ if there is } i \in \mathcal{I}_{\text{HI}} \text{ such that } x_t^i \geq c_i(\text{LO}) \text{ and } y_t^i = 1 \\ \{e_i : y_t^i = 1\} & \text{if } (*) \text{ not satisfied and there is } i \in [n] \text{ such that } y_t^i = 1 \\ \{\mathbf{0}\} & \text{otherwise (all jobs finished execution).} \end{cases}$$

Control Model

Scheduling decisions are made at every $t \in \{0, \dots, N-1\}$ exclusively. If a certain job is chosen to execute at some t , then this job occupies the processor for the duration $[t, t+1]$, without interruption, until the scheduler is invoked again at $t+1$. We call $[t, t+1]$ the t th **control interval**. At any $t > 0$, if job J_i was chosen to occupy the processor during $[t-1, t]$ (i.e., $a_{t-1} = e_i$), then the scheduler knows at time t whether or not job J_i requires more execution by observing the value of y_t^i , which will be set to finished if job J_i signals that it has finished execution at time t . The other jobs' demands are not affected by scheduling job J_i , and whether or not the other jobs require more execution does not change in $[t-1, t]$. The information available to the scheduler at the beginning of the t th control interval is a_{t-1} , y_t , x_t , and r_t .

Let $s = (t, y, x, r)$ and $\hat{s} = (\hat{t}, \hat{y}, \hat{x}, \hat{r})$. It is necessary for transition (s, a, \hat{s}) to be valid that all the following be satisfied:

$$\begin{aligned} \mathbf{NC} : \quad & \hat{t} = t + 1, \\ & \sum_{i=1}^n x^i = t, \\ & a = \hat{x} - x = e_i \text{ for some } i \in [n], \text{ or } a = \hat{x} - x = \mathbf{0} \\ & y - \hat{y} \in \{\mathbf{0}, e_i\} \text{ for the same } i, \text{ and} \\ & (r, \hat{r}) \notin \{(\text{error}, \text{not error}), (\text{error}, \text{potential error}), (\text{not error}, \text{error}'), \\ & \quad (\text{potential error}, \text{error}'), (\text{error}', r) \mid \forall r \in \mathbb{R} \setminus \{\text{error}'\}\}. \end{aligned}$$

However, not all state transitions satisfying **NC** are valid, as we will describe below. All invalid state transitions have $Q(\{\hat{s} \mid s, a) = 0$, however. To this end, we note that the state includes all the information necessary to determine whether or not the system criticality level is inferred, and if so, determine its value. To simplify the exposition, we define a function $\text{critState} : \mathcal{S} \rightarrow \{\text{LO}, \text{HI}, \text{UNKNOWN}\}$, where $\text{critState}(s)$ is the system criticality level realization, and is defined as follows: For $s = (t, x, y, r)$,

- (1) $\text{critState}(s) = \text{LO}$ if all HI-criticality jobs finished execution with LO demand realizations; that is, if $y^i = 0$ and $x^i \leq c_i(\text{LO})$ for all $i \in \mathcal{I}_{\text{HI}}$;
- (2) $\text{critState}(s) = \text{HI}$ if either
 - (i) there is $i \in \mathcal{I}_{\text{HI}}$ such that $x^i(\text{LO}) = c_i(\text{LO})$ and $y^i = 1$, or
 - (ii) there is $i \in \mathcal{I}_{\text{HI}}$ such that $x^i(\text{LO}) > c_i(\text{LO})$;
- (3) If neither of the above holds, then $\text{critState}(s) = \text{UNKNOWN}$.

Now assuming transition (s, a, \hat{s}) satisfies **NC**, we will use monotonicity of $t \mapsto x_t$ and $t \mapsto y_t$, and that s_0 is fixed, to list additional conditions regarding the error flags under which (s, a, \hat{s}) is a valid transition in an exact sense. In what follows, for a state $s = (t, x, y, r)$ and $\ell \in \{\text{LO}, \text{HI}\}$, the statement “an ℓ -criticality job misses its deadline at time t ” is to be understood formally as “there is $i \in \mathcal{I}_\ell$ such that $d_i = t$ and $y^i = 1$ (not finished).”

- E1. ($r = \text{not error}, \hat{r} = \text{potential error}$): If all of the following conditions hold:
- (i) no HI-criticality job misses its deadline at time $\hat{t} = t + 1$,
 - (ii) a LO-criticality job misses its deadline at time \hat{t} , and
 - (iii) the system criticality level is not yet determinable at \hat{t} ; that is, $\text{critState}(\hat{s}) = \text{UNKNOWN}$ ($r = \text{no error}$ says that no HI-criticality jobs missed their deadlines up to time t);
- E2. ($r = \text{not error}, \hat{r} = \text{error}$): Either
- (i) a HI-criticality job misses its deadline at time \hat{t} , or
 - (ii) a LO-criticality job misses its deadline at time \hat{t} and $\text{critState}(\hat{s}) = \text{LO}$;
- E3. ($r = \text{potential error}, \hat{r} = \text{error}$): Same as 2, except that we dispense with the condition in 22ii that a LO-criticality job misses its deadline at time \hat{t} . $r = \text{potential error}$ is saying that no HI-criticality job missed its deadline till t , and the criticality level could not be inferred till t , but a LO-criticality job has missed its deadline already;
- E4. ($r = \text{potential error}, \hat{r} = \text{potential error}$): Same as conditions (i) + (iii) of E1;
- E5. ($r = \text{potential error}, \hat{r} = \text{not error}$): If $\text{critState}(\hat{s}) = \text{HI}$ and no HI-criticality job misses its deadlines at time $\hat{t} = t + 1$;
- E6. ($r = \text{not error}, \hat{r} = \text{not error}$): The combined conditions of ($r = \text{not error}, \hat{r} \neq \text{potential error}$) and ($r = \text{not error}, \hat{r} \neq \text{error}$);
- E7. ($r = \text{error}, \hat{r} = \text{error}'$): always;
- E8. ($r = \text{error}', \hat{r} = \text{error}'$): always.

The following summarizes the control model:

1. At $t = 0$, all jobs are ready to execute and they all demand execution, and the scheduler needs to pick a job to schedule for exactly one time unit before it is invoked again at $t = 1$ (i.e., a_0 needs to be set). Then $y_0 = \mathbf{1}$, $x_0 = \mathbf{0}$, and $r_0 = \text{no error}$;
2. At the beginning of the t th control interval:
 - 2.1 **Update** the cumulative system allocation by setting $x_t \leftarrow x_{t-1} + a_{t-1}$;
 - 2.2 **Observe** (acquire) y_t ;
 - 2.3 **Set Error**: Set r_t given r_{t-1} according to one of E1–E8;
 - 2.4 **Act**: Set a_t to one of the vectors in $A(s_t)$.

We will say that a state is **valid** if it can be generated by the control model above. The state space \mathbf{S} contains only the valid states; i.e., \mathbf{S} is the subset of $\{0, \dots, N\} \times \mathbf{X} \times \mathbf{Y} \times \mathbf{R}$ that can be generated by the control model. For instance, if $s = (t, x, y, r)$ is such that $t = d_i + 1$ and $y^i = 1$ (not finished) for some $i \in [n]$, and $r = \text{not error}$, then for no x is s valid, even if x is such that $\sum_{j=1}^n x^j = t$ (necessary for a state to be valid) and $x^j < c_j(\chi_j)$ for all j .

We point out that the state transition diagram has the simple structure of a directed tree of depth at most N (the maximum horizon length), with fixed root s_0 , and each node in level t , $t \in \{0, \dots, N\}$, corresponds to a possible state at time t (i.e., s_t). Each intermediate node (state) has at most $|\mathbf{A}||\mathbf{Y}| = 2n$ children, each corresponding to a unique current action and next finish signal pair (a_t, y_{t+1}) . Note that the next cumulative allocation vector, x_{t+1} , and the next error flag, r_{t+1} , are deterministic once we know a_t and y_{t+1} , so there is only one choice for each given s_t and a_t .

3.2 The Transition Probabilities

We describe the evolution of the system state by a transition kernel (transition matrix) $Q(d\hat{s}|s, a) : 2^{\mathbf{S}} \times (\mathbf{S} \times \mathbf{A}) \rightarrow [0, 1]$. Since our state space \mathbf{S} is finite, transition kernel Q should satisfy the following for *fixed* action a and previous state s ,

- Q1. $Q(\emptyset|s, a) = 0$;
- Q2. $Q(\mathbf{S}|s, a) = 1$;

Q3. $Q(U|s, a) = \sum_{\hat{s} \in U} Q(\{\hat{s}\}|s, a)$ for every $U \subset S$;

Q4. $0 \leq Q(U|s, a) \leq Q(V|s, a) \leq 1$ for every $U \subset V \subset S$.

We shall abuse notation and write $Q(\hat{s}|s, a)$ for $Q(\{\hat{s}\}|s, a)$. Let (s, a, \hat{s}) be a valid transition. If $s = (t, y, x, r)$, then $\hat{t} = t + 1$, and we shall use the more time-suggestive notation $\hat{s} \equiv s_{t+1}$, where $\hat{y} \equiv y_{t+1}$, $\hat{x} \equiv x_{t+1}$ and $\hat{r} \equiv r_{t+1}$, and we similarly denote s as s_t . If (s, a, \hat{s}) is not valid, then $Q(\hat{s}|s, a) = 0$. Fix an action $a_t = e_i$. Then for transition (s_t, e_i, s_{t+1}) to be valid, we must have $y_t^i = 1$ and $x_{t+1}^i = x_t^i + 1$. Also, scheduling J_i does not affect the execution time demands of the other jobs, so s_{t+1} must satisfy $y_t^j = y_{t+1}^j$ for every $j \neq i$. For our fixed state-action pair (s_t, e_i) , we know at time t that $Z_i > x_t^i$, and for $j \neq i$, $Z_j \in C_j$ for some $C_j \subset [c_j(\chi_j)]$. The following is a complete list of all the possible next states s_{t+1} and the corresponding transition probabilities for the fixed action-state pair $(s_t, a_t = e_i)$ (it is here where we fully utilize the assumption of independent job demands):

- $y_{t+1}^i = 1$ (not finished): This says that the scenario ω is such that $Z_i(\omega) > x_{t+1}^i = x_t^i + 1$, and since $Z_j(\omega)$ remains in C_j for every $j \neq i$ at time $t + 1$, we have

$$\begin{aligned} Q(s_{t+1}|s_t, e_i) &= \mathbb{P}(Z_i > x_t^i + 1, Z_j \in C_j \ \forall j \neq i \mid Z_i > x_t^i, Z_j \in C_j \ \forall j \neq i) \\ &= \frac{\mathbb{P}(Z_i > x_t^i + 1, Z_j \in C_j \ \forall j \neq i)}{\mathbb{P}(Z_i > x_t^i, Z_j \in C_j \ \forall j \neq i)} \\ &= \frac{\mathbb{P}(Z_i > x_t^i + 1)\mathbb{P}(Z_j \in C_j \ \forall j \neq i)}{\mathbb{P}(Z_i > x_t^i)\mathbb{P}(Z_j \in C_j \ \forall j \neq i)} \\ &= \frac{\mathbb{P}(Z_i > x_t^i + 1)}{\mathbb{P}(Z_i \geq x_t^i + 1)} \end{aligned} \quad (5)$$

if $x_t^i < c_i(\chi_i) - 1$, and $Q(s_{t+1}|s_t, e_i) = 0$ otherwise. The second to last equality follows by independence of job demands.

- $y_{t+1}^i = 0$ (finished): Here the demand of job J_i is realized at time $t + 1$; that is, we know that the scenario ω is such that $Z_i(\omega) = x_{t+1}^i = x_t^i + 1$. Using the same reasoning as in the previous case,

$$Q(s_{t+1}|s_t, e_i) = \begin{cases} \frac{\mathbb{P}(Z_i = x_t^i + 1)}{\mathbb{P}(Z_i \geq x_t^i + 1)} & \text{if } x_t^i < c_i(\chi_i) - 1, \\ 1 & \text{if } x_t^i = c_i(\chi_i) - 1 \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Then for fixed $(s_t, a_t = e_i)$, summing over all possible next states; i.e., adding (5) and (6), we have

$$\frac{\mathbb{P}(Z_i > x_t^i + 1) + \mathbb{P}(Z_i = x_t^i + 1)}{\mathbb{P}(Z_i \geq x_t^i + 1)} = \frac{\mathbb{P}(Z_i \geq x_t^i + 1)}{\mathbb{P}(Z_i \geq x_t^i + 1)} = 1.$$

That is, our prescribed transition kernel Q satisfies property Q2, and indeed all the others.

3.3 The Underlying Probability Space

In this section we will outline in detail the construction of the probability space that we will be working with. We will shift our attention from scenario spaces (the Ω_i s, section 2) to *trajectory spaces*, which consist of the sample paths induced by executing policies. We will need this construction when stating the formal definition of our problem, and we shall make several references to it. Readers acquainted with the theory of Markov decision processes may only wish to familiarize themselves with our notation.

Consider the product space

$$(\mathbf{S} \times \mathbf{A})^\infty = \prod_{t=0}^{\infty} (\mathbf{S}_t \times \mathbf{A}_t),$$

where $\mathbf{S}_0 = \{s_0\}$, $s_0 \equiv (t = 0, x_0 = \mathbf{0}, y_0 = \mathbf{1}, r_0 = \text{not error})$ is our *fixed* initial state (all jobs are allocated 0 execution time and they all require execution,) $\mathbf{S}_t \subset \{t\} \times \mathbf{Y} \times \mathbf{X} \times \mathbf{R}$, and \mathbf{A}_t is a copy of \mathbf{A} . We will consider the subset of $(\mathbf{S} \times \mathbf{A})^\infty$ where each sequence of state-action pairs can be generated by our control model, and we will call such sequences the **valid trajectories**. We denote the set of valid trajectories as H_∞ , and we call H_∞ the **trajectory space** induced by all work-conserving scheduling policies. Every $h \in H_\infty$ is a trajectory induced by executing some work-conserving policy, and is of the form

$$h = (s_0, a_0, s_1, a_1, \dots).$$

That is, every h is a realization of a schedule. We endow H_∞ with the product σ -algebra, which we denote as \mathcal{H}_∞ . Let $S_t : H_\infty \rightarrow \mathbf{S}$ be the projection (coordinate) map on H_∞ such that $S_t(h) = s_t$, $h \in H_\infty$. Define $A_t : H_\infty \rightarrow \mathbf{A}(s_t)$ similarly. Given policy $\pi = \{\pi_t : t \in \mathbb{Z}_+\}$, transition kernel Q , and initial distribution ν on \mathbf{S} , the Ionescu-Tulcea extension theorem ([32], Theorem 14.32; [5], Theorem 2.7.2) asserts that there exists a *unique* probability measure \mathbb{P}_ν^π on H_∞ such that

$$\mathbb{P}_\nu^\pi(S_t \in U \mid h_{t-1}, a_{t-1}) = Q(U \mid s_{t-1}, a_{t-1}) \quad (U \subset \mathbf{S}).$$

We have $\nu = \delta_{s_0}$ for $s_0 = (0, \mathbf{0}, \mathbf{1}, \text{not error})$, where δ_{s_0} is Dirac measure on H_∞ , so for brevity we write $\mathbb{P}^\pi \equiv \mathbb{P}_{\delta_{s_0}}^\pi$. We denote expectation with respect to \mathbb{P}^π (on H_∞) as \mathbb{E}^π . Moreover, because every policy is Markov as mentioned above, it follows that the induced state process $\{S_t : t \in \mathbb{Z}_+\}$ is a Markov chain for every policy π . That is, for every $U \subset \mathbf{S}$ and $t \in \mathbb{Z}_+$,

$$\mathbb{P}^\pi(S_{t+1} \in U \mid s_t, \dots, s_0) = \mathbb{P}^\pi(S_{t+1} \in U \mid s_t) = Q(U \mid s_t, \pi_t),$$

where for fixed s_t ,

$$Q(U \mid s_t, \pi_t) = \int_{\mathbf{A}} Q(U \mid s_t, a_t) \pi_t(da_t \mid s_t) = \sum_{i=1}^n Q(U \mid s_t, e_i) \pi_t(e_i \mid s_t).$$

From now on, all subsequent random variables will be defined on H_∞ .

► **Remark.** For a given s_t , each action random variable A_t of the induced action process $\{A_t : t \in \mathbb{Z}_+\}$ is distributed according to $\pi_t(\cdot \mid s_t)$; that is, $\mathbb{P}^\pi(A_t \in C \mid S_t = s_t) = \pi_t(C \mid s_t)$ for every $C \subset \mathbf{A}(s_t)$.

3.4 Problem Statement

We start by formally defining the random variables that make up our objective function and constraints, in terms of the induced MDP. First, we note that to every trajectory corresponds at least one scenario in Ω , and that all scenarios that correspond to a trajectory have the same criticality level. Consequently, we define the criticality level of a trajectory as the criticality level of any scenario corresponding to it³.

³ To ensure that the trajectory criticality level is well-defined, we must assume that every policy assigns every HI-criticality job the processor for at least $c_i(\text{LO})$ time units. However, this is readily satisfied by every policy since, by the way we specified the set of admissible actions $\mathbf{A}(s_t)$, HI-criticality jobs are never dropped.

To this end, let $\mathcal{I}_\ell = \{i \in [n] : \chi_i \geq \ell\}$. Let $h \in H_\infty$ be a trajectory of execution. If h 's criticality level is HI, then the earliest time at which this can be inferred is the first instant at which some job J_i with $\chi_i = \text{HI}$ demands more than $c_i(\text{LO})$ units of execution. We denote this random time as $T_{\text{HI}} : H_\infty \rightarrow \mathbb{N} \cup \{\infty\}$ and, in accordance with our control model, we define it as

$$T_{\text{HI}} = \min\{t \in \mathbb{N} : X_t^i = c_i(\text{LO}) \text{ and } Y_t^i = 1 \text{ (not finished) for some } i \in \mathcal{I}_{\text{HI}}\}, \quad (7)$$

with the usual convention $\min \emptyset = \infty$.

If the trajectory's criticality level is LO, then this can be inferred with certainty only at the first instant at which *all* HI-criticality jobs finish execution. We denote this random time instant as $T_{\text{LO}} : H_\infty \rightarrow \mathbb{N} \cup \{\infty\}$, and we define it as

$$T_{\text{LO}} = \min\{t \in \mathbb{N} : Y_t^i = 0 \text{ (finished) and } X_t^i \leq c_i(\text{LO}) \text{ for all } i \in \mathcal{I}_{\text{HI}}\}. \quad (8)$$

We note that the events $\{T_{\text{LO}} < \infty\}$ and $\{T_{\text{HI}} < \infty\}$ are mutually exclusive, since a trajectory of execution cannot be both LO and HI criticality. This makes the events $\{T_{\text{LO}} = \infty\}$ and $\{T_{\text{HI}} = \infty\}$ mutually exclusive, since every trajectory *must* have a criticality. There are therefore exactly two possibilities, and one of them must occur: Either $\{T_{\text{LO}} < \infty\}$ and $\{T_{\text{HI}} = \infty\}$, or $\{T_{\text{LO}} = \infty\}$ and $\{T_{\text{HI}} < \infty\}$, exclusively. Accordingly, we define the T_{CI} of a trajectory under policy π as the random (finite) time

$$T_{\text{CI}} = \min(T_{\text{HI}}, T_{\text{LO}}).$$

Recalling the manner in which we specified the set of admissible actions $\mathbf{A}(s_t)$, every policy drops LO-criticality jobs as soon as the scenario's criticality level is realized as HI; moreover, only allocations made to LO-criticality jobs *prior to the* T_{CI} are considered as WTF, and this allocation is regarded as WTF only if the system criticality level realization is HI. Let $\text{critPath}(h)$ denote criticality level realization of trajectory (path) $h \in H_\infty$; $\text{critPath} : H_\infty \rightarrow \{\text{LO}, \text{HI}\}$. Then one way to define the WTF of a trajectory $h \in H_\infty$ is as the random variable

$$W(h) = \mathbb{1}_{\{\text{critPath}=\text{HI}\}}(h) \sum_{i:\chi_i=\text{LO}} X_{T_{\text{CI}}}^i(h), \quad (9)$$

where $X_{T_{\text{CI}}}^i(h) \equiv X_{T_{\text{CI}}(h)}^i(h)$ is job J_i 's total allocation sampled at T_{CI} . That is, for valid trajectory $h \in H_\infty$, $W(h) = 0$ if $\text{critPath}(h) = \text{LO}$, and $W(h)$ is equal to the total allocation given to the LO-criticality jobs up to the T_{CI} if $\text{critPath}(h) = \text{HI}$.

We define the **finish time** of job J_i with respect to policy π as the stopping time $F_i : H_\infty \rightarrow \mathbb{N}$, where

$$F_i = \min\{t \in \mathbb{N} : Y_t^i = 0 \text{ (finished)}\}.$$

Objective: A dual-criticality probabilistic MC (pMC) instance I is described by the tuple $(\{J_1, \dots, J_n\}, \varepsilon_{\text{LO}}, \varepsilon_{\text{HI}})$, where for every $i \in [n]$, J_i is specified by the tuple $((\Omega_i, \mathcal{M}_i, \mathbb{P}_i), \zeta_i, \chi_i, c_i, d_i)$. The error parameters ε_{LO} and ε_{HI} are the desired upper bounds on the deadline miss probabilities, and both are in the interval $[0, 1]$. We seek a scheduling policy π such that the expected WTF, $\mathbb{E}^\pi W$, is minimized, while simultaneously guaranteeing probabilistic MC-feasibility in the following sense: (1) Conditioned on $\text{critPath} = \text{LO}$, a job, among all n jobs, may miss its deadline with probability at most ε_{LO} , and (2) conditioned on $\text{critPath} = \text{HI}$, a HI-criticality job may miss its deadline with probability at most ε_{HI} .

We write our problem as the following CMDP:

$$\begin{aligned} \text{CMDP : minimize} \quad & \mathbb{E}^\pi W \\ \text{subject to} \quad & \mathbb{P}^\pi(\bigcup_{i \in [n]} \{F_i > d_i\} \mid \text{critPath} = \text{LO}) \leq \varepsilon_{\text{LO}} \\ & \mathbb{P}^\pi(\bigcup_{i \in \mathcal{I}_{\text{HI}}} \{F_i > d_i\} \mid \text{critPath} = \text{HI}) \leq \varepsilon_{\text{HI}}. \end{aligned} \quad (10)$$

We note that for any $\ell \in \{\text{LO}, \text{HI}\}$, $\{\text{critPath} = \ell\} = \{T_{\text{CI}} = T_\ell\} = \{T_\ell < \infty\} \subset H_\infty$, and that $T_{\text{CI}}^\pi < \infty$ \mathbb{P}^π -almost surely for any work conserving policy π .

Having formalized the problem, we are now able to give precise definitions of what it means for an instance with probabilistic information to be feasible in the MC setting.

► **Definition 4** (Probabilistic MC-feasibility). A pMC instance I is probably MC-feasible (pMC-feasible) if there is a policy π such that the constraints (10) are satisfied.

A scheduling policy under which pMC instance I is pMC-feasible is said to **correctly pMC-schedule** I . Let $\Pi(I)$ denote the set of scheduling policies that correctly pMC-schedule instance I .

► **Definition 5** (Expected WTF-Optimality). A scheduling policy π is said to be WTF-optimal for pMC instance I in expectation (or expected-WTF-optimal) if $\pi \in \Pi(I)$ and $\mathbb{E}^\pi W \leq \mathbb{E}^{\pi'} W$ for all $\pi' \in \Pi(I)$.

Accordingly, given pMC instance I , our goal is to compute an expected WTF-optimal scheduling policy for I .

► **Remark.** For $\ell \in \{\text{LO}, \text{HI}\}$, $\mathbb{P}^\pi(\cdot \mid \text{critPath} = \ell)$ is a probability measure on the restriction of H_∞ to trajectories of criticality ℓ . In the special case where H_∞ is finite and $\mathbb{P}^\pi(\cdot \mid \text{critPath} = \ell)$ is uniform measure, the constraint $\mathbb{P}^\pi(F_i > d_i \text{ for some } i \in [n] \mid \text{critPath} = \ell) \leq \varepsilon_\ell$ has the following simple interpretation: If we let $H_\infty(\ell)$ be the subset of H_∞ consisting of the trajectories whose criticality is ℓ , then the number of trajectories in $H_\infty(\ell)$ where all n jobs do not miss their deadlines is required to be at least $(1 - \varepsilon_\ell)|H_\infty(\ell)|$.

4 Solution Approach: Risk-Constrained MDP

The WTF as defined in (9) depends on the whole trajectory, so in its current form is not suitable in the MDP framework, where costs are accrued *per stage*. We wish to define the WTF as a sum, over the horizon, of functions that depend at each $t \in \mathbb{Z}_+$ on s_{t-1} and s_t only. To this end, we will use the following

► **Proposition 6.** Let $h = (s_0, a_0, s_1, a_1)$ be a valid trajectory (in H_∞). Then

- (a) If $\text{critState}(s_t) = \text{UNKNOWN}$ for some $t \in \mathbb{N}$, then $\text{critState}(s_m) = \text{UNKNOWN}$ for every $m < t$; and
- (b) If $\text{critState}(s_t) = \ell$ for some $t \in \mathbb{N}$ and $\ell \in \{\text{LO}, \text{HI}\}$, then $\text{critState}(s_{m'}) = \ell$ for every $m' > t$.

Proof. Follows readily from monotonicity of $t \mapsto X_t^i(h)$ and $t \mapsto Y_t^i(h)$ for every $i \in [n]$ and $h \in H_\infty$, together with the fact that $x_0 = \mathbf{0}$ and $y_0 = \mathbf{1}$, and that we consider only work conserving policies. ◀

We define the local (per stage) objective cost function $w : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{Z}_+$, where

$$w(s, \hat{s}) = \mathbb{1}\{\text{critState}(s) = \text{UNKNOWN}\} \mathbb{1}\{\text{critState}(\hat{s}) = \text{HI}\} \sum_{i: \chi_i = \text{LO}} \hat{x}^i$$

for $s, \hat{s} \in \mathcal{S}$. That is, $w(s, \hat{s})$ is equal to the total LO criticality allocation $\sum_{i: \chi_i = \text{LO}} \hat{x}^i$ only if the criticality level realization is inferred as a consequence of moving from state s to state \hat{s} and is HI criticality; otherwise, $w(s, \hat{s}) = 0$. Since we are working with the set valid trajectories exclusively, we may use Proposition 6 to write the WTF as

$$W(h) = \sum_{t=0}^{N-1} w(S_t(h), S_{t+1}(h)).$$

4.1 The Risk Constraints

We will leverage the ideas of Geibel and Wysotzki [21] to carry out the transformation of the deadline miss probabilities into a single “risk” constraint that takes the form of the expectation of immediate costs.

We may write the first constraint in **CMDP** as

$$\mathbb{P}^\pi(F_i > d_i \text{ for some } i \in [n], \text{critPath} = \text{Lo}) \leq \varepsilon_{\text{Lo}} \mathbb{P}^\pi(\text{critPath} = \text{Lo}), \quad (11)$$

where

$$\mathbb{P}^\pi(F_i > d_i \text{ for some } i \in [n], \text{critPath} = \text{Lo}) = \mathbb{P}^\pi(\{h \in H_\infty(\text{Lo}) : \exists t \in \mathbb{N} \text{ s.t. } r_t = \text{error}\}). \quad (12)$$

Following Geibel and Wysotzki [21], constraint (11) defines the **risk** across the LO-criticality trajectories associated with executing policy π . Similarly to the LO-criticality case, we write the second constraint in **CMDP** as

$$\mathbb{P}^\pi(F_i > d_i \text{ for some } i \in \mathcal{I}_{\text{Hi}}, \text{critPath} = \text{Hi}) \leq \varepsilon_{\text{Hi}} \mathbb{P}^\pi(\text{critPath} = \text{Hi}), \quad (13)$$

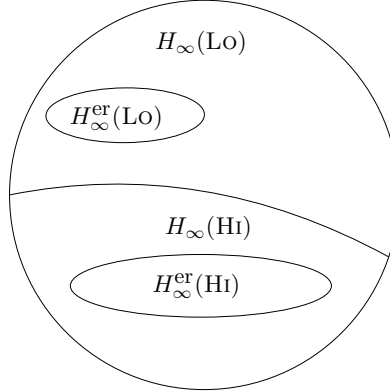
where

$$\mathbb{P}^\pi(F_i > d_i \text{ for some } i \in \mathcal{I}_{\text{Hi}}, \text{critPath} = \text{Hi}) = \mathbb{P}^\pi(\{h \in H_\infty(\text{Hi}) : \exists t \in \mathbb{N} \text{ s.t. } r_t = \text{error}\}). \quad (14)$$

The trajectories

$$H_\infty^{\text{er}} \equiv \{h = (s_0, a_1, s_1, \dots) \in H_\infty : \exists t \in \mathbb{N} \text{ s.t. } r_t = \text{error}\}$$

are the **error trajectories** that we want to avoid with high probabilities.



■ **Figure 2** Every policy π induces a probability measure \mathbb{P}^π on the trajectory space H_∞ , where the latter consists of two disjoint sets: The LO and HI-criticality trajectories $H_\infty(\text{Lo})$ and $H_\infty(\text{Hi})$, respectively. The ovals are graphical representations of the error trajectory sets that we wish to avoid. Given a policy, say π , the “sizes” of the induced error sets $H_\infty^{\text{er}}(\text{Lo})$ and $H_\infty^{\text{er}}(\text{Hi})$ (relative to the entire trajectory space H_∞) are given by the unique probability measure \mathbb{P}^π . The smaller the size a policy assigns to the error subsets, the better it is at avoiding trajectories in them. Roughly speaking, each ℓ th (criticality-specific) constraint in **ECMDP** is a restriction on the “size” of the corresponding error set $H_\infty^{\text{er}}(\ell)$ *relative to the trajectories of the same criticality*; i.e., relative to the size of $H_\infty(\ell)$ (and not to the whole trajectory space H_∞); hence the conditioning in the constraints.

We now combine the constraints in **CMDP**. Put $p_\ell = \varepsilon_\ell \mathbb{P}(\text{crit} = \ell)$, $\ell \in \{\text{Lo}, \text{Hi}\}$, and let

$$\Delta = \min(p_{\text{Lo}}, p_{\text{Hi}}).$$

Since H_∞ is the disjoint union of $H_\infty(\text{Lo})$ and $H_\infty(\text{Hi})$, we may combine (12) and (14), and require the satisfaction of the more conservative risk

$$\mathbb{P}^\pi(H_\infty^{\text{er}}) = \mathbb{P}^\pi(\{h \in H_\infty : h \text{ is error}\}) = \mathbb{P}^\pi(\exists t : R_t = \text{error}) \leq \Delta. \quad (15)$$

For then,

$$\mathbb{P}^\pi(H_\infty^{\pi, \text{er}}(\ell)) \leq \mathbb{P}^\pi(H_\infty^{\text{er}}) \leq \Delta = \min(p_{\text{Lo}}, p_{\text{Hi}}) \leq p_\ell, \quad \text{for all } \ell \in \{\text{Lo}, \text{Hi}\}. \quad (16)$$

Next we write the risk constraint (15) as an expectation under \mathbb{E}^π of immediate costs having a form similar to κ . Inspired by Geibel and Wysotzki [21], we define the per-stage *constraint cost* function $\kappa : \mathbb{K} \rightarrow \{0, 1\}$ as

$$\kappa(s, a) \equiv \kappa(s) = \begin{cases} 1 & \text{if } r = \text{error}, \\ 0 & \text{otherwise.} \end{cases} \quad (17)$$

This way, if a trajectory $h = (s_0, a_0, s_1, a_1, \dots)$ is error, then since there is $t \in \mathbb{N}$ such that $r_t = \text{error}$ and $r_{t'} = \text{error}'$ for all $t' > t$, it follows that the sequence of constraint costs corresponding to this trajectory is such that

$$\kappa(s_0) = 0, \dots, \kappa(s_{t-1}) = 0, \underbrace{\kappa(s_t) = 1}_{\text{error}}, \kappa(s_{t+1}) = 0, \dots, \kappa(s_N) = 0.$$

If h is not error, then $\kappa(s_t) = 0$ for all $t \in \mathbb{Z}_+$. If we let $C = \sum_{t=0}^N \kappa(S_t)$, then $C \in \{0, 1\}$; that is, C is a Bernoulli random variable with probability of success $\mathbb{P}^\pi(C = 1)$. Success of this Bernoulli trial happens if there is $t \in \mathbb{N}$ such that $R_t = \text{error}$. That is,

$$\{C = 1\} = \{\exists t : R_t = \text{error}\},$$

from which it follows that

$$\mathbb{P}^\pi(C = 1) = \mathbb{P}^\pi(\exists t : R_t = \text{error}).$$

Since C is Bernoulli, it follows that

$$\mathbb{E}^\pi C = \mathbb{P}^\pi(C = 1),$$

from which we may write the risk constraint as

$$\mathbb{P}^\pi(\exists t : R_t = \text{error}) = \mathbb{P}^\pi(C = 1) = \mathbb{E}^\pi C = \mathbb{E}^\pi \sum_{t=0}^N \kappa(S_t) \leq \Delta.$$

As a result of this transformation, all costs are now expectations of immediate costs, and **CMDP** has the form

$$\begin{aligned} \mathbf{ECMDP} : \quad & \underset{\pi \in \Pi}{\text{minimize}} \quad \mathbb{E}^\pi W = \mathbb{E}^\pi \sum_{t=0}^{N-1} w(S_t, S_{t+1}) \\ & \text{subject to} \quad \mathbb{E}^\pi C = \mathbb{E}^\pi \sum_{t=0}^N \kappa(S_t) \leq \Delta. \end{aligned} \quad (18)$$

We denote as V^* the optimal value of **ECMDP**, where

$$V^* = \inf_{\pi \in \Pi} V(s_0, \pi), \quad V(s_0, \pi) = \mathbb{E}^\pi W,$$

subject to the constraint $\mathbb{E}^\pi C \leq \Delta$.

4.2 The Linear Programming Approach

Most of the results that we apply in what follows regarding LP formulations for constrained MDPs are due to Kallenberg [27] and Altman [3]. These formulations are also discussed in several other references [4, 28, 29].

We define the immediate objective cost of taking action a_t in state s_t as the expected cost over all possible next states:

$$w(s_t, a_t) = \mathbb{E}^\pi w(s_t, a_t, S_{t+1}) = \sum_{s_{t+1} \in \mathcal{S}} w(s_t, a_t, s_{t+1}) Q(s_{t+1} | s_t, a_t) = \sum_{s_{t+1} \in \mathcal{S}} w(s_t, s_{t+1}) Q(s_{t+1} | s_t, a_t)$$

(see Puterman [35] equation (2.1.1).) To this end, recall that the schedule concludes when all jobs finish execution. Moreover, costs (both objective and constraint) is (possibly) incurred only until all jobs finish execution. That is, from the costs' perspective, we are concerned with the set of states

$$\mathcal{S}' = \{s = (t, x, y, r) \in \mathcal{S} : y^i = 1 \text{ (not finished) for some } i \in [n]\}.$$

Costs keep (possibly) accruing until the state process hits $\mathcal{S} \setminus \mathcal{S}'$. Moreover, the set $U \equiv \mathcal{S} \setminus \mathcal{S}'$ is always reached under any work-conserving policy in finite time that is bounded above by $\max_{i \in [n]} \{F_i\} \leq N$. Since we are considering work-conserving policies only, \mathcal{S}' also excludes any states $s = (t, x, y, r)$ for which $t = N$. Once the state process hits the set U , it never departs U ; that is, U is absorbing under any work-conserving policy. In this case, our MDP is called **\mathcal{S}' -transient** (Altman [3]). In fact, our MDP is in a more restricted class that is a subset of \mathcal{S}' -transient MDPs. If we let T_U be the hitting time of set U ; i.e.,

$$T_U = \inf\{t \in \mathbb{N} : S_t \in U\},$$

then $\mathbb{E}^\pi T_U \leq \mathbb{E}^\pi \max_{i \in [n]} \{F_i\} \leq N < \infty$ for any work-conserving policy π , and our MDP is said to be **\mathcal{S}' -absorbing**, or absorbing to U .

An optimal policy can be derived by solving the following linear program (Altman [4], equation (8.18)):

$$\begin{aligned} \mathbf{LP} : \text{minimize} \quad & \left[\sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}(s)} w(s, a) \rho(s, a) = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}(s)} \rho(s, a) \sum_{\hat{s} \in \mathcal{S}} w(s, \hat{s}) Q(\hat{s} | s, a) \right] \\ \text{subject to} \quad & \sum_{s \in \mathcal{S}} \kappa(s) \sum_{a \in \mathcal{A}(s)} \rho(s, a) \leq \Delta \\ & \sum_{a \in \mathcal{A}(s)} \rho(s, a) - \sum_{s' \in \mathcal{S}'} \sum_{a \in \mathcal{A}(s')} \rho(s', a) Q(s | s', a) = \delta_{s_0}(s) \quad \forall s \in \mathcal{S} \quad (*) \\ & \rho(s, a) \geq 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s). \end{aligned}$$

An equivalent formulation is also given by Kallenberg [27], Theorem 6, and Kallenberg [29], equation (57) and Theorem 26, where time is explicit. We note that in constraint (*) of **LP**, for every $s \in \mathcal{S}$, the second summation is taken only over the states in \mathcal{S}' that are *predecessors* to state s ; that is, over $s' \in \mathcal{S}'$ for which there is $a \in \mathcal{A}(s')$ such that $Q(s | s', a) > 0$. For instance, if the state is $s = (t, x, y, r)$, then $s' = (t', x', y', r') \in \mathcal{S}'$ is a predecessor of s if $t' = t - 1$.

If we let $M = \sum_{s \in \mathcal{S}} |\mathcal{A}(s)|$, then the decision variables involved in **LP** are $(\rho(s, a) : s \in \mathcal{S}, a \in \mathcal{A}(s)) \in [0, \infty)^M$. If we let $\mathbb{K} = \{(s, a) : s \in \mathcal{S}, a \in \mathcal{A}(s)\}$ be the set of **admissible state-action pairs**, then the vectors ρ over which the optimization in **LP** is carried out are non-negative finite measures on $(\mathbb{K}, 2^{\mathbb{K}})$.

By Altman [4] Theorem 8.2, an optimal policy is the following: When the state is s , if $\rho(s, A(s)) > 0$, then π chooses action $a \in A(s)$ with probability

$$\pi(a|s) = \frac{\rho(s, a)}{\rho(s, A(s))} = \frac{\rho(s, a)}{\sum_{a' \in A(s)} \rho(s, a')}, \quad (19)$$

and otherwise chooses a arbitrarily from $A(s)$.

► **Remark.** For $(s_t, a_t) \in \mathbb{K}$, $\rho(s_t, a_t)$ has the interpretation of being the probability that both state s_t is occupied and action $a_t \in A(s_t)$ is taken at time t under policy π defined by (19). In fact,

$$\rho(s_t, a_t) = \mathbb{E}^\pi \sum_{q=1}^N \mathbb{1}\{S_q = s_t, A_q = a_t\}, \quad (20)$$

where $\mathbb{1}\{S_q = s_t, A_q = a_t\} \equiv \mathbb{1}\{S_q = s_t, A_q = a_t\}(h)$ is the indicator function that evaluates to 1 if the trajectory of execution h is such that both $S_q(h) = s_t$ and $A_q(h) = a_t$, and to 0 otherwise. The RHS of (20) is the expected number of times that state-action pair (s_t, a_t) is visited under policy π , and is termed the state-action *occupation measure* (or visitation frequency) associated with policy π . The summation in (20) reduces to $\mathbb{1}\{S_t = s_t, A_t = a_t\}$, from which it follows that

$$\rho(s_t, a_t) = \mathbb{P}^\pi(S_t = s_t, A_t = a_t).$$

► **Remark.** Although we embedded time in the state to leverage results for computing stationary policies, the policy defined by transformation (19) is time-dependent (non-stationary); it is Markov nonetheless. However, we do not need to include time in the state; this is because all jobs start at time 0, the schedule concludes when all jobs finish execution, and we consider only work-conserving policies, so for a valid triple (x, y, r) , $\sum_{i=1}^n x_i$ maps to the unique time instant during the schedule when the state is occupied. All the previous discussion can be modified so that $s = (x, y, r)$, and implicitly using $t = \sum_{i=1}^n x_i$. We chose to include t explicitly in the state to make the exposition clearer.

Our main result follows from the previous discussion, and is contained in

► **Theorem 7.** *Given a pMC instance I , if \mathbf{LP} is feasible, then I is pMC-feasible. Moreover, if I is pMC-feasible, then the policy given by transformation (19) is expected-WTF-optimal for I .*

4.3 A Less Pessimistic Exact Formulation

Recall that we combined the constraints of **CMDP** into the single risk constraint (15). The combined constraint is more conservative than the original constraints, and it certainly restricts the feasible region of **CMDP**, in the sense that there might be a policy that is feasible for the original problem but not for the one with combined constraints. We did so to simplify the exposition, so as to have a CMDP with a single constraint (and a single cost random variable). Now we detail how to handle the exact case. This will be done at the expense of a constant increase in the size of the state space, and slightly more complicated transition dynamics.

We distinguish two *error types*: LO-errors and HI-errors. The LO-error trajectories are described by the event

$$H_\infty^{\text{er}}(\text{LO}) \equiv \{h \in H_\infty : F_i(h) > d_i \text{ for some } i \in [n]\} \cap \{\text{critPath} = \text{LO}\},$$

whereas the HI-error trajectories are precisely

$$H_\infty^{\text{er}}(\text{HI}) \equiv \{h \in H_\infty : F_i(h) > d_i \text{ for some } i \in \mathcal{I}_{\text{HI}}\} \cap \{\text{critPath} = \text{HI}\}.$$

Thus, if a HI-criticality job misses its deadline under a policy but the trajectory is LO criticality, then this is a LO-error. On the other hand, if a LO-criticality job misses its deadlines but the trajectory is HI criticality, then this is not an error.

We enlarge the set of error flags by adding criticality-specific ones:

$$R = \{\text{not error}, \text{potential error}, \text{error}, \underbrace{\text{error}_{\text{LO}}, \text{error}_{\text{HI}}}_{\text{error}'}, \text{error}'\}.$$

For $h = (s_0, a_0, s_1, \dots) \in H_\infty$ and $\ell \in \{\text{LO}, \text{HI}\}$, $h \in H_\infty^{\text{er}}(\ell)$ iff there is $t \in \mathbb{N}$ such that $r_t = \text{error}_\ell$. Here, the flag **error** no longer identifies an error trajectory, but $r = \text{error}$ means that a HI-criticality jobs missed its deadline but the system criticality level realization is not yet determinable. If $r_t = \text{error}$ for some t , then an error will happen at some $t' > t$ when the criticality level of the trajectory becomes known, at which point the type of the error will be determined; therefore, it is always the case that $r_{t'} = \text{error}_\ell$ for some ℓ and all $t' > t$. The flags error_ℓ communicate the following information: They indicate the occurrence of an error and the *type* of the error; that is, if $r_t = \text{error}_\ell$ for some $t \in \mathbb{N}$ and ℓ , then the trajectory is an error and it is a type ℓ error (so it is also saying that the system criticality level realization is determined). We will define criticality-specific constraint cost functions, where the ℓ -error cost function charges a unit cost whenever the state's error flag is error_ℓ .

The following is the modification of conditions E1–E8 to accommodate the new setup:

- E1'**. ($r = \text{not error}, \hat{r} = \text{potential error}$): If a LO-criticality job misses its deadline at time $\hat{t} = t + 1$ and no HI-criticality job misses its deadline at time \hat{t} , but $\text{critState}(\hat{s}) = \text{UNKNOWN}$;
- E2'**. ($r = \text{not error}, \hat{r} = \text{error}$): If a HI-criticality job misses its deadline at time $\hat{t} = t + 1$, but $\text{critState}(\hat{s}) = \text{UNKNOWN}$;
- E3'**. ($r = \text{not error}, \hat{r} = \text{error}_{\text{LO}}$), ($r = \text{potential error}, \hat{r} = \text{error}_{\text{LO}}$): If $\text{critState}(\hat{s}) = \text{LO}$ and either
 1. a LO-criticality job misses its deadline at time \hat{t} but no HI-criticality jobs miss their deadlines at \hat{t} , or
 2. a HI-criticality job misses its deadline at time \hat{t} ;
- E4'**. ($r = \text{not error}, \hat{r} = \text{error}_{\text{HI}}$), ($r = \text{potential error}, \hat{r} = \text{error}_{\text{HI}}$): If a HI-criticality job misses its deadline at time \hat{t} and $\text{critState}(\hat{s}) = \text{HI}$;
- E5'**. ($r = \text{potential error}, \hat{r} = \text{error}$): Same as E2';
- E6'**. ($r = \text{error}, \hat{r} = \text{error}_\ell$) for any ℓ : If $\text{critState}(\hat{s}) = \ell$;
- E7'**. ($r = \text{potential error}, \hat{r} = \text{potential error}$): Same as before;
- E8'**. ($r = \text{potential error}, \hat{r} = \text{not error}$): Same as before;
- E9'**. ($r = \text{not error}, \hat{r} = \text{not error}$): Same as before;
- E10'**. ($r = \text{error}_\ell, \hat{r} = \text{error}'$) for any ℓ : always;
- E11'**. ($r = \text{error}', \hat{r} = \text{error}'$): always.

The control model is modified so that the step 2.3 that sets the error flags uses rules E1'–E11' instead.

Now we define two immediate constraint-cost functions $\kappa_{\text{LO}}, \kappa_{\text{HI}} : S \rightarrow \{0, 1\}$, where $\kappa_\ell(s) = \mathbb{1}\{r = \text{error}_\ell\}$, $\ell \in \{\text{LO}, \text{HI}\}$. If $h = (s_0, a_0, s_1, \dots)$ is ℓ -error, then there is exactly one $t \in \mathbb{N}$ such that $r_t = \text{error}_\ell$, and $r_{t'} = \text{error}'$ for all $t' > t$. For this trajectory and the t in the preceding sentence, $\kappa_\ell(s_t) = 1$, and $\kappa_\ell(s_{t'}) = 0$ for all $t' \neq t$. Therefore, $\sum_{m=0}^\infty \kappa_\ell(s_m) = 1$, and $\sum_{m=0}^\infty \kappa_{\ell'}(s_m) = 0$ for $\ell' \neq \ell$. If h is not error, then $\sum_{m=0}^\infty \kappa_{\text{LO}}(s_m) = \sum_{m=0}^\infty \kappa_{\text{HI}}(s_m) = 0$. If we define the constraint-cost random variables $C_{\text{LO}}, C_{\text{HI}} : H_\infty \rightarrow \mathbb{R}_+$ such that $C_\ell = \sum_{t=0}^\infty \kappa_\ell(s_t)$, $\ell \in \{\text{LO}, \text{HI}\}$, then every C_ℓ is a Bernoulli random variable, with probability of success $\mathbb{P}^\pi(C_\ell = 1) = \mathbb{P}^\pi(\exists t \in \mathbb{N} \text{ such that } R_t = \text{error}_\ell) = \mathbb{P}^\pi(H_\infty^{\text{er}}(\ell))$, where $\mathbb{P}^\pi(C_\ell = 1) = \mathbb{E}^\pi C_\ell$.

Then, we have the following optimization problem:

$$\begin{aligned} \mathbf{ECMDP}' : \underset{\pi \in \Pi}{\text{minimize}} \quad & \mathbb{E}^\pi W = \mathbb{E}^\pi \sum_{t=0}^{N-1} w(S_t, S_{t+1}) \\ \text{subject to} \quad & \mathbb{E}^\pi C_\ell = \mathbb{E}^\pi \sum_{t=0}^N \kappa(S_t) \leq p_\ell, \quad \ell \in \{\text{Lo}, \text{Hi}\}. \end{aligned} \quad (21)$$

Finally, an exact WTF-optimal policy may be derived by solving a modification of **LP**, where the constraint $\sum_{s \in \mathbf{S}} \kappa(s) \sum_{a \in \mathbf{A}(s)} \rho(s, a) \leq \Delta$ is replaced by the criticality-specific cost constraints

$$\sum_{s \in \mathbf{S}} \kappa_{\text{Lo}}(s) \sum_{a \in \mathbf{A}(s)} \rho(s, a) \leq p_{\text{Lo}}, \quad \sum_{s \in \mathbf{S}} \kappa_{\text{Hi}}(s) \sum_{a \in \mathbf{A}(s)} \rho(s, a) \leq p_{\text{Hi}}.$$

Computational Complexity

An optimal satisfying assignment for the variables $\{\rho(s, a) : s \in \mathbf{S}, a \in \mathbf{A}(s)\}$ can be found using any variant of the simplex algorithm, which, in practice, is efficient in the number of decision variables and the number of constraints. However, linear program **LP** can have as many as $n|\mathbf{S}|$ decision variables and $|\mathbf{S}|$ constraints, so it requires explicit enumeration of the state space \mathbf{S} , and therein lies the trouble. If we let $c = \max_{i \in [n]} \{c_i(\chi_i)\}$, then a “very” crude estimate of the size of \mathbf{S} is $2^n 4(c+1)^n$ (since we do not need to include time in the state as mentioned earlier).

An MC instance might look like the following: $n = 10$ jobs and $c = 100$; for this instance, $|\mathbf{S}|$ might be as large as $2^{10} \times 4 \times 101^{10} \approx 10^{23}$, which is astronomical. In a typical MC instance, $c \gg n$, so our estimate of $|\mathbf{S}|$ indicates that the main cause of this “state space explosion” is the state variable x that records the cumulative execution time allocations, and which introduces the factor $(c+1)^n$ into our estimate of $|\mathbf{S}|$.

A Note on Approximation. We point out that despite the high computational complexity, the problem can be approximated efficiently and accurately using the factored MDP representation framework [13]. In it, instead of explicitly enumerating the states, the transition kernel is stored compactly by considering only the variables on which each state variable depends. To make use of the compact state space representation, we may utilize the symmetric primal-dual LP approximation techniques by Dolgov and Durfee [20], in which the variables of both **LP** and its dual are replaced with linear combinations of basis functions that are defined only on subsets of the state space—the so called *features*. Because jobs are independent, each variable (feature) comprising the state space depends only on a small number of variables, and our MDP falls in a special category of MDPs that are amenable to approximation, namely *loosely (weakly) coupled MDPs* [34]. The basic idea is to decompose the MDP into smaller MDPs—which is possible by the independence assumption—whose *exact* solutions can be obtained in manageable time, and then combine the solutions for the subMDPs in a proper way to construct a solution for the global MDP. This, together with the simple additive form of our cost functions, results in a linear program that contains substantially less decision variables and constraints, which can then be solved efficiently to get an approximate policy that is close to optimal.

5 Quantitative Evaluations

The purpose of this section is to provide intuition on how worst-case based approaches may underperform when worst-case demands are not realized, or when errors may be allowed. Despite the sizeable state space, we were able to compare the performance of our approach to the OCBP algorithm [10] on small instances.

OCBP builds offline a fixed priority table, and then schedules jobs according to their computed priorities. At every iteration of the priority computation procedure, OCBP finds the lowest priority job as follows: Job J_i has the lowest priority among the set of jobs that have not been assigned a priority yet if there is $c_i(\chi_i)$ time in $[0, d_i]$ when the other jobs j have executed for their demand at J_i 's criticality; that is, for $c_j(\chi_i)$ each. If a lowest priority job is found at a certain iteration, then it is removed from the set of jobs that have not yet been assigned a priority, and the priority computation procedure is applied to the remaining set. If a total ordering is found on the whole input job set, then the instance is said to be OCBP-schedulable.

In our evaluation, we consider 14 pMC instances, each comprised of 3 or 4 jobs. The instances are handcrafted to showcase different aspects of our approach. For each instance, we generate job J_i 's probability mass functions (pmf) over $\{1, 2, \dots, c_i(\chi_i)\}$, $i \in [n]$, as a uniform vector in the standard $(c_i(\chi_i) - 1)$ -simplex. For this purpose, we use the UUniFast Algorithm [12], which generates uniformly distributed task utilization vectors. By setting the target system utilization of UUniFast to unity, we get the desired pmfs. We generate the state space using a recursive procedure, and we use Gurobi optimization software [24] to solve **LP**. The simulation is written in C, and is publicly available at <https://github.com/RADICAL-UBC/mc-simulation.git>.

For each instance, we simulate job execution on 100,000 demand vectors (behaviors) that are randomly drawn from the job pmfs. Table 3 lists the input job parameters for each instance and the results of job executions. The entry “**# Errors**” is the number of simulation samples—out of 100,000—where an execution error occurred under the policy derived by solving the more conservative formulation **LP**. An error occurs for a sample if either some job misses its deadline and the sample is LO-criticality, or a HI-criticality job misses its deadline and the sample is HI criticality (the definition of error according to the classical job dropping model.) An error is counted only once for a sample if it happens, even if multiple jobs miss their deadlines. The entry “**# Deadline misses**” in Table 3 is the number of samples per job for which the job missed its deadline. Note that, by definition of error, if a job misses its deadline for some sample, then this does not necessarily mean that an error occurred for that sample.

Putting $\varepsilon_\ell = 0$ for any criticality ℓ results in $\Delta = 0$, which means that not a single error is permitted. We observed that if an instance is OCBP-schedulable, then it is pMC-feasible with $\varepsilon_\ell = 0$ for any ℓ . Instances I_1 , I_2 and I_3 are examples of this situation, in which no error happens in any of the simulated demand samples under the policy derived from a solution to **LP**. This is not surprising, however, because worst-case OCBP-schedulability implies worst-case MC-feasibility, so our approach would not make sense if it cannot correctly schedule, with zero errors, pMC instances that are OCBP-schedulable. We note that for I_3 , even though J_3 misses its deadline in some samples, none of those deadline misses are errors.

The remaining instances are all not OCBP-schedulable. The job sets comprising instances I_4 and I_5 are identical, and all jobs in both instances have the same execution time pmfs. Moreover, both instances were simulated over the same execution time samples. These instances show how one can control the desired error by supplying different error parameters. Instance I_4 results in 10,874 error executions (i.e., 10.87% of the samples) when $\varepsilon_{\text{Lo}} = \varepsilon_{\text{Hi}} = 1.0$ (100% allowed error; we do not care about errors,) whereas in instance I_5 , where $\varepsilon_{\text{Lo}} = 0.2$ (20%) and $\varepsilon_{\text{Hi}} = 0.4$ (40%), only 7,921 samples (i.e., 7.92% of the samples) were erroneously scheduled. For the results to make sense, we simulated both I_4 and I_5 on the same demand samples.

Instance I_6 is not even (deterministically) MC-schedulable by the clairvoyant algorithm, so one would not expect this instance to be pMC-feasible for vanishing error parameters. This is indeed the case, and our algorithm reported that I_6 is not pMC-feasible when $\varepsilon_\ell = 0$ for any ℓ . However, when allowing for some (controlled) error, I_6 might become pMC-feasible, and this is the case for I_7 , which is identical to I_6 (including its job execution time distribution,) except that

some error is permitted. Instance I_8 has the same jobs and error parameters as I_7 , except that the job demand distributions are different. Also here we simulated both I_7 and I_8 on the same demand samples.

Instance I_9 is identical to I_8 (including their job demand distributions,) but instance I_9 's error parameters are an order of magnitude less than those of I_8 . The number of errors dropped from 1,862 for I_8 with $\varepsilon_{Lo} = 0.4$ and $\varepsilon_{Hi} = 0.5$ to 890 for the same simulation samples in I_9 with $\varepsilon_{Lo} = \varepsilon_{Hi} = 0.05$. We observe that the reduction in the number of error executions is not linear with respect to the error parameters.

Instance I_{11} is not OCBP-schedulable but it is schedulable by the clairvoyant algorithm, and it turns out that this instance is pMC-feasible when the error parameters are vanishing. There are no error executions out of all the simulated samples. This is one of many MC instances where OCBP overestimates the resources required for correct (deterministic) MC-feasibility, a problem that our approach tackles effectively. Whereas OCBP requires an s -speed processor to schedule this instance for some $s > 1$ (the speed-up factor,) our approach is capable of scheduling this instance on a unit-speed processor without incurring any errors, thus maximizing the processor utilization.

Instance I_{12} and I_{13} are identical (including job demand distributions,) and they are another example where the resulting error executions can be controlled by controlling the error parameters, but for 3 job instances. Finally, instance I_{14} is a pMC-feasible instance whose job execution times are significantly larger than all of the other instances.

In all of the simulated instances, we observed that our algorithm favors Hi criticality jobs; in every simulated instance, the number of Hi-criticality deadline misses is much less than Lo-criticality deadline misses. This can be attributed to the way the set of admissible actions $A(s)$ is prescribed, where Lo-criticality jobs are always dropped whenever the scenario's criticality level is inferred as Hi.

6 Concluding Remarks

We developed a probabilistic framework for reasoning about dual-criticality MC jobs systems when job demand distributions are available. We transformed the problem of constructing optimal scheduling policies into a risk-constrained MDP, where risk is the probability of missing deadlines at the two criticalities, taken over the relevant trajectories induced by the MDP. We solved the constrained MDP using a Linear Programming formulation, and showed how to construct optimal randomized Markov policies from the solution of the Linear Program. We also provided simulation results of our approach on some representative MC instances to verify and sharpen intuition.

We assumed complete knowledge of job demand distributions, and we did not consider the problem of obtaining and estimating those distributions. The latter is an important problem of investigation, and complements the probabilistic framework that we have developed. In the same vein, it is natural to consider variants of our problem where only samples of the demand distributions are available instead of full fledged distributions. This entails that the state transition probabilities are not known a priori, and the problem becomes that of *adaptive control*. If a sampler that can be queried is available, then sample average approximation (SAA) techniques can be utilized to approximate the expectation-based objectives through suitably defined empirical measures. Moreover, learning techniques, such as unsupervised learning (Q-Learning), can be used to estimate the transition probabilities and construct reasonable heuristics efficiently.

Although our approach is optimal in expectation, it is nevertheless computationally expensive, and its practicality is limited to small instances. This is due to the large state space that grows exponentially in the number of input jobs. Our next-step is to investigate provable approximation schemes that trade optimality for efficiency.

■ **Table 3** Simulation instances and job execution results. The execution of each instance is simulated, using the policy generated by solving **LP**, over 100,000 demand samples (vectors)

Instance							OCBP Schedulability	# Deadline Misses	# Errors (Lo + Hi)
Instance	ε_{Lo}	ε_{Hi}	Job	χ	$c(Lo)$	$c(Hi)$	d		
I_1	0.0	0.5	J_1	Hi	70	75	160	0	0
			J_2	Lo	50	50	50	0	
			J_3	Hi	8	20	85	0	
			J_4	Hi	1	15	65	0	
I_2	0.0	0.5	J_1	Lo	100	100	218	0	0
			J_2	Lo	50	50	50	0	
			J_3	Lo	8	8	58	0	
			J_4	Hi	60	61	119	0	
I_3	0.0	0.0	J_1	Lo	60	60	161	0	0
			J_2	Lo	50	50	50	0	
			J_3	Lo	20	20	70	1,447	
			J_4	Hi	30	31	101	0	
I_4	1.0	1.0	J_1	Lo	10	10	140	0	10,874
			J_2	Lo	75	75	75	20,251	
			J_3	Lo	50	50	100	29,931	
			J_4	Hi	5	15	120	0	
I_5	0.2	0.4	J_1	Lo	10	10	140	0	7,921
			J_2	Lo	75	75	75	43,781	
			J_3	Lo	50	50	100	7,109	
			J_4	Hi	5	15	120	0	
I_6	0.0	0.0	J_1	Lo	20	20	70	–	not pMC feasible
			J_2	Lo	30	30	80	–	
			J_3	Hi	27	30	50	–	
			J_4	Hi	8	25	70	–	
I_7	0.4	0.5	J_1	Lo	20	20	70	240	192
			J_2	Lo	30	30	80	1,669	
			J_3	Hi	27	30	50	0	
			J_4	Hi	8	25	70	183	
I_8	0.4	0.5	J_1	Lo	20	20	70	4,640	1,862
			J_2	Lo	30	30	80	10,667	
			J_3	Hi	27	30	50	0	
			J_4	Hi	8	25	70	1,845	
I_9	0.05	0.05	J_1	Lo	20	20	70	5,504	890
			J_2	Lo	30	30	80	9,982	
			J_3	Hi	27	30	50	0	
			J_4	Hi	8	25	70	883	
I_{10}	0.07	0.07	J_1	Hi	20	49	50	0	18
			J_2	Lo	30	30	80	14,822	
			J_3	Hi	1	12	50	0	
			J_4	Hi	5	23	110	13	
I_{11}	0.0	0.0	J_1	Hi	3	10	27	0	0
			J_2	Lo	15	15	17	39,561	
			J_3	Hi	2	5	7	0	
I_{12}	0.05	0.09	J_1	Lo	50	50	90	3,726	2,239
			J_2	Lo	30	30	50	50,563	
			J_3	Hi	19	30	35	0	
I_{13}	0.4	0.5	J_1	Lo	50	50	90	4,493	2,342
			J_2	Lo	30	30	50	33,981	
			J_3	Hi	19	30	35	0	
I_{14}	0.03	0.03	J_1	Hi	49	75	100	0	685
			J_2	Lo	120	120	210	31,883	
			J_3	Hi	125	275	400	588	

We assumed that the given job demands are independent; however, we can drop the independence assumption as long as we are given the joint distribution of job demands. Our framework can handle this with minimal modifications; in particular, the product space construction (that we carried out in section 2) is no longer needed, but one needs to be able to compute the marginal demand distributions in order to compute the MDP's transition probabilities. This, however, places a greater burden on the system designer, because the joint distribution of job demands might be harder to obtain compared to individual job demand distributions. Nevertheless, our framework, as presented, offers great flexibility, because it allows the individual job distributions to be defined on different probability spaces (the Ω_i s) that might correspond to different operating environments and settings.

Finally, two important extensions to our model deserve special mention and are due future studies. The first is the case of arbitrary number of criticality levels, and the second is the sporadic task model.


References

- 1 AdaCore. What is do-178b?, 2014. URL: <http://www.adacore.com/gnatpro-safety-critical/avionics/do178b/>.
- 2 Bader Alahmad, Sathish Gopalakrishnan, Luca Santinelli, and Liliana Cucu-Grosjean. Probabilities for Mixed-Criticality Problems: Bridging the Uncertainty Gap. In *The Work in Progress session of the 32nd IEEE Real-time Systems Symposium - RTSS 2011*, Wien, Austria, November 2011. URL: <https://hal.inria.fr/hal-00646586>.
- 3 Eitan Altman. Constrained markov decision processes with total cost criteria: Lagrangian approach and dual linear program. *Math. Meth. of OR*, 48(3):387–417, 1998. doi:10.1007/s001860050035.
- 4 Eitan Altman. *Constrained Markov Decision Processes*. Chapman and Hall/CRC, 1999.
- 5 Robert B. Ash. *Real Analysis and Probability*. Academic Press, 1972.
- 6 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034.
- 7 Sanjoy Baruah. Mixed criticality schedulability analysis is highly intractable. to appear. URL: <http://www.cs.unc.edu/~baruah/Submitted/02cxty.pdf>.
- 8 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. In Petr Hlinený and Antonín Kucera, editors, *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6281 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2010. doi:10.1007/978-3-642-15155-2_10.
- 9 Sanjoy K. Baruah and Zhishan Guo. Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 31–40. IEEE Computer Society, 2014. doi:10.1109/RTSS.2014.15.
- 10 Sanjoy K. Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In Marco Caccamo, editor, *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, pages 13–22. IEEE Computer Society, 2010. doi:10.1109/RTAS.2010.10.
- 11 Sanjoy K. Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 147–155. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.26.
- 12 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.
- 13 Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121(1-2):49–107, 2000. doi:10.1016/S0004-3702(00)00033-3.
- 14 Alan Burns and Robert I. Davis. Mixed criticality systems - a review. Preprint, 2015. URL: <http://www-users.cs.york.ac.uk/burns/review.pdf>.
- 15 Yao Chen, Qiao Li, Zheng Li, and Huagang Xiong. Efficient schedulability analysis for mixed-criticality systems under deadline-based scheduling. *Chinese Journal of Aeronautics*, 27(4):856 – 866, 2014. doi:http://dx.doi.org/10.1016/j.cja.2014.05.003.
- 16 José Luis Díaz, Daniel F. García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*, pages 289–300. IEEE Computer Society, 2002. doi:10.1109/REAL.2002.1181583.
- 17 José Luis Díaz and José María López. Probabilistic analysis of the response time in a real-time system. *Technical Report*, 2001.

- 18 José Luis Díaz and José María López. Safe extensions to the stochastic analysis of real-time systems. *Technical Report*, 2004.
- 19 José Luis Díaz, José María López, Manuel García Vazquez, Antonio M. Campos, Kanghee Kim, and Lucia Lo Bello. Pessimism in the stochastic analysis of real-time systems: Concept and applications. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004), 5-8 December 2004, Lisbon, Portugal*, pages 197–207. IEEE Computer Society, 2004. doi:10.1109/REAL.2004.41.
- 20 Dmitri A. Dolgov and Edmund H. Durfee. Symmetric approximate linear programming for factored mdps with application to constrained problems. *Ann. Math. Artif. Intell.*, 47(3-4):273–293, 2006. doi:10.1007/s10472-006-9038-x.
- 21 Peter Geibel and Fritz Wyszotzki. Risk-sensitive reinforcement learning applied to control under constraints. *J. Artif. Intell. Res.*, 24:81–108, 2005. doi:10.1613/jair.1666.
- 22 Zhishan Guo and Sanjoy K. Baruah. Implementing mixed-criticality systems upon a preemptive varying-speed processor. *LITES*, 1(2):03:1–03:19, 2014. doi:10.4230/LITES-v001-i002-a003.
- 23 Zhishan Guo, Luca Santinelli, and Kecheng Yang. EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 187–196. IEEE Computer Society, 2015. doi:10.1109/RTCSA.2015.8.
- 24 Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015. URL: <http://www.gurobi.com>.
- 25 Onésimo Hernández-Lerma. *Adaptive Markov control processes*. Applied mathematical sciences. Springer, New York, 1989.
- 26 Onésimo Hernández-Lerma and Jean B Lasserre. *Discrete-Time Markov Control Processes: Basic Optimality Criteria*. Stochastic Modelling and Applied Probability. Springer-Verlag, New York, 1996.
- 27 L. C. M. Kallenberg. Unconstrained and constrained dynamic programming over a finite horizon. *Technical Report*, 1981.
- 28 L. C. M. Kallenberg. *Linear Programming and Finite Markovian Control Problems*. Amsterdam : Mathematisch Centrum, 1983.
- 29 Lodewijk C. M. Kallenberg. Survey of linear programming for standard and nonstandard markovian control problems. part I: theory. *Math. Meth. of OR*, 40(1):1–42, 1994. doi:10.1007/BF01414028.
- 30 Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000. doi:10.1145/347476.347479.
- 31 Kanghee Kim, José Luis Díaz, Lucia Lo Bello, José María López, Chang-Gun Lee, and Sang Lyul Min. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *IEEE Trans. Computers*, 54(11):1460–1466, 2005. doi:10.1109/TC.2005.174.
- 32 Achim Klenke. *Probability Theory: A Comprehensive Course*. Universitext. Springer, 2 edition, 2014.
- 33 Dorin Maxim and Liliana Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 224–235. IEEE Computer Society, 2013. doi:10.1109/RTSS.2013.30.
- 34 Pascal Poupart, Craig Boutilier, Relu Patrascu, and Dale Schuurmans. Piecewise linear value function approximation for factored mdps. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 292–299. AAAI Press / The MIT Press, 2002. URL: <http://www.aaai.org/Library/AAAI/2002/aaai02-045.php>.
- 35 M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York, 5 edition, 2005.
- 36 Martin L. Shooman. Avionics software problem occurrence rates. In *Seventh International Symposium on Software Reliability Engineering, ISSRE 1996, White Plains, NY, USA, October 30, 1996-Nov. 2, 1996*, pages 55–64. IEEE Computer Society, 1996. doi:10.1109/ISSRE.1996.558695.
- 37 Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 93–102. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.20.
- 38 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA*, pages 239–243. IEEE Computer Society, 2007. doi:10.1109/RTSS.2007.47.

Errata for Three Papers (2004-05) on Fixed-Priority Scheduling with Self-Suspensions*


Konstantinos Bletsas

CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
Porto, Portugal
ksbs@isep.ipp.pt
 <https://orcid.org/0000-0002-3640-0239>


Neil C. Audsley

University of York
York, United Kingdom
neil.audsley@york.ac.uk
 <https://orcid.org/0000-0003-3739-6590>


Wen-Hung Huang

TU Dortmund
Dortmund, Germany
wen-hung.huang@tu-dortmund.de
 <https://orcid.org/0000-0001-9446-4719>

Jian-Jia Chen

TU Dortmund
Dortmund, Germany
jian-jia.chen@tu-dortmund.de
 <https://orcid.org/0000-0001-8114-9760>

Geoffrey Nelissen

CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
Porto, Portugal
grpn@isep.ipp.pt
 <https://orcid.org/0000-0003-4141-6718>

Abstract

The purpose of this article is to (i) highlight the flaws in three previously published works [3, 2, 7] on the worst-case response time analysis for tasks

with self-suspensions and (ii) provide straightforward fixes for those flaws, hence rendering the analysis safe.

2012 ACM Subject Classification Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Software and its engineering → Real-time schedulability

Keywords and Phrases real-time; scheduling; self-suspension; worst-case response time analysis

Digital Object Identifier 10.4230/LITES-v005-i001-a002

Received 2015-07-17 **Accepted** 2018-02-12 **Published** 2018-05-30

* This paper is supported by DFG, as part of the Collaborative Research Center SFB876 (<http://sfb876.tu-dortmund.de/>) project B2.



1 Introduction

Often, in embedded systems, a computational task running on a processor must suspend its execution to, typically, access a peripheral or launch computation on a remote co-processor. Those tasks are commonly referred to as *self-suspending*. During the duration of the self-suspension, the processor is free to be used by any other tasks that are ready to execute. This seemingly simple model is non-trivial to analyse from a worst-case response time (WCRT) perspective since the classical “critical instant” of Liu and Layland [13] (i.e., simultaneous release of all tasks) no longer necessarily provides the worst-case scenario when tasks may self-suspend. A simple solution consists in modelling the duration of the self-suspension as part of the self-suspending task’s execution time. This so-called “self-suspension oblivious” approach allows to use the “critical instant” of Liu and Layland but often at the cost of too much pessimism. Therefore, various efforts have been made to derive less pessimistic, but still safe, analyses.

The results published in [3, 2, 7, 6] propose solutions for computing upper bounds on the response times of self-suspending tasks. However, we have now come to understand that they were flawed, i.e., they do not always output safe upper bounds on the task WCRTs. Through this paper, we therefore seek to highlight the respective flaws and propose appropriate fixes, rendering the two analysis techniques previously proposed in [3][2][7] safe.

2 Process model and notation

We assume a single processor and n independent sporadic¹ computational tasks scheduled under a fixed-priority policy. Each task τ_i has a distinct priority p_i , an inter-arrival time T_i and a relative deadline D_i , with $D_i \leq T_i$ (constrained deadline model). Each job released by τ_i may execute for at most X_i time units on the processor (its *worst-case execution time in software* – S/W WCET) and spend at most G_i time units in self-suspension (its “H/W WCET”). What in the works [3, 2, 7, 6] is referred to as (simply) “the worst-case execution time” of τ_i , denoted by C_i , is the time needed for the task to complete, in the worst-case, in the absence of any interference from other tasks on the processor. Hence C_i also accounts for the latencies of any self-suspensions in the task’s critical path². This terminology differs somewhat from that used in other works, which call WCET what we call the S/W WCET. This is mainly because it echoes a view inherited from hardware/software co-design that the task *is* executing even when self-suspended on the processor, albeit remotely (i.e., on a co-processor).

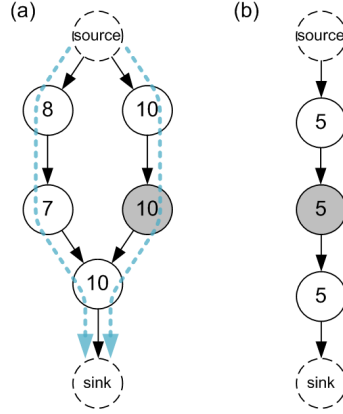
As illustrated on Figure 1, in the general case, $C_i \geq X_i$, $C_i > G_i$ but $C_i \leq X_i + G_i$, because X_i and G_i are not necessarily observable for the same control flow, unless it is explicitly specified or inferable from information about the task structure that $C_i = X_i + G_i$.

Additionally, lower bounds on the S/W and the “H/W” best-case execution times are denoted by \hat{X}_i and \hat{G}_i , respectively.

Our past work considered two submodels (referred to as “simple” and “linear”), depending on the degree of knowledge that one has regarding the location of the self-suspending regions inside the process activation and whether or not $C_i = X_i + G_i$.

¹ The original papers, assumed periodic tasks with *unknown* offsets. It was in the subsequent PhD thesis [6] that the observation was made that the results apply equally to the sporadic model, which is more general in terms of the possible legal schedules that may arise.

² We assume, as in [3, 2, 7, 6], that there is no contention over the co-processors or peripherals accessed during a self-suspension.



■ **Figure 1** Examples of task graphs for task with self-suspensions. White nodes represent sections of code with single-entry/single-exit semantics. Grey nodes represent remote operations, i.e., self-suspending regions. The nodes are annotated with execution times, which in this example are deterministic for simplicity. The directed edges denote the transition of control flow. Any task execution corresponds to a path from source to sink. For task graph (a), two different control flows exist (shown with dashed lines). In this case, the software execution and the time spent in self-suspension are maximal for different control flows. As a result of this, $C < X + G$; specifically, $C = X = 25$ and $G = 10$. However, task graph (b) is linear, so it holds that $C = X + G$ for that task.

2.1 The simple model

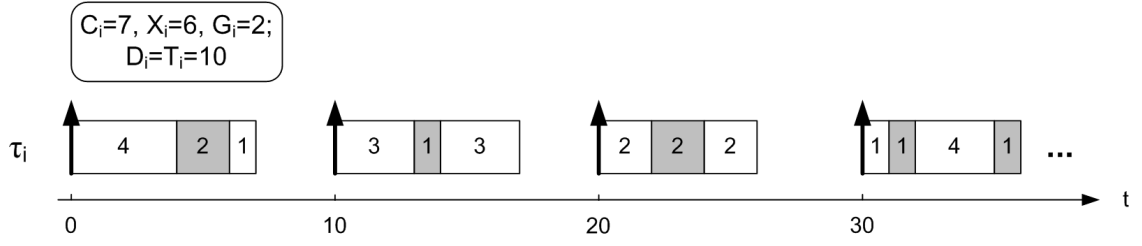
The simple model, assumed in [2, 3], is also called “floating” or “dynamic self-suspension model” in many later works of the state-of-the-art. This model is entirely agnostic about the location of self-suspending regions in the task code. Hence, there is no information on the number of self-suspending regions, on the instants at which they may be activated and for how long each of them may last at run-time. Moreover, the self-suspension pattern may additionally differ for subsequent jobs released by the same task τ_i . The sums of the lengths of the “S/W” and “H/W” execution regions are however subject to the constraints imposed by the attributes C_i , X_i and G_i . Figure 2 illustrates this concept.

2.2 The linear model

The linear model, which was presented in [7], is also known as the “multi-segment self-suspension model” in many later works. It assumes that each task is structured as a “pipeline” of interleaved software and self-suspending regions, or “segments”. Each of these segments has known upper and lower bounds on its execution time. This means that, in all cases, $C_i = X_i + G_i$ and the task-level upper and lower bounds on its software (respectively, hardware) execution time, X_i and \hat{X}_i (respectively, G_i and \hat{G}_i) are obtained as the sum of the respective estimates of all the software (respectively, hardware) segments.

3 The analysis in [2, 3], its flaws and how to fix it.

The two works [2, 3] that targeted the simple model, sought to derive the task WCRTs by shifting the distribution of software execution and self-suspension intervals *within* the activation of each higher-priority task in order to create the most unfavorable pattern, across job boundaries. This also involved aligning the task releases accordingly, in order to obtain (what we thought to be) the worst case. In order to facilitate the explanation of the specifics, it is perhaps best to first



■ **Figure 2** Under the simple model any job by a given task τ_i can execute for at most X_i units in software, at most G_i time units in hardware and at most C_i time units overall. The locations and number of the hardware operations (self-suspensions, from the perspective of software execution) may vary arbitrarily for different jobs by the same task, subject to the previous constraints. This is depicted here for a task τ_i , with the parameters shown, which (for simplicity) is the only task in its system. Upward-pointing arrows denote task arrivals (and deadlines, since the task set happens to be implicit-deadline). Shaded rectangles denote remote execution (i.e., self-suspension).

present the corresponding equation for computing the WCRT of a task τ_i derived in [3]:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j \quad (1)$$

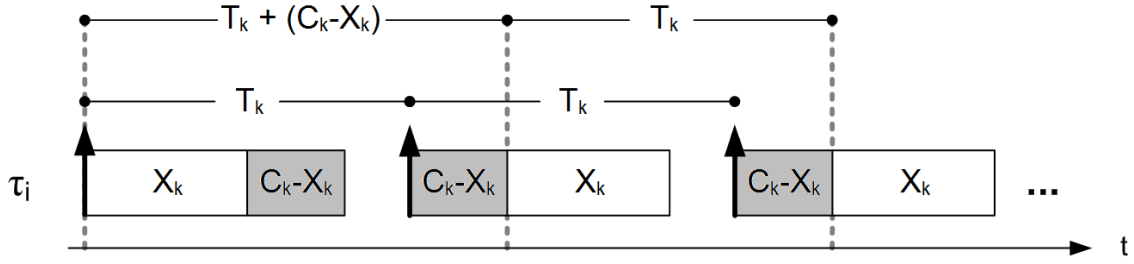
where the term $hp(i)$ is the set of tasks with higher-priority than τ_i . For the special case where $C_i = X_i + G_i, \forall i$, the above equation can be rewritten as [2]

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + G_j}{T_j} \right\rceil X_j \quad (2)$$

Intuitively, τ_i is pessimistically treated as preemptible at any instant, even those at which it is self-suspended. Each interfering job released by a higher-priority task τ_j contributes up to X_j time units of interference to the response time of τ_i . However, the variability in the location of self-suspending regions creates a jitter in the software execution of each interfering task. The term $(C_j - X_j)$, for each $\tau_j \in hp(i)$, in the numerator, which is akin to a jitter in Equation 1, attempted to account for this variability. Intuitively, it represents the potential internal jitter, *within* an activation of τ_j , i.e., when its net execution time (in software or in hardware) is considered, and disregarding any time intervals when τ_j is preempted. Figure 3 illustrates this concept for some task τ_k .

However, as we will show in Example 1, in the general case the jitter can be larger than $(C_j - X_j)$. This is because the software execution of τ_j can be pushed further to the right along the axis of time, due to the interference that τ_j suffers from even higher-priority tasks.

It is worth noting that the authors of [2] were fully aware at the time that the term $\left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j$ is not an upper bound on the worst-case interference exerted upon τ_i from any *individual* task $\tau_j \in hp(i)$. However, it was considered (and erroneously claimed, with faulty proof) that $\sum_{j \in hp(i)} \left\lceil \frac{R_i + (C_j - X_j)}{T_j} \right\rceil X_j$ was nevertheless an upper bound for the total interference *jointly* caused by all tasks in $hp(i)$, in the worst case. The flaw in that reasoning came from assuming that the effect of any additional jitter of interfering task τ_j , caused by interference exerted upon it by even higher-priority tasks would already be “captured” by the corresponding terms modelling the interference upon τ_i by $hp(j) \subset hp(i)$. This would then suppress the need to include it twice.



■ **Figure 3** For a job by some task τ_k that executes in software for X_k time units and C_k time units overall (i.e., in software and in hardware), the latest that it can start executing in software, in terms of net execution time (i.e., excluding preemptions) is after having executed for $C_k - X_k$ time units in hardware. Differences in the placement of software and hardware execution across different jobs of τ_k manifest themselves as jitter for its software execution.

Accordingly, then, the worst-case scenario for the purposes of maximisation of the response time of a task τ_i , released without loss of generality at time $t = 0$ would happen when each higher-priority task

- is released at time $t = -(C_j - X_j)$ and then releases its subsequent jobs with its minimum inter-arrival time (i.e., at instants $t = T_j - (C_j - X_j), 2T_j - (C_j - X_j), \dots$;
- switches for the first time to execution in software (for a full X_j time units) at $t = 0$, for its first interfering job, i.e., after a self-suspension of $C_j - X_j$ time units;
- executes in software for X_j time units as soon as possible for its subsequent jobs.

Figure 4(a) plots the schedule that reproduces this alleged worst-case scenario, for the lowest-priority task in the example task set of Table 1. In this case, the top-priority task τ_1 happens to be a regular non-self-suspending task, so its worst-case release pattern reduces to that of Liu and Layland. However, for the middle-priority task τ_2 which self-suspends, its execution pattern matches that described above.

However, this schedule does not constitute the worst-case, as evidenced by the following counter-example:

► **Example 1.** Consider the task set of Table 1. Assume that the execution times of software segments and the durations of self-suspending regions are deterministic. As shown below using a fixed point iteration over Equation 1, the analysis in [2, 3] would yield $R_3 = 12$:

$$R_3 = C_3 + \left\lceil \frac{R_3 + C_1 - X_1}{T_1} \right\rceil X_1 + \left\lceil \frac{R_3 + C_2 - X_2}{T_2} \right\rceil X_2 \Rightarrow R_3 = 1 + \left\lceil \frac{R_3}{2} \right\rceil 1 + \left\lceil \frac{R_3 + 5}{20} \right\rceil 5$$

$$R_3^{(0)} = 1$$

$$R_3^{(1)} = 1 + \left\lceil \frac{1}{2} \right\rceil 1 + \left\lceil \frac{1 + 5}{20} \right\rceil 5 = 7$$

$$R_3^{(2)} = 1 + \left\lceil \frac{7}{2} \right\rceil 1 + \left\lceil \frac{7 + 5}{20} \right\rceil 5 = 10$$

$$R_3^{(3)} = 1 + \left\lceil \frac{10}{2} \right\rceil 1 + \left\lceil \frac{10 + 5}{20} \right\rceil 5 = 12$$

$$R_3^{(4)} = 1 + \left\lceil \frac{12}{2} \right\rceil 1 + \left\lceil \frac{12 + 5}{20} \right\rceil 5 = 12$$

The corresponding schedule is shown in Figure 4(a). However, the schedule of Figure 4(b), which is perfectly legal, disproves the claim that $R_3 = 12$, because τ_3 in that case has a response time

■ **Table 1** A set of tasks with self-suspensions. The lower the task index, the higher its priority.

τ_i	C_i	X_i	G_i	T_i
τ_1	1	1	0	2
τ_2	10	5	5	20
τ_3	1	1	0	∞

of $22 - 5\epsilon$, where ϵ is an arbitrarily small quantity. It therefore proves that the analysis initially presented in [2] and [3] is unsafe.

Let us now inspect what makes the scenario depicted in the schedule of Figure 4 so unfavourable that the analysis in [2, 3] fails, and at the same time let us understand how the analysis could be fixed.

Looking at the first interfering job released by τ_2 in Figure 4, one can see that almost all its software execution is still distributed to the very right (which was supposed to be the worst-case in [3]). However, by “strategically” breaking up what would have otherwise been a contiguous self-suspending region of length G_2 in the left, with arbitrarily short software regions of length ϵ beginning at the same instants that the even higher-priority task τ_1 is released, a particularly unfavourable effect is achieved. Namely, the execution of τ_1 on the processor and the self-suspending regions of τ_2 , “sandwiched” in between are effectively serialised. In practical terms, it is the equivalent of the execution of τ_1 on the processor preempting the execution of τ_2 on the co-processor! This means that, when finally τ_2 is done with its self-suspensions, its remaining execution in software is almost its entire X_2 , but occurs with a jitter far worse than that modelled by Equation 1. And, when analysing τ_3 , this effect was not captured indirectly, via the term modelling the interference exerted by τ_1 onto τ_3 .

So in retrospect, although each job by each $\tau_j \in hp(i)$ can contribute at most X_j time units of interference to τ_i , the terms $(C_j - X_j)$ in Equation 1, that are analogous to jitters, are unsafe. The obvious fix is thus to replace those with the true jitter terms for software execution. As proven in Lemma 2 below, safe upper bounds for these are $R_j - C_j$, $\forall \tau_j \in hp(i)$.

Reconsidering the analysis presented in [2, 3] in light of this counter-example, one can draw the following conclusions:

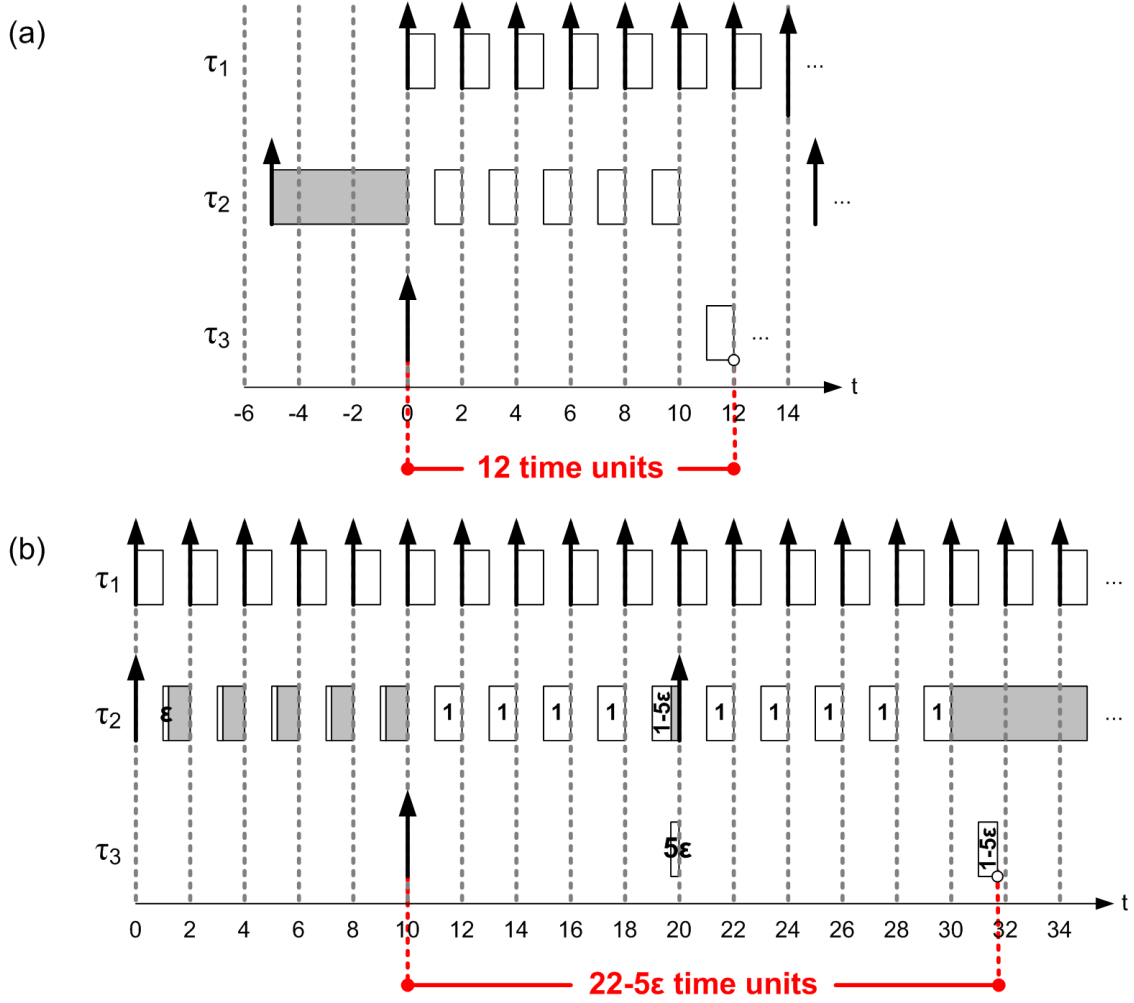
1. the terms X_j , one for every higher-priority task, in Equation 1, which model the fact that each job released by a task $\tau_j \in hp(i)$ can contribute at most X_j time units of interference, do not introduce optimism;
2. the terms $(C_j - X_j)$, one for every higher-priority task, in Equation 1, that are analogous to jitters, are unsafe.

Formally, these conclusions can be summarised by the following Lemma 2, that serves as a sufficient schedulability test:

► **Lemma 2** (Corresponding to Corollary 1 in [9]). *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. If every task $\tau_j \in hp(i)$ is schedulable (i.e., if an upper bound R_j on the worst-case response time of τ_j exists with $R_j \leq D_j \leq T_j$) and, additionally, the smallest solution to the following recursive equation is upper-bounded by D_i ,*

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + (R_j - X_j)}{T_j} \right\rceil X_j \quad (3)$$

then τ_i is also schedulable and its worst-case response time is upper-bounded by R_i , as computed by Equation 3.



■ **Figure 4** Subfigure (a) depicts the schedule, for the task set of Table 1 that was supposed to result in the WCRT for τ_3 according to the analysis presented in [2, 3]. Subfigure (b) depicts a different legal schedule that results in a higher response time for τ_3 .

3.1 Proof of Lemma 2

Consider a schedule Ψ of the self-suspending task system in consideration whereby some job of task τ_i is released at time r_i and completed at time f_i .

We define a transformed schedule Ψ' as the schedule in which (i) the jobs of every higher-priority task $\tau_j \in hp(i)$ are released at the exact same instants as in Ψ ; (ii) only one job by τ_i is released, at time r_i ; (iii) no jobs by lower-priority tasks are released and (iv) the suspensions by all higher-priority jobs take place during the exact same intervals as in Ψ ; additionally (v) we modify the job of τ_i (which in Ψ executed on the processor for x_i time units and was suspended for g_i time units) such that it executes on the processor for $C_i \geq x_i + g_i$ time units. Recall that C_i is defined as the worst-case combined execution in software and hardware, i.e., sum of processor-based execution and self-suspension. After this last conversion (a safe, widely used transformation known in the literature as “conversion of suspension to processor-based computation”, followed by a potential increase of that processor-based execution time), we can verify (see also Lemma 3 just below) that: (i) Over the interval $[r_i, f_i]$, for every instant that the job by τ_i in Ψ is executing

or suspended or suspended and no higher-priority task is executing on the processor, the job by τ_i in Ψ' is executing on the processor, at the same instant. And (ii) for the completion time f'_i of τ'_i in Ψ' , it holds that $f'_i \geq f_i$; in other words the response time of the job in consideration in Ψ' does not decrease over that in Ψ .

For notational brevity, we denote the (only) job of τ_i in Ψ' as originating from a task τ'_i with $C'_i = X'_i = C_i$, $G'_i = 0$, $D'_i = D_i$, $T'_i = T_i$. Note that Ψ' remains a fixed-priority schedule.

► **Lemma 3** (Corresponding to Lemma 2 in [9] with minor variations). *Assuming that the worst-case response time of τ_i is upper bounded by T_i and given the definition of schedule Ψ' , the response time of the job of τ'_i in consideration in Ψ' is not smaller than the response time of the corresponding job of τ_i in Ψ , for any possible x_i, g_i such that $x_i \leq X_i$ and $g_i \leq G_i$ and $x_i + g_i \leq C_i$.*

Proof. We know, by definition of fixed-priority schedules, that jobs by lower-priority tasks do not impact the response time of the jobs by τ_i . Therefore, their elimination in Ψ' has no impact on the response time of the jobs of τ_i . Moreover, since from the assumption in the claim, the worst-case response time of τ_i is upper-bounded by T_i , no other job by τ_i in Ψ impacts the schedule of the job by τ_i released at r_i . Since all other parameters (i.e., releases and suspensions of higher-priority tasks) that may influence the scheduling decisions are kept identical between Ψ and Ψ' , the response time (\bar{R}) of the job by τ_i released at time r_i would have been identical in Ψ' to the one in Ψ if we had not converted that job's suspension time to processor-based computation.

Let x_i and g_i respectively denote the total duration of processor-based execution and self-suspension characterising the job of τ_i in consideration. Given that $x_i + g_i \leq C_i$ for any job by τ_i means that additionally substituting in Ψ' the particular job τ_i by a job by τ'_i as defined above cannot result in the response time being lower than \bar{R} , which in turn was shown to be no less than the response time of the job in Ψ . ◀

We now analyse the properties of the fixed-priority schedule Ψ' . For any interval $[r_i, t)$, with $t \leq f_i$, we are going to prove an upper bound (denoted as $\text{exec}(r_i, t)$) on the amount of time during which the processor is executing tasks.

Because in Ψ' there exist no jobs of lower priority than that of τ'_i , we only focus on the execution of the tasks in $hp(i) \cup \tau'_i$. (Recall that we use the notation τ'_i here instead of simply τ_i , because when constructing Ψ' from Ψ , we replaced the self-suspending job of τ_i released at r_i by a job of the same priority that executes entirely in software for $X'_i \stackrel{\text{def}}{=} C_i \leq X_i + G_i$ time units.)

► **Lemma 4.** *For any t such that $r_i \leq t < f'_i$, the cumulative amount of time that τ'_i executes on the processor over the interval $[r_i, t)$, denoted by $\text{exec}_i(r_i, t)$ is strictly smaller than C_i .*

Proof. Since the finishing time of the transformed job by τ_i is $f'_i > t$, it means that it has executed for strictly less than its total execution time of C_i . ◀

► **Lemma 5** (Corresponding to Lemma 8 in [9]). *Assume that $R_j \leq T_j$ for all jobs by τ_j in Ψ' . Let J_j be the last job of τ_j released before r_i in Ψ' and let x_j^* be the remaining processor execution time of J_j at time r_i . For any task $\tau_j \in hp(i)$ and any $\Delta \geq 0$, it holds that*

$$\text{exec}_j(r_i, r_i + \Delta) \leq \hat{W}_j^0(\Delta, x_j^*)$$

where

$$\hat{W}_j^0(\Delta, x_j^*) \stackrel{\text{def}}{=} \begin{cases} W_j^1(\Delta) & \text{if } x_j^* = 0 \\ \Delta & \text{if } x_j^* > 0 \text{ and } \Delta \leq x_j^* \\ x_j^* & \text{if } x_j^* > 0 \text{ and } x_j^* < \Delta \leq \rho_j \\ x_j^* + W_j^1(\Delta - \rho_j) & \text{if } x_j^* > 0 \text{ and } \rho_j < \Delta \end{cases} \quad (4)$$

with

$$W_j^1(\Delta) \stackrel{\text{def}}{=} \left\lfloor \frac{\Delta}{T_j} \right\rfloor + \min \left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j \right\} \quad (5)$$

and $\rho_j \stackrel{\text{def}}{=} T_j - R_j + x_j^*$

Proof. We explore two complementary cases:

- **Case $x_j^* = 0$:** In this case, there is no residual (sometimes called carry-in) workload of τ_j at time r_i . Furthermore, $\text{exec}_j(r_i, r_i + \Delta)$ is maximised when every job of τ_j released after r_i executes on the processor for its full processor execution time X_j , with any self-suspension strictly occurring (if at all) after it completes its X_j time units of execution on the processor. (Remember that there is no carry-in workload and hence pushing the execution of a job later by means of self-suspension will not increase the amount of computation within the window $[r_i, t)$). This is analogous, in terms of processor-based workload pattern, to τ_j being a sporadic, non-self-suspending task with a worst-case execution time of X_j time units on the processor. Since, as already shown in the literature [5], $W_j^1(\Delta)$, which is usually called workload function, is an upper bound on the cumulative amount of time that a sporadic task with a worst-case execution time X_j and inter-arrival time T_j can execute on the processor without self-suspension, we know that $\text{exec}_j(r_i, r_i + \Delta) \leq W_j^1(\Delta)$. This proves case 1 of (4).
- **Case $x_j^* > 0$:** By assumption, there is $R_j \leq T_j$. Additionally, the earliest completion time for the job J_j of τ_j with residual workload x_j^* at time r_i must be $r_i + x_j^*$ (from the definition of x_j^*). Therefore, the earliest arrival time of a job of τ_j *strictly after* r_i is at least $r_i + x_j^* + (T_j - R_j)$, which is equal to $r_i + \rho_j$. Since no other job of τ_j is released in $[r_i, r_i + \rho_j)$, this means that $\text{exec}_j(r_i, r_i + \Delta)$ is upper-bounded by $\min\{\Delta, x_j^*\}$ for $\Delta \leq \rho_j$, thereby proving cases 2 and 3 of (4). Furthermore, by assumption, the job of τ_j with residual workload x_j^* at time r_i completes no earlier than time $r_i + \rho_j$. Therefore, following the same reasoning as for the case that $x_j^* = 0$, it holds that $\text{exec}_j(r_i + \rho_j, r_i + \Delta)$ is upper bounded by $W_j^1(\Delta - \rho_j)$ when $\Delta > \rho_j$. This proves the fourth case of (4). ◀

► **Lemma 6** (Lemma 9 in [9]). $\forall \Delta > 0$, it holds that $\hat{W}_j^0(\Delta, X_j) \geq \hat{W}_j^0(\Delta, x_j^*)$.

Proof. See proof in [9]. ◀

► **Lemma 7.** For any $\Delta > 0$, it holds that

$$\hat{W}_j^0(\Delta, X_j) \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j \quad (6)$$

Proof. From the definition of $W_j^1(\Delta)$ in (5), we have

$$\begin{aligned} W_j^1(\Delta) &= \left\lfloor \frac{\Delta}{T_j} \right\rfloor X_j + \min \left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, X_j \right\} \\ &\leq \left\lceil \frac{\Delta}{T_j} \right\rceil X_j \end{aligned} \quad (7)$$

If $0 < \Delta \leq X_j$, then by (4), it holds that $\hat{W}_j^0(\Delta, X_j) = \Delta$. Moreover, because the worst-case response time R_j of a task cannot be smaller than its worst-case execution time $C_j \geq X_j$, we have that $\frac{\Delta + R_j - X_j}{T_j} > 0$. Hence, $\hat{W}_j^0(\Delta, X_j) = \Delta \leq X_j \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j$

If $\Delta > X_j$, then by the third and fourth cases of (4) and using (7) that we just proved, it holds that $\hat{W}_j^0(\Delta, X_j) \leq X_j + W_j^1(\Delta - (T_j - R_j + X_j)) \leq X_j + \left\lceil \frac{\Delta - T_j + (R_j - X_j)}{T_j} \right\rceil X_j \leq \left\lceil \frac{\Delta + R_j - X_j}{T_j} \right\rceil X_j$. ◀

02:10 Errata for Three Papers on FP Scheduling with Self-Suspensions

Now that we have derived an upper bound on the cumulative execution time $\text{exec}_j(r_i, r_i + \Delta)$ by each task τ_j in Ψ' , we can use these upper bounds in order to derive properties for the schedule over any interval $[r_i, t)$.

Recall that, for the schedule Ψ' , the finishing time of the job of τ_i in consideration is $f'_i \geq f_i$ (where f_i is its corresponding finishing time in Ψ).

► **Lemma 8.** *Assuming that the worst-case response time of τ_i is upper bounded by T_i , and assuming that $R_j \leq T_j$ for all jobs by τ_j in Ψ' . $\forall t \mid r_i \leq t < f'_i$ it holds that:*

$$C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j > t - r_i \quad (8)$$

Proof. When we constructed Ψ' , we transformed any suspension time of τ_i into processor execution time. Hence, it must hold that there is no idle time within $[r_i, f'_i)$, i.e., between the release and completion time of the transformed job of τ_i . Indeed, if there was an idle time within $[r_i, f'_i)$, it would mean that either τ_i completed its job before f'_i or the scheduler would not be work conserving. A contradiction with the assumptions of this problem in both cases.

Therefore, for every t such that $r_i \leq t < f'_i$, it holds that $\sum_{j=1}^i \text{exec}_j(r_i, t) = t - r_i$. By application of Lemmas 5 and 6 to the LHS, we get

$$\text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \hat{W}_j^0(t - r_i, X_j) \geq t - r_i$$

Further, applying Lemma 7,

$$\text{exec}_i(r_i, t) + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j \geq t - r_i$$

The fact that the (transformed) job by τ_i has not yet completed at $t < f'_i$ in Ψ' also means (see Lemma 4) that $\text{exec}_i(r_i, t) < C_i$. Substituting to the LHS of the above equation yields $C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t - r_i + R_j - X_j}{T_j} \right\rceil X_j > t - r_i$. ◀

► **Corollary 9.** *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. Assume that the worst-case response time of τ_i does not exceed T_i and also that $R_j \leq T_j, \forall \tau_j \in \text{hp}(i)$, where R_j denotes an upper bound on the worst-case response time of the respective task τ_j . Then, the worst-case response time of τ_i is upper-bounded by the minimum t greater than 0 for which the following inequality holds.*

$$C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j \leq t \quad (9)$$

Proof. Direct consequence of Lemma 8. ◀

Having proven Corollary 9, what remains to show is the following:

► **Lemma 10.** *Consider a uniprocessor system of constrained-deadline self-suspending tasks and one task τ_i among those, in particular. Assume that $R_j \leq T_j, \forall \tau_j \in \text{hp}(i)$, where R_j denotes an upper bound on the worst-case response time of the respective task τ_j . If the worst-case response time of τ_i is greater than T_i or unbounded (which implies that τ_i is unschedulable), it holds that*

$$C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{t + (R_j - X_j)}{T_j} \right\rceil X_j > t, \forall t \mid 0 < t \leq T_i \quad (10)$$

Proof. By the assumption that $R_i > T_i$ for some task τ_i , there exists a schedule Ψ such that the response time of at least one job of τ_i is strictly larger than T_i . Consider the first such job in the schedule, and suppose that it arrives at time r_i . At that instant, there is no other unfinished job by τ_i in the system (or else, this would contradict the assumption that the job arriving at r_i is the first job of τ_i whose response time exceeds T_i). So by Lemma 7 we can safely remove all other jobs by task τ_i that arrived before or at time r_i , without affecting the response time of the job that arrived at time r_i . Nor is its response time affected, if we additionally remove all other jobs of τ_i that arrived after time r_i . Let f_i be the finishing time of the job by τ_i that arrived at r_i in the above schedule, after removing all other jobs of that task. We therefore know that $f_i - r_i > T_i$.

Then, we can follow all the procedures and steps in the proof of Corollary 9, to eventually reach Equation 10. \blacktriangleleft

The joint consideration of Corollary 9 and Lemma 10, which we have now proven, serves as proof of Lemma 2.

3.2 Discussion

We had already publicised the flaws in [2, 3] and the proposed fix, immediately upon realising the problem, in a technical report [8]. However, this article addresses the issue more rigorously, in terms of proofs.

Note also that Huang et al. already proposed a correct variation of Equation 3 in [12], using the deadline D_j of each higher priority task as the equivalent jitter term in the numerator of Equation 1 (see Theorem 2 in [12]). Although slightly more pessimistic, this solution has the advantage of remaining compatible with Audsley's Optimal Priority Assignment algorithm [1].

The fix proposed in Lemma 2, in this article, mirrors the approach taken by Nelissen et al. in [15], for which a proof sketch had already been provided (see Theorem 2 in [15]). Later, that approach was also extended for a more general result [9]. Compared to [9], the corrected analysis in the present article has the following differences:

1. In [9], the authors combine a second, newer technique for upper-bounding task response times, that had not been invented at the time that the papers under correction [2, 3] were published. That aspect of their analysis makes it more general.
2. In [9], the authors assume a model whereby $C_i = X_i + G_i$, $\forall i$. Instead, in this article, as in [3], we assume a slightly more general model whereby $C_i \leq X_i + G_i$. This makes the present analysis more general, in that regard, although there is no fundamental reason why the result in [9] cannot be similarly extended.

Other than the above observations, one “side-effect” of the proposed fix is that the WCRT estimate output by Equation 3 is no longer guaranteed to always dominate the estimate derived under the pessimistic but jitterless “suspension-oblivious” approach. In the “suspension-oblivious” approach, self-suspensions are treated as regular S/W executions on the processor. That is, every task $\tau_i \in \tau$ is modelled as a sporadic non-self-suspending task with a WCET equal to $C_i \geq X_i$. Using our notation described above, the corresponding WCRT equation for the suspension-oblivious approach is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (11)$$

A simple way for obtaining a WCRT upper bound that dominates the suspension-oblivious one is to always pick the smallest of the two WCRT estimates, output by Equations 3 and 11.

4 The analysis in [7], its flaws and how to fix it.

For the “linear model” described earlier, a different analysis was proposed in [7]. It uses the additional information available on the execution behaviour of each task, to provide tighter bounds on the task WCRTs. That analysis was called *synthetic* because it attempts to derive the WCRT estimate by synthesising (from the task attributes) task execution distributions that might not necessarily be observable in practice but (were supposed to) dominate the real worst-case execution scenario. Unfortunately, that analysis too, was flawed – and as we will see, the flaw was somehow inherited from the “simple” analysis already discussed in Section 3.

The linear model permits breaking up, for modelling purposes, the interference from each task τ_j upon a task τ_i into distinct terms X_{j_k} , each corresponding to one of the software segments of τ_j . These software segments are spaced apart by the corresponding self-suspending regions of τ_j , which, for analysis purposes, translates to a worst-case offset (see below) for every such term X_{j_k} . This allows in principle, for more granular and hence less pessimistic modelling of the interference. However, one problem that such an approach entails is that different arrival phasings between τ_i and every interfering task τ_j would need to be considered to find the worst-case scenario. This is yet undesirable from the perspective of computational complexity.

The main idea behind the synthetic analysis was to calculate the interference from a higher-priority task τ_j exerted upon the task τ_i under analysis assuming that the software segments and the self-suspending regions of τ_j appear in a potentially different rearranged order from the actual one. This so-called synthetic execution distribution would represent an interference pattern that dominates all possible interference patterns caused by τ_j on τ_i , without having to consider every possible phasing in the release of τ_j relative to τ_i . This approach is conceptually analogous to converting a task conforming to the generalised multiframe model [4] into an accumulatively monotonic execution pattern [14] – with the added complexity that the spacing among software segments is asymmetric and also variable at run-time (since the self-suspension intervals vary in duration within known bounds).

In terms of equations, the upper bound on the WCRT of a task τ_i claimed in [7] is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > {}^\xi O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^\xi O_{j_k} + A_j}{T_j} \right\rceil {}^\xi X_{j_k} \quad (12)$$

where $n(\tau_j)$ is the number of software segments of the linear task τ_j and the terms ${}^\xi X_{j_k}$ (a per-software-segment interference term), ${}^\xi O_{j_k}$ (a per-software-segment offset term) and A_j (a per-task term analogous to a jitter) are defined in terms of the worst-case synthetic execution distribution for τ_j .

For a rigorous definition, we refer the reader to [6]. However, for all practical purposes, one can intuitively define ${}^\xi X_{j_1}$ as the WCET of the longest software segment of τ_j ; ${}^\xi X_{j_2}$ as the WCET of the second longest software segment; and so on. Analogously, ${}^\xi G_{j_1}$ is the **best-case** length of the **shortest** hardware segment (i.e., self-suspending region) of τ_j (in terms of their BCETs); ${}^\xi G_{j_2}$ is that of the second shortest one; and so on. However, in addition to the actual self-suspending regions of τ_j , when creating this sorted sequence ${}^\xi G_{j_1}, {}^\xi G_{j_2}, \dots$ a so-called “notional gap” N_j of length $T_j - R_j$ is considered³. For tasks that both start and end with a software segment, this is the minimum spacing between the completion of a job by τ_j (i.e. its last software segment) and

³ In [7], the length of the notional gap was incorrectly given as $T_j - C_j$. In this paper, we consider the correct length of $T_j - R_j$, as in the thesis [6].

the time that the next job by τ_j arrives⁴. This is so that the interference pattern considered dominates all possible arrival phasings between τ_j and τ_i .

As for ${}^\xi O_{j_k}$, it was defined⁵ as

$${}^\xi O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} ({}^\xi X_{j_\ell} + {}^\xi G_{j_\ell}), & \text{otherwise} \end{cases} \quad (13)$$

Finally, A_j is given by

$$A_j = G_j - \hat{G}_j \quad (14)$$

As we will now demonstrate with the following counter-example, it is in the quantification of this final term A_j , that the analytical flaw lies.

► **Example 11.** Consider a task set with the parameters shown in Table 2. Each task is described as a vector consisting of the execution time ranges of its segments in the order of their activation; self-suspending regions are enclosed in parentheses. In this example, the execution times of the various software segments and self-suspending regions are deterministic. The analysis in [7], as sanitised in [6] with respect to the issue of Footnote 3, would be reduced to the familiar uniprocessor analysis of Liu and Layland [13] for the first few tasks, since τ_1 and τ_2 lack self-suspending regions. So we would get $R_1 = 2$ and $R_2 = 4$.

Using Equation 12 for τ_3 would yield $R_3 = 19$. Note that since the software segments and the intermediate self-suspending region of τ_3 execute with strict precedence constraints, it is also possible to derive another estimate for R_3 by calculating upper bounds on the WCRTs of the software/hardware segments and adding them together⁶. Doing this, and taking into account that the hardware operation suffers no interference, yields $R_3 = 5 + G_3 + 5 = 15$. This is in fact the exact WCRT, as evidenced in the schedule of Figure 5, for the job released by τ_3 at $t = 0$.

Next, to obtain R_4 we need to generate the worst-case execution distribution of τ_3 . Since, in the worst-case, τ_3 completes just before its next job arrives (see time 15 in Figure 5) its “notional gap” $N_3 = (T_3 - R_3)$ is 0. Then, the synthetic worst-case execution distribution for τ_3 is

$$[1, (0), 1, (5)]$$

which is equivalent to a non-self-suspending task with a WCET $C_3 = 2$.

From the fact that software and self-suspending region lengths are deterministic, we also have $A_3 = 0$ (using Equation 14). In other words, to compute R_4 according to this analysis is akin to replacing τ_3 with a (jitterless) sporadic task without any self-suspension, with $C_3 = 2$ and $D_3 = T_3 = 15$. Then, the corresponding upper bound computed with Equation 12 for the WCRT of τ_4 is $R_4 = 15$.

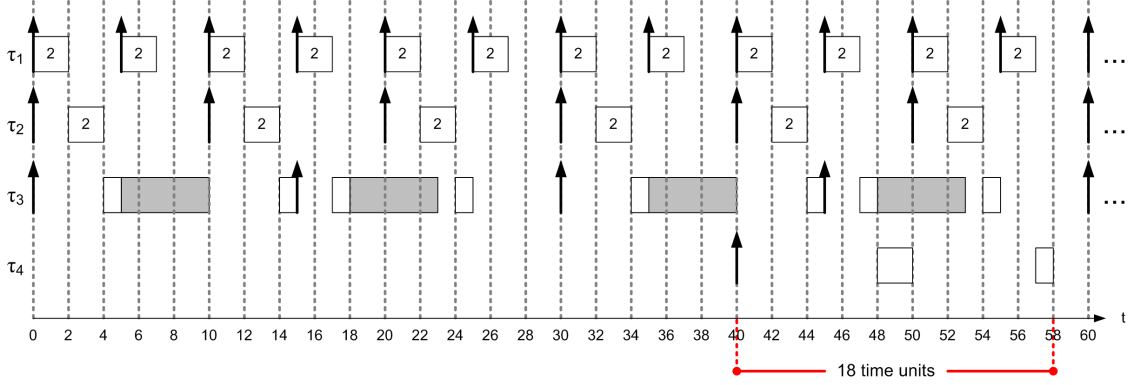
⁴ For tasks that start and/or end with a self-suspending region, the \hat{G} of the corresponding self-suspending region(s) is also incorporated to the notional gap. But that is part of a normalisation stage that precedes the formation of the worst-case synthetic execution distribution, so the reader may assume, without loss of generality, that the task both starts and ends with a software segment. For details, see page 115 in [6].

⁵ It is an opportunity to mention that in the corresponding equation (Eq. 12) of that thesis [6], there existed two typos: (i) the condition for the first case has “ $k = 0$ ” instead of “ $k = 1$ ” and (ii) the right-hand side for the second case does not have parentheses as should. We have rectified both typos in Equation 13 presented here.

⁶ In [6], the definition of WCRT is extended from tasks to software or hardware segments: The WCRT R_{i_j} of a segment τ_{i_j} is the maximum possible interval from the time that τ_{i_j} is eligible for execution until it completes. This approach of computing the WCRT of a self-suspending task by decomposing it in subsequences of one or more segments and adding up the WCRTs of those subsequences is also described there.

■ **Table 2** A set of linear tasks where the numbers within parentheses represent the lengths of the self-suspending regions and the other numbers represent the lengths of the S/W execution regions.

τ_i	execution distribution	D_i	T_i
τ_1	[2]	5	5
τ_2	[2]	10	10
τ_3	[1, (5), 1]	15	15
τ_4	[3]	20	∞



■ **Figure 5** A schedule, for the task set of Table 2, that highlights the flawedness of the synthetic analysis [7]. The job released by τ_4 at time 40 has a response time of 18 time units, which is more than the estimate for R_4 (i.e., 15) output by the analysis presented in [7].

However, the schedule of Figure 5, which is perfectly legal, disproves this. In that schedule, τ_1 , τ_2 , and τ_3 arrive at $t = 0$ and a job by τ_4 arrives at $t = 40$ and has a response time of 18 time units, which is larger than the value obtained for R_4 with Equation 12. Therefore, the analysis in [7] is also flawed.

For the purposes of fixing the analysis, we note that the characterisation of the interference by τ_j upon τ_i is correct for any schedule where no software segment by τ_j interferes more than once with τ_i . This holds by design, because the longest software segments and the shortest interleaved self-suspending regions are selected in turn (according to the property of accumulative monotonicity). Moreover, even in the case that there is interference multiple times by one or more software segments of the synthetic τ_j , i.e., when some γ segments interfere $\beta > 1$ times with τ_i and the remaining segments interfere $\beta - 1$ times with it, by the design of the equation it is ensured that these are its γ longest segments and that they are clustered together in time as closely as possible. Therefore, the problem lies in the quantification of the per-task term A_j , that acts as jitter for the task execution. Given that, for the simpler dynamic model, it was shown before that a value of $R_j - X_j$ for this jitter was safe, one may conjecture that using $A_j = R_j - X_j$ would also make the synthetic analysis for the segmented linear self-suspension model safe. After all, in the latter model, there is a smaller degree of freedom, in the execution and self-suspending behaviour of the tasks.

Indeed, not only is the above conjecture true, but below we are going to show that a smaller jitter term of $A_j = R_j - X_j - \hat{G}$ also works and makes the analysis safe.

► **Lemma 12.** *Consider a uniprocessor system of constrained-deadline linear (i.e., segmented) self-suspending tasks and one task τ_i among those, in particular. If for every task $\tau_j \in hp(i)$ an upper bound $R_j \leq T_j$ on its WCRT exists, and, additionally, the smallest positive solution R_i to*

the following recursion is upper-bounded by T_i , then the WCRT of τ_i is upper-bounded by R_i , as defined below.

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > \xi O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - \xi O_{j_k} + A_j}{T_j} \right\rceil \xi X_{j_k} \quad (15)$$

where

$$\xi O_{j_k} = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{\ell=1}^{k-1} (\xi X_{j_\ell} + \xi G_{j_\ell}), & \text{otherwise} \end{cases}$$

and

$$A_j = R_j - X_j - \hat{G}_k$$

Proof. Let us convert the self-suspension of τ_i to computation. Then, whenever τ_i is present in the system and a higher-priority task is executing τ_i is preempted. Then the response time of a job of τ_i is maximised if the total execution time by higher-priority tasks, between its release and its completion, is maximised. Therefore we can upper-bound the WCRT of τ_i by upper-bounding the total execution time of higher-priority tasks during its activation. We are, pessimistically, going to do that by upper-bounding the execution time of every $\tau_j \in hp(i)$ and then taking the sum.

Consider some $\tau_j \in hp(i)$. Without loss of generality we will consider the canonical form where it both starts and ends with a software segment. Then, it has the form

$$[x_{j_1}, g_{j_1}, x_{j_2}, \dots, g_{j_{n(\tau_j)-1}}, x_{j_{n(\tau_j)}}]$$

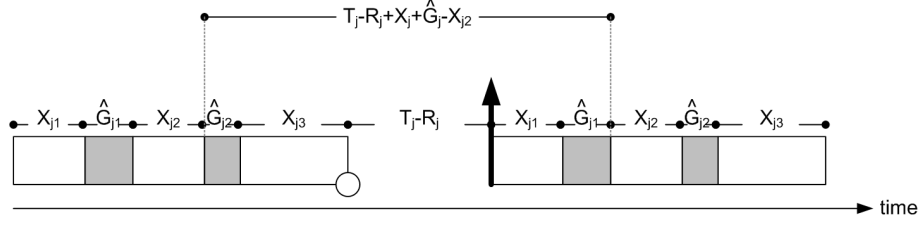
Let us consider one software segment x_{j_k} . As shown in Figure 6, from the moment that this segment completes, until another instance of the same segment (belonging to the next job of τ_j) executes for one time unit, there is a minimum time separation. Indeed:

- All subsequent self-suspensions and software segments of the original job (if any) must execute, i.e., $g_{j_k}, x_{j_{k+1}}, \dots, g_{j_{n(\tau_j)-1}}, x_{j_{n(\tau_j)}}$.
- Then, there is at least $N_j = T_j - R_j$ time units until the next job of τ_j arrives (i.e., what we earlier called the notional gap).
- Then all preceding software segments and self-suspensions (if any) of the next job of τ_j must complete, i.e., $[x_{j_1}, g_{j_1}, x_{j_2}, \dots, g_{j_{k-1}}]$

The workload generated by τ_j in any window of a given length is maximised when its execution segments execute for their respective WCETs and those belonging to jobs released after τ_i are released as early as possible whereas those belonging to a carry-in job by τ_j (if any) are released as late as possible. This implies that self-suspending regions of τ_j overlapping with that time window execute for their respective minimum suspension time. Under this scenario, it follows that the minimum time separation between time instants where two different instances of segment x_{j_k} execute is

$$\begin{aligned} & \sum_{k \leq \ell \leq n(\tau_j)-1} \hat{G}_{j_\ell} + \sum_{k < \ell \leq n(\tau_j)} X_{j_\ell} + \underbrace{T_j - R_j}_{\text{notional gap}} + \sum_{1 \leq \ell \leq k-1} X_{j_\ell} + \sum_{1 \leq \ell \leq k-1} \hat{G}_{j_\ell} \\ & = T_j - R_j + X_j + \hat{G}_j - X_{j_k} \end{aligned} \quad (16)$$

This is also illustrated in Figure 6. Note that for successive instances of x_{j_k} released no earlier than τ_i , under this worst-case scenario, the corresponding minimum time separation is $T_j - X_{j_k}$.



■ **Figure 6** Illustration of the minimum time separation between two different instances of a segment of the same task τ_j .

This means that, in the above scenario, within any time interval of length $\Delta t \leq T_j - R_j + X_j + \hat{G}_j - X_{j_k}$, the execution by segment x_{j_k} is at most X_{j_k} time units. And within any time interval of length $\Delta t = (T_j - R_j + X_j + \hat{G}_j) + M$, with $M > 0$, the total execution time by segment x_{j_k} is no more than $X_{j_k} + \lfloor \frac{M}{T_j} \rfloor X_{j_k} + \min(X_{j_k}, M - \lfloor \frac{M}{T_j} \rfloor T_j)$.

This means that, over a time interval of length Δt , the worst-case amount of execution by segment x_{j_k} is the same as the corresponding worst-case amount of execution, over an interval of length Δt , of an independent periodic non-suspending task with a WCET equal to X_{j_k} , a period of T_j and a release jitter equal to $(R_j - X_j - \hat{G}_j)$.

Then, for any particular given phasing of the interfering tasks, the response time of a job of τ_i is upper-bounded by the smallest solution to

$$R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x_{j_k} \in \tau_j} \left\lceil \frac{R_i^* + (R_j - X_j - \hat{G}_j) - O_{j_k}}{T_j} \right\rceil_0 X_{j_k} \quad (17)$$

where O_{j_k} is an offset that describes the phasings of the different segments and $\lceil \cdot \rceil_0 \stackrel{\text{def}}{=} (\max\lceil \cdot \rceil, 0)$.

Now, observe that the leftmost interfering segment of τ_j , within the interval under consideration, will not necessarily be τ_{j_1} . It could be any other segment, depending on the release offset. So, it will not hold in the general case that $O_{j_k} < O_{j_{k+1}}$, $k \in \{0, 1, n(\tau_j)\}$. Let us use introduce some notation to refer to the segments of τ_j by the order that they first appear in the time interval under consideration. So, if the β^{th} segment of τ_j is the one to appear first (i.e., leftmost), then let

$$x'_{j_1} \stackrel{\text{def}}{=} x_{j_\beta}$$

and

$$x'_{j_k} \stackrel{\text{def}}{=} x_{j_{\beta+k-1}}, \quad \forall k \in \{1, 2, \dots, n(\tau_j)\}$$

Accordingly Equation 17 can be rewritten as

$$R_i^* = C_i + \sum_{j \in hp(i)} \sum_{x'_{j_k} \in \tau_j} \left\lceil \frac{R_i^* + A'_j - O'_{j_k}}{T_j} \right\rceil_0 X'_{j_k} \quad (18)$$

where $A'_j = R_j - X_j - \hat{G}_j$ and it will hold that $O'_{j_k} < O'_{j_{k+1}}$, $k \in \{0, 1, n(\tau_j)\}$. Intuitively, the RHS is maximised when the O'_{j_k} positive offsets are minimised. And a lower-bound on each

of those is

$$\begin{aligned}
 O'_{j_1} &= 0 \\
 O'_{j_2} &= X'_{j_1} + \hat{G}'_{j_1} \\
 &\dots \\
 O'_{j_k} &= \left(\sum_{\ell=1}^{k-1} X'_{j_\ell} \right) + \left(\sum_{\ell=1}^{k-1} \hat{G}'_{j_\ell} \right), \quad k \in \{1, \dots, n(\tau_j)\}
 \end{aligned} \tag{19}$$

where g'_{j_k} is defined as the self-suspension interval immediately after segment x'_{j_k} (or, the notional gap, in the special case that x'_{j_k} is $x_{j_{n(\tau_j)}}$.)

Now compare Equation 19 with Equation 15, from the claim of this lemma. By the design of the latter equation, it holds that

$$\begin{aligned}
 \sum_{\ell=1}^k \xi X_{j_\ell} &\geq \sum_{\ell=1}^k X'_{j_\ell}, \forall j, \quad k \in \{1, 2, \dots, n(\tau_j)\} \\
 \xi O_{j_k} &\leq O'_{j_k}, \forall j, \quad k \in \{1, 2, \dots, n(\tau_j)\} \\
 A_j &= A'_j
 \end{aligned}$$

This means that the RHS of Equation 15 dominates the RHS of Equation 18, so the respective solution to the former upper-bounds the response time of τ_i under any possible combination of release phasings of higher-priority tasks. This proves the claim. \blacktriangleleft

5 Additional discussion

Priority assignment. In [2], it was claimed that the bottom-up Optimal Priority Assignment (OPA) [1] algorithm could be used in conjunction with the simple analysis. However, once the proposed fix is applied, it becomes evident that this is not the case. Namely, we now need knowledge of R_j , $\forall j \in hp(i)$ in order to compute R_i . In turn, these values depend on the relative priority ordering of tasks in $hp(i)$. This contravenes the basic principle upon which OPA relies [1].

Resource sharing. In [3], WCRT equations are augmented with blocking terms, for resource sharing under the Priority Ceiling Protocol. However, there was an omission of a term in those formulas (since those blocking terms have to be multiplied with the number of software segments of the task – or, equivalently, the number of interleaved self-suspensions plus one). This has already been acknowledged and rectified in [6], p. 101, but we repeat it here too, since this is the erratum for that paper.

Multiprocessor extension of the synthetic analysis. In Section 4 of [7], a multiprocessor extension of the synthetic analysis is sketched, assuming multiple software processors and a global fixed-priority scheduling policy. Showing whether or not this would work for the corrected analysis is a conjecture that we would like to tackle in future work.

6 Some experiments

Finally, we provide some small-scale experiments, with synthetic randomly-generated tasks in order to have some indication about:

- The performance of the corrected analysis techniques, as compared to the baseline suspension-oblivious approach.

- The extent by which the original flawed techniques were potentially optimistic.

The metric by which we compare the approaches is the scheduling success ratio. We generated⁷ hundreds of implicit-deadline task sets with $n = 6$ tasks each. The total processor utilisation ($\sum_{i=1}^n \frac{X_i}{T_i}$) of each task set did not exceed 1, in order to avoid generating task sets that would be *a priori* unschedulable. Additionally, the suspension-oblivious task set utilisation ($\sum_{i=1}^n \frac{C_i}{T_i}$) of each task set ranged between 0.6 and 1.2, with a step of 0.05. Each generated task consisted of 3 software segments and 2 interleaved self-suspending regions. For simplicity, the best-case execution time of each software segment and self-suspending region matched its worst-case execution time. Task inter-arrival times were uniformly chosen in the range 10^5 to 10^6 . For each suspension-oblivious task set utilisation (i.e., 0.6, 0.65, ..., 1.2) we generated 100 such task sets. For each target suspension-oblivious utilisation we used the `randfixedsum` function [11] to randomly generate the suspension-oblivious utilisations of the individual tasks, which could not exceed 1. Then, the suspension-oblivious execution time C_i of each task was derived by multiplying with the task inter-arrival time T_i . Subsequently, for each task, we randomly generated its X_i and G_i parameters: G_i was randomly chosen between 5% and 50% of C_i and X_i was set to $C_i - G_i$. The function `randfixedsum` was again invoked to randomly generate the execution times of the individual software segments and self-suspending regions from X_i and G_i , respectively.

Figure 7 plots the results from applying the following schedulability tests.

- **obl** The baseline suspension-oblivious approach (Equation 11).
- **simple** The simple approach from [2, 3] as corrected in Section 3 (namely Equation 3).
- **simpleUobl** Applying both “**simple**” and “**obl**” and picking the smallest WCRT.
- **synth** The “synthetic” approach from [7], already partially corrected⁸ in the Thesis [6] and as further corrected in Section 4 (namely Equation 15, that uses for A_j the value perscribed by Lemma 12).
- **synthUobl** Applying both “**synth**” and “**obl**” and picking the smallest WCRT of the two.
- **simple-bad** The original, flawed technique from [2, 3], which was proven to be *unsafe* in Section 3.
- **synth-bad** The “synthetic” analysis technique from [7], as partially corrected in [6], which was proven *unsafe* in Section 4.

The main findings from this experiment are as follows:

1. The suspension-oblivious analysis trails all other approaches in performance.
2. The benefits of the synthetic approach over the simple approach when used as a schedulability test are limited but non-negligible.
3. Combining either of the suspension-aware tests with the suspension-oblivious test offers a slight improvement in the middle region of the plot. This means that a small but not negligible number of task sets is found schedulable by the suspension-oblivious test but *not* by the suspension-aware tests.
4. The original flawed formulations of the simple and the synthetic analysis “perform” identically. The region of the plot enclosed between these curves and **synthUobl** upper-bounds the potential incidence of task sets that are in fact unschedulable but would have been erroneously deemed schedulable by those flawed tests.

⁷ We are grateful to José Fonseca, for having granted us use of his Matlab-based task generator and schedulability testing tool, which he has been developing in the context of his ongoing PhD.

⁸ With respect to the length of the “notional gap”.

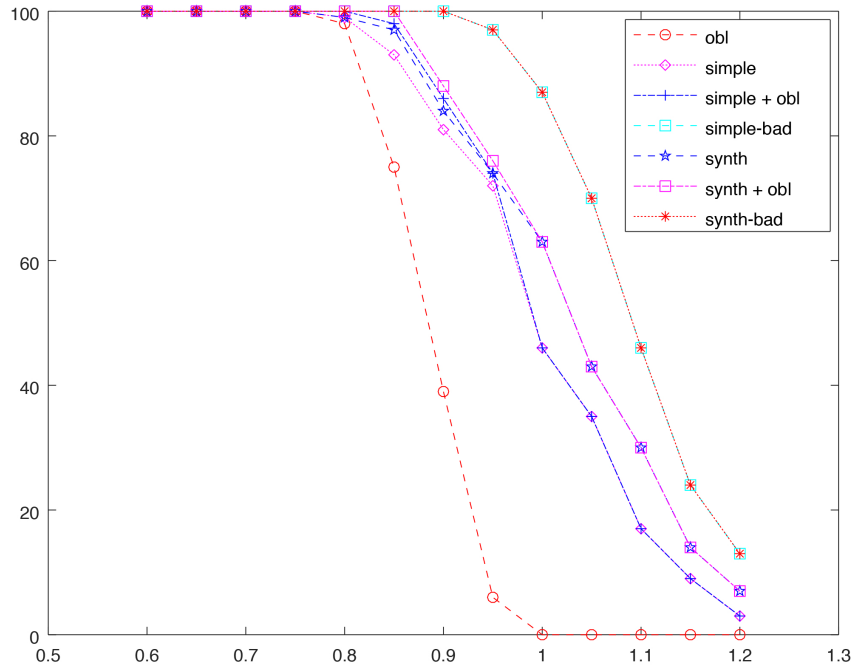


Figure 7 A comparison of the performance of different schedulability tests. The y-axis is the fraction of task sets deemed schedulable. The x-axis is the suspension-oblivious task set utilisation, defined as $\sum_{i=1}^n \frac{C_i}{T_i}$. The original flawed variants of the analysis techniques corrected by this paper are also included in the plot.

7 Conclusions

It is very unfortunate that the above flaws found their way to publication undetected. However, as obvious as they may seem in retrospect, they were not at the time, to the authors and reviewers alike. At least, this errata paper comes at a time when the topic of scheduling with self-suspensions is attracting more attention by the real-time community. Therefore we hope that it will serve as a stimulus for researchers in the area to revisit past results and scrutinise them for correctness. For more details regarding the state of the art, Chen et al [10] have recently provided high-level summaries of the general analytical methods for self-suspending tasks, the existing flaws in the literature, and potential fixes.

References

- 1 Neil C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001. doi:10.1016/S0020-0190(00)00165-4.
- 2 Neil C. Audsley and Konstantinos Bletras. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, 30 June - 2 July 2004, Catania, Italy, *Proceedings*, pages 231–238. IEEE Computer Society, 2004. doi:10.1109/ECRTS.2004.12.
- 3 Neil C. Audsley and Konstantinos Bletras. Realistic analysis of limited parallel software / hardware implementations. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, 25-28 May 2004, Toronto, Canada, pages 388–395. IEEE Computer Society, 2004. doi:10.1109/RTAS.2004.1317285.
- 4 Sanjoy K. Baruah, Deji Chen, Sergey Gorinsky, and Aloysius K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999. doi:10.1023/A:1008030427220.
- 5 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles*

- of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers, volume 3974 of *Lecture Notes in Computer Science*, pages 306–321. Springer, 2005. doi:10.1007/11795490_24.
- 6 Konstantinos Bletsas. *Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism*. PhD thesis, Dept of Computer Science, University of York, UK, 2007.
 - 7 Konstantinos Bletsas and Neil C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005)*, 17-19 August 2005, Hong Kong, China, pages 525–531. IEEE Computer Society, 2005. doi:10.1109/RTCSA.2005.48.
 - 8 Konstantinos Bletsas, Neil C. Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical report, CISTER Research Centre, ISEP, Porto, Portugal, 2015.
 - 9 Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 61–71. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.31.
 - 10 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil, Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, 2nd version, Faculty of Informatik, TU Dortmund, 2017. URL: <http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-chen-techreport-854-v2.pdf>.
 - 11 P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
 - 12 Wen-Hung Huang, Jian-Jia Chen, Husheng Zhou, and Cong Liu. PASS: priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 154:1–154:6. ACM, 2015. doi:10.1145/2744769.2744891.
 - 13 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.
 - 14 Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, December 4-6, 1996, Washington, DC, USA, pages 22–29. IEEE Computer Society, 1996. doi:10.1109/REAL.1996.563696.
 - 15 Geoffrey Nelissen, José Carlos Fonseca, Gurulingesh Raravi, and Vincent Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 80–89. IEEE Computer Society, 2015. doi:10.1109/ECRTS.2015.15.

The Semantic Foundations and a Landscape of Cache-Persistence Analyses*


Jan Reineke

Saarland University

Saarland Informatics Campus

Saarbrücken, Germany

reineke@cs.uni-saarland.de

 <http://orcid.org/0000-0002-3459-2214>

Abstract

We clarify the notion of cache persistence and contribute to the understanding of persistence analysis for caches with least-recently-used replacement.

To this end, we provide the first formal definition of persistence as a property of a trace semantics. Based on this trace semantics we introduce a semantics-based, i.e., abstract-interpretation-based persistence analysis framework.

We identify four basic persistence analyses and prove their correctness as instances of this analysis

framework.

Combining these basic persistence analyses via two generic cooperation mechanisms yields a lattice of ten persistence analyses.

Notably, this lattice contains all persistence analyses previously described in the literature. As a consequence, we obtain uniform correctness proofs for all prior analyses and a precise understanding of how and why these analyses work, as well as how they relate to each other in terms of precision.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture, Theory of computation → Caching and paging algorithms, Hardware → Safety critical systems

Keywords and Phrases caches, persistence analysis, WCET analysis

Digital Object Identifier 10.4230/LITES-v005-i001-a003

Received 2017-10-23 **Accepted** 2018-05-15 **Published** 2018-08-02

1 Introduction

Due to the large processor-memory gap, essentially all modern processors employ some form of memory hierarchy, consisting of smaller but faster memories, such as caches, on the one hand, and larger but slower memories, such as DRAM-based main memory, on the other hand. Memory hierarchies of general-purpose processors usually contain one or multiple levels of caches. Caches are small but fast hardware-managed memories that store a subset of the contents of main memory. Memory accesses that “hit” the cache may be served at a much lower latency than those accesses that “miss” the cache and as a consequence have to be served from slow main memory. The execution time of a program thus heavily depends on how effective the processor’s caches are.

For safety-critical systems, it is imperative to demonstrate before deployment that the system will always behave as intended. Many safety-critical systems are real-time systems, i.e., in order to function correctly, they have to perform their actions within limited amounts of wall-clock time. A major task in verifying a system’s real-time behavior is to analyze each software component’s worst-case execution time (WCET). Due to the large influence of caches on execution times, WCET analyses have to soundly and precisely characterize a software component’s cache behavior. To this end, various static cache analyses have been developed. Simply assuming each memory access to yield a cache miss would result in extremely pessimistic execution-time bounds.

* This work was supported by the Deutsche Forschungsgemeinschaft as part of the project PEP.



© Jan Reineke;

licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)
Leibniz Transactions on Embedded Systems, Vol. 5, Issue 1, Article No. 3, pp. 03:1–03:52



Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Listing 1** Example program motivating persistence analysis.

```
for (int i=0; i < N; i++) {
    if (read_sensor())
        a++;
    else
        b++;
}
```

Static cache analysis comes in two flavours [19]: 1. *Classifying Cache Analysis* aims to classify *individual* memory accesses as cache hits or misses. 2. *Quantitative Cache Analysis* aims to determine bounds on the number of misses resulting from a *set* of memory accesses in a program.

Classifying cache analysis for caches with least-recently-used (LRU) replacement has been well-understood since the introduction of may and must analysis by Ferdinand et al. [11, 12] in the late 1990s. May and must analysis are formalized and proven correct in the framework of abstract interpretation [3]. Both may and must analysis answer questions about the *set of reachable cache states*. For example, must analysis determines whether a memory block is guaranteed to be cached in all cache states that may be reached at a given program point. If that is the case, an access to such a block must result in a cache hit.

One instance of quantitative cache analysis is *persistence analysis*. Persistence analysis collectively considers all memory accesses in a program, or a fragment of a program such as a loop, that access the *same* memory block. Various slightly different interpretations of cache persistence exist in the literature [1, 12], which we discuss later in this article. Intuitively, a memory block is persistent if, during any possible program execution, all memory accesses referring to this block may cumulatively result in at most one cache miss.

Consider the program in Listing 1 for a motivating example. Assume that variables **a** and **b** are kept in two distinct memory blocks. Further, assume that in each loop iteration it is equally possible for the program to take the then- and the else-branch of the conditional, as the outcome of `read_sensor()` depends on external inputs. Then it is impossible to classify the memory accesses to **a** or **b** in any loop iteration as guaranteed cache hits and a WCET analysis would have to pessimistically account for misses upon all memory accesses. However, provided the cache is large enough to hold **a** and **b** simultaneously, among all memory accesses to **a** (and similarly to **b**) only the very first may result in a cache miss¹. Both **a** and **b** are persistent, and WCET analysis can safely account for at most two misses.

Various persistence analyses have been proposed in the literature starting from Mueller’s [20, 1, 31, 21] and Ferdinand’s [11, 12] work in the 1990s up until today [2, 6, 17, 23, 22, 8, 7, 32]. In our opinion, however, persistence analysis is so far not as well-understood as classifying cache analysis. In particular, even though persistence analysis clearly determines semantic properties of programs, it has never been formalized and proven correct as a semantics-based program analysis, i.e., as an abstract interpretation of an appropriate semantics in which persistence is expressible. Instead, persistence analyses have so far been described and argued correct in rather ad hoc manners. Possibly as a consequence of this lack of foundations, a flaw in one of the early persistence analyses [12] was long overlooked.

In this article we seek to fill this gap by providing a solid semantic underpinning for persistence analysis. We observe that persistence is a property of *traces* rather than *states*. Thus, semantics that capture sets of reachable states – such as those used as a basis for may and must analysis,

¹ Assuming that neither **a** nor **b** are evicted by `read_sensor()`.

and in fact most other static program analyses – are not appropriate to understand and prove correct persistence analyses. In Section 2, we define a *trace collecting semantics*, which captures all possible cache traces of a program, i.e., alternating sequences of cache states and memory accesses. On this basis, we are then able to provide the first formal definition of the various cache persistence notions found in the literature.

After discussing standard abstractions and simplifications in Section 3, we introduce a generic abstract-interpretation-based persistence analysis framework in Section 4. This framework defines the components of a persistence analysis and provides conditions on these components that are sufficient to guarantee the correctness of the persistence classifications of the analysis. As is usual in abstract interpretation, the framework uses concretization functions to capture the relation between concrete and abstract semantics. The key difference to prior work on abstract-interpretation-based cache analysis is that concretization functions map to sets of cache traces rather than sets of cache states, as persistence is a property of traces rather than states. To analyze the relative precision of two different persistence analyses, we also provide conditions on the components of two arbitrary analyses A and B that are sufficient to show that A is more precise than B , i.e., if B classifies a memory block as persistent then so does A .

A generic analysis framework is only useful if it has interesting instantiations. In Section 5, we identify four basic persistence analyses. Using the framework introduced in Section 4 we prove their correctness and determine their relative precision. Then we introduce two generic cooperation mechanisms that enable the exchange of analysis information between different persistence analyses in order to obtain more precise combined analyses. Combining the four basic persistence analyses using these two cooperation mechanisms yields a lattice of ten persistence analyses of varying precision. These ten analyses include, to the best of our knowledge, *all* persistence analyses previously described in the literature. Thus we obtain uniform correctness proofs for all these analyses and a precise understanding of how and why these analyses work, as well as how they relate to each other in terms of precision. In Section 6, we discuss how the persistence analyses from the literature map to the lattice of persistence analyses developed in Section 5.

Due to uncertainty about the memory accesses induced by loads and stores in a program, persistence analysis is more challenging for data caches than for instruction caches. We briefly describe a generic approach to data-cache persistence analysis in Section 7. Finally, we conclude the article by summarizing our results and discussing future work in Section 8.

This article may be read in different ways depending on a reader's intent:

- Readers primarily interested in understanding the intuition of the various persistence analyses may focus their attention on Section 5. To enable readers to quickly obtain a basic understanding of the state of the art, many of the correctness proofs have been moved to the appendix.
- Readers who would like to understand the semantic foundations of persistence analysis in detail, will have to study Sections 2 to 4 more carefully. Further, they may selectively read the detailed correctness proofs in the appendix. These proofs are linked from the respective theorems and lemmas of Section 5.
- Readers interested in a historical perspective may focus on Sections 5.3 and 6.

2 A Formal Definition of Cache Persistence

In this section, we provide a formal definition of cache persistence. As persistence is a semantic property of a program's execution traces, we first introduce a generic trace collecting semantics in Section 2.1. As persistence involves cache behavior, we require a semantics taking into account caches. We show how to instantiate the generic trace collecting semantics accordingly in Section 2.2. Finally, in Section 2.3 we formally capture the various notions of persistence found in the literature as properties of the semantics introduced in the two preceding sections.

2.1 Programs, Computations, Trace Collecting Semantics

A program $P = \langle \Sigma, \mathcal{I}, \mathcal{E}, \mathcal{T} \rangle$ consists of the following components:

- Σ - a set of *program states*
- $\mathcal{I} \subseteq \Sigma$ - a set of *initial states*
- \mathcal{E} - a set of *events*
- $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ - a *transition relation*, which captures how a computation of the program may step through its state space.

An *execution trace* t of P is an alternating sequence of states and events $t = \sigma_0 e_0 \sigma_1 e_1 \dots \sigma_n$ such that $\sigma_0 \in \mathcal{I}$ and for all $i \in \{0, \dots, n-1\}$, $\langle \sigma_i, e_i, \sigma_{i+1} \rangle \in \mathcal{T}$. The set of all execution traces of P is its *trace collecting semantics* $Col(P) \subseteq Traces$, where $Traces$ denotes the set of all alternating sequences of states and events. When considering terminating programs, the trace collecting semantics can be formally defined as the least fixpoint² of the *next* operator containing \mathcal{I} :

$$Col(P) = lfp_{\mathcal{I}}^{\subseteq} next$$

where *next* describes the effect of one computation step:

$$next(S) = \{t.\sigma_n e_n \sigma_{n+1} \mid t.\sigma_n \in S \wedge \langle \sigma_n, e_n, \sigma_{n+1} \rangle \in \mathcal{T}\}$$

In the definition of *next* above, $t.\sigma_n$ denotes a trace that ends in state σ_n following its prefix t . Similarly, $t.\sigma_n e_n \sigma_{n+1}$ is a trace obtained by extending $t.\sigma_n$ by $e_n \sigma_{n+1}$.

In other words, $Col(P)$ is the least solution, i.e., the smallest set of traces X that satisfies the equation $X = \mathcal{I} \cup next(X)$.

Cousot and Cousot [5] give a detailed proof of why the set of all finite execution traces of a program is indeed captured by the least fixpoint of a *next* operator as in our definition above.

2.2 Taking Caches Into Account

For reasoning about caches, we need to consider a semantics in which the state of the cache is part of the program state. To this end, the state will consist of two components: (1) the logical memory state in \mathcal{M} (representing the values of memory locations and CPU registers) and (2) the cache state in \mathcal{C} . So $\Sigma = \mathcal{M} \times \mathcal{C}$. The set of initial states is the product of the set of initial memory states and the set of initial cache states, i.e., $\mathcal{I} = \mathcal{I}_{\mathcal{M}} \times \mathcal{I}_{\mathcal{C}}$.

We define the transition relation on this domain based on four functions that model the evolution of the logical memory and the cache as well as their interaction:

1. The *memory update* is a function $update_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{M}$ that captures the logical memory state the system transitions into from a given logical memory state.
2. The *memory effect* is a function $eff_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{E}_{\mathcal{M}}$ that determines the memory block, if any, that is accessed when transitioning from a given logical memory state. We denote the set of memory blocks by \mathcal{B} . Thus, the set of memory events is defined as $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\perp\}$, where \perp denotes that no memory block is accessed.
3. The *cache update* is a function $update_{\mathcal{C}}: \mathcal{C} \times \mathcal{E}_{\mathcal{M}} \rightarrow \mathcal{C}$ that determines the successor cache state given a memory effect.

² Here, and later, we denote by $lfp_{\mathcal{I}}^{\subseteq} next$, the least fixpoint of the function *next* that is greater than or equal to \mathcal{I} . This is the same as the least fixpoint of the function $next_{\mathcal{I}}(X) := \mathcal{I} \cup next(X)$.

4. The *cache effect* is a function $eff_C: \mathcal{C} \times \mathcal{E}_M \rightarrow \mathcal{E}_C$ that determines whether or not the memory access results in a cache hit or a cache miss. Thus, the set of cache events is defined as $\mathcal{E}_C = \{hit, miss, \perp\}$, where \perp is used when the memory effect is \perp , and so no memory block is actually accessed.

Events are pairs of memory events and cache events, i.e., $\mathcal{E} = \mathcal{E}_M \times \mathcal{E}_C$.

The definition of persistence as a property of traces in Section 2.3, as well as the persistence analysis framework developed in Sections 3 and 4 applies to arbitrary replacement policies. Particular replacement policies can be captured by appropriately defining $update_C$ and eff_C . Below, we provide definitions of these two functions for the LRU strategy, denoted as $update_C^{LRU}$ and eff_C^{LRU} . This is because all the persistence analysis instances that we will introduce in Section 5 apply to caches with LRU replacement. In the following, whenever we make statements that hold for arbitrary caches, we use $update_C$ and eff_C . Whenever our statements refer to LRU, we use $update_C^{LRU}$ and eff_C^{LRU} .

Upon a cache miss, LRU replaces the least-recently-used memory block. To this end, it tracks the ages of memory blocks within each cache set, where the youngest block has age 0 and the oldest cached block has age $k - 1$, where k is the associativity of the cache. Thus, the state of the cache can be modeled as a function that assigns an age to each memory block, where non-cached blocks are assigned age k . For simplicity of exposition, we consider a fully-associative cache³, in other words, all blocks map to the same cache set.

$$\mathcal{C} := \{c \in \mathcal{B} \rightarrow A \mid \forall a, b \in \mathcal{B} : a \neq b \Rightarrow (c(a) \neq c(b) \vee c(a) = c(b) = k)\},$$

where $A := \{0, \dots, k - 1, k\}$ is the set of ages. The constraint encodes that no two cached blocks can have the same age. For readability we omit the additional constraint that blocks of non-zero age are preceded by other blocks, i.e. that cache sets do not contain “holes”.

Below, we define the cache update and the cache effect for the cases where a memory access occurs, i.e., for the subset \mathcal{B} of their domain $\mathcal{E}_M = \mathcal{B} \cup \{\perp\}$. Both cache update and cache effect are naturally extended to the case where no memory access occurs.

The cache update for LRU is given by

$$update_C^{LRU}(c, b) := \lambda b' \in \mathcal{B}. \begin{cases} 0 & : b' = b, \\ c(b') + 1 & : c(b') < c(b), \\ c(b') & : c(b') \geq c(b). \end{cases}$$

The accessed block attains age 0 (case 1), blocks younger than the accessed block age by 1 (case 2), and the ages of other blocks are not affected (case 3).

The cache effect captures that a hit occurs whenever the age of the accessed block is less than the cache’s associativity:

$$eff_C^{LRU}(c, b) := \begin{cases} hit & : c(b) < k, \\ miss & : \text{else.} \end{cases}$$

Both $update_C^{LRU}$ and eff_C^{LRU} are naturally extended to the case where no memory access occurs. Then, the cache state remains unchanged and the cache effect is \perp .

³ Set-associative caches as they are typically found in actual caches can be considered as arrays of fully-associative caches. Thus, analyses that apply to fully-associative caches can be lifted to set-associative caches in a rather straightforward manner.

With this, we can now connect the components and obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ by

$$\mathcal{T} = \{ \langle \langle m, c \rangle, \langle e_m, e_c \rangle, \langle m', c' \rangle \rangle \mid m' = \text{update}_{\mathcal{M}}(m) \wedge e_m = \text{eff}_{\mathcal{M}}(m) \\ \wedge c' = \text{update}_{\mathcal{C}}(c, e_m) \wedge e_c = \text{eff}_{\mathcal{C}}(c, e_m) \},$$

which formally captures the asymmetric relationship between caches, logical memories, and events:

- The *memory update* determines the next memory states: $m' = \text{update}_{\mathcal{M}}(m)$, and the *memory effect* determines the memory block, if any, that is accessed on the transition: $e_m = \text{eff}_{\mathcal{M}}(m)$.
- Based on the memory effect, the *cache update* determines the next cache state: $c' = \text{update}_{\mathcal{C}}(c, e_m)$, and the *cache effect* determines whether or not the current memory access results in a hit or a miss: $e_c = \text{eff}_{\mathcal{C}}(c, e_m)$.

2.3 Persistence as a Property of Traces

Given a trace collecting semantics that takes caches into account as defined in the previous two sections, we are now ready to formally capture multiple notions of persistence found in the literature. Persistence is a property of traces, and thus, the following predicates determine for a given trace τ and a memory block b , whether b is persistent in τ according to a particular notion of persistence.

The most liberal notion of persistence is that a persistent block may cause at most one miss:

$$\text{AtMostOneMiss}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := |\{i \mid e_i = \langle b, \text{miss} \rangle\}| \leq 1 \quad (1)$$

A stronger notion of persistence is that only the very first access to a persistent block may result in a miss [1]:

$$\text{FirstAccessIsAMiss}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := \\ \forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j < i : (e_j \neq \langle b, \text{hit} \rangle \wedge e_j \neq \langle b, \text{miss} \rangle). \quad (2)$$

This condition is stronger, i.e., $\text{FirstAccessIsAMiss}(\tau, b)$ implies $\text{AtMostOneMiss}(\tau, b)$ but not vice versa, as $\text{AtMostOneMiss}(\tau, b)$ is equivalent to $\forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j < i : (e_j \neq \langle b, \text{miss} \rangle)$.

Another stronger notion of persistence is that, after a block has been brought into the cache via a miss, it is not evicted from the cache anymore [12]:

$$\text{NoEviction}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := \forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j > i : b \in \sigma_j, \quad (3)$$

where $b \in \sigma_l$ means that b is cached in state σ_l , i.e., it is an abbreviation for $\sigma_l = \langle m_l, c_l \rangle \wedge \text{eff}_{\mathcal{C}}(c_l, b) = \text{hit}$. Clearly, persistence according to (3) also implies persistence according to (1).

The above definitions refer to individual traces. We say that a memory block b is persistent in program P , if it is persistent in all traces of P 's trace collecting semantics:

► **Definition 1** (Persistence in a Program). *Memory block b is persistent in program P , if*

$$\forall \tau \in \text{Col}(P) : \text{AtMostOneMiss}(\tau, b).$$

We choose to use the most liberal notion of persistence in Definition 1, because it corresponds to the property that is being exploited in the later phases of WCET analysis. As we will see, all persistence analyses introduced in Section 5 are in fact based on the stronger *NoEviction* notion expressed by (3). It is conceivable though that future persistence analyses will take advantage of the more liberal *AtMostOneMiss* notion to classify more memory blocks as persistent.

3 Preliminaries: Standard Abstractions and Simplifications

The trace collecting semantics as defined above is not practically computable. In this section, we discuss two very common abstractions that lead to an abstract semantics that is closer to being computable. Based on the resulting sticky-collecting semantics, we then develop further abstractions in Sections 4 and 5 that allow to prove the persistence of memory blocks in practice.

3.1 Control Flow Graph Abstraction

In contrast to data accesses, instruction accesses depend solely on the flow of control through the program and are thus much easier to predict. As the focus of this article is on the analysis of the cache behavior rather than the analysis of the memory accesses generated by a program, we initially limit ourselves to the analysis of instruction caches. Later, in Section 7 we discuss how to lift this restriction. Thus we abstract the state of the memory, in \mathcal{M} , to the program location, in \mathcal{L} , that the program is currently at.

A common abstraction of a program P is its control flow graph $G_P = \langle \mathcal{L}, E, i \rangle$, where

- the nodes in \mathcal{L} represent program locations,
- the edges in $E \subseteq \mathcal{L} \times \mathcal{L}$ represent possible control flow, and
- $i \in \mathcal{L}$ represents the start node, which has no incoming edges.

The set of edges E can be seen as an abstraction of the memory update $update_{\mathcal{M}}$. While $update_{\mathcal{M}}$ is a function, E is a relation, because the successor location may depend on values of registers that have been abstracted away. Similarly, let $eff_{\mathcal{L}} : E \rightarrow \mathcal{B}$ capture the memory block holding the instruction that needs to be fetched when moving from one program location to another. This corresponds to the memory effect $eff_{\mathcal{M}}$. As we limit ourselves to instruction accesses, which are precisely determined by control flow, there is no loss in precision moving from $eff_{\mathcal{M}}$ to $eff_{\mathcal{L}}$. Also, each edge corresponds to exactly one memory access, and so we do not need to consider the trivial case that no memory access is performed upon a transition. With this abstraction, the set of states is $\Sigma_{ins} = \mathcal{L} \times \mathcal{C}$, and the set of initial states is $\mathcal{I}_{ins} = \{i\} \times \mathcal{I}_{\mathcal{C}}$.

Based on these notions, we obtain the following global transition relation \mathcal{T}_{ins} :

$$\mathcal{T}_{ins} = \{ \langle \langle l, c \rangle, \langle b, h \rangle, \langle l', c' \rangle \rangle \mid \langle l, l' \rangle \in E \wedge b = eff_{\mathcal{L}}(l, l') \wedge c' = update_{\mathcal{C}}(c, b) \wedge h = eff_{\mathcal{C}}(c, b) \},$$

which yields the abstraction P_{ins} of P : $P_{ins} = \langle \Sigma_{ins}, \mathcal{I}_{ins}, \mathcal{E}, \mathcal{T}_{ins} \rangle$.

One could formally relate $Col(P_{ins})$ and $Col(P)$ by concretization and abstraction functions and derive correctness conditions on E and $eff_{\mathcal{L}}$, but we omit this here⁴ and assume that the control flow graph is the starting point of the analyses presented below, as is common in the literature [1, 20, 31, 11, 12, 21, 2, 6, 17, 8, 7, 32].

We note, however, that more precise results can be obtained if persistence analysis is carried out on more precise abstractions of the program's memory access behavior, which can be obtained by e.g. trace partitioning [28].

3.2 Abstraction from Locations in Traces

As we can see from Definition 1, to determine whether a block is persistent it suffices to inspect the cache and memory effects of the trace collecting semantics. We thus further abstract the trace collecting semantics to a semantics that only maintains traces of cache and memory effects, forgetting about the intermediate locations. We denote the set of such traces by $CacheTraces$.

⁴ We consider the problem of soundly abstracting a program's memory access behavior to be distinct from the problem of cache persistence analysis based on such an abstraction, which is the topic of this article. See Section 4.2 in [10] for a concretization function relating $Col(P_{ins})$ and $Col(P)$.

This *sticky cache trace collecting semantics*⁵, $StickyCol(P_{ins}) : \mathcal{L} \rightarrow 2^{CacheTraces}$, captures the set of traces of cache states and cache and memory effects that may reach a given program location. It is defined as the least fixpoint of $next_{ins}$, defined below, including $Init$:

$$StickyCol(P_{ins}) := lfp_{Init}^{\leq} next_{ins},$$

where $Init = \lambda l. (l = i ? \mathcal{I}_C : \emptyset)$, the partial order \leq denotes the point-wise comparison, i.e., $S \leq T := \forall l \in \mathcal{L} : S(l) \subseteq T(l)$, which induces the join $S \vee T := \lambda l \in \mathcal{L}. S(l) \cup T(l)$, and $next_{ins}$ is defined as follows:

$$\begin{aligned} next_{ins}(S) &= \lambda l' \in \mathcal{L}. \bigcup_{(l, l') \in E} \{t.c \langle l, c \rangle, e, \langle l', c' \rangle \mid t.c \in S(l) \wedge \langle \langle l, c \rangle, e, \langle l', c' \rangle \rangle \in \mathcal{T}_{ins}\} \\ &\stackrel{\text{Def. of } \mathcal{T}_{ins}}{=} \lambda l' \in \mathcal{L}. \bigcup_{(l, l') \in E} \{t.c \langle b, h \rangle c' \mid t.c \in S(l) \wedge b = eff_{\mathcal{L}}(l, l') \\ &\quad \wedge h = eff_C(c, b) \wedge c' = update_C(c, b)\} \end{aligned}$$

The function $next_{ins}$ defined above captures how the set of cache traces reaching location l' is recursively determined by the set of cache traces reaching predecessor locations of l' and the memory accesses on the edges from the predecessors to l' .

We can relate the sticky cache trace collecting semantics to the corresponding trace collecting semantics by the concretization function γ_{ins} :

$$\gamma_{ins}(S) = \{\langle l_0, c_0 \rangle e_0 \langle l_1, c_1 \rangle \dots e_{n-1} \langle l_n, c_n \rangle \in Traces \mid \forall i \leq n : c_0 e_0 \dots c_i \in S(l_i)\} \quad (4)$$

It can be shown that $Col(P_{ins}) \subseteq \gamma_{ins}(StickyCol(P_{ins}))$.

4 A Generic Persistence Analysis Framework

The sticky cache trace collecting semantics defined above associates sets of cache traces with each program point. These traces may be arbitrarily long and there may be infinitely many associated with a single program point. Thus, an effective analysis needs to further abstract from this semantics, by representing potentially infinite sets of cache traces in a finite fashion. Before discussing particular abstractions of cache traces in Section 5, we show in this section how to lift any such abstraction to a sound persistence analysis in Sections 4.1 and 4.2, and in Section 4.3 we show how to characterize the relative precision of different persistence analyses.

4.1 Sound Cache Trace Abstractions

Before formally defining cache trace abstractions, let us informally state their components. First, we need a set of abstract traces, which will be used by the analysis in place of sets of concrete cache traces. To enable a proof of correctness, these abstract traces need to be related to sets of concrete traces by a concretization function, which specifies the set of concrete traces represented by each abstract trace. Usually no information is available about the initial state of the cache. Thus, the abstract traces need to contain an initial abstract trace that represents all possible initial cache states. To combine analysis information at control flow joins, a join operator on abstract traces is required. The core of a cache trace abstraction is the abstract update function, which captures the effect of a memory access on abstract cache traces. Finally, a persistence classification function is required to determine whether a memory block is persistent in all concrete cache traces represented by an abstract trace. These components yield the following definition of a cache trace abstraction.

⁵ We call this semantics “sticky” because it sticks sets of traces to each program location.

► **Definition 2** (Cache Trace Abstraction). A cache trace abstraction is a tuple

$$A = \langle C_A^\#, \gamma_A, \widehat{\mathcal{I}}_A, \sqsubseteq_A, \sqcup_A, \text{update}_A^\#, \text{classify}_A^\# \rangle,$$

consisting of the following components:

1. $C_A^\#$, a set of abstract traces,
2. $\gamma_A : C_A^\# \rightarrow 2^{\text{CacheTraces}}$, a concretization function, which specifies the set of concrete cache traces represented by each abstract trace,
3. $\widehat{\mathcal{I}}_A \in C_A^\#$, an abstract initial trace that represents all possible initial cache states,
4. \sqsubseteq_A , a partial order on $C_A^\#$, such that $\langle C_A^\#, \sqsubseteq_A \rangle$ is a complete lattice [9],
5. \sqcup_A , a join operator on abstract traces⁶,
6. $\text{update}_A^\# : C_A^\# \times \mathcal{B} \rightarrow C_A^\#$, an abstract update function,
7. $\text{classify}_A^\# : C_A^\# \times \mathcal{B} \rightarrow \mathbb{B}$, a persistence classification function.

We will introduce requirements on the components of a cache trace abstraction in Theorems 3 and 4 that together imply correct analysis results.

Given a cache trace abstraction A we can define the abstract next operator as follows:

$$\text{next}_{ins,A}^\#(\widehat{S}) = \lambda l' \in \mathcal{L}. \bigsqcup_{\langle l, l' \rangle \in E} \{ \text{update}_A^\#(\widehat{S}(l), b) \mid b = \text{eff}_{\mathcal{L}}(l, l') \}$$

Intuitively, the abstract next operator captures how the analysis state at location l' depends on the analysis state at predecessor locations and the abstract update function of the cache trace abstraction.

Based on the abstract initial trace $\widehat{\mathcal{I}}_A$, we can define the initial analysis state $\widehat{\text{Init}}_A := \lambda l \in \mathcal{L}. (l = i ? \widehat{\mathcal{I}}_A : \perp_A)$ analogously to the definition of Init earlier. The *abstract sticky trace collecting semantics* $\widehat{\text{StickyCol}}_A$ is then defined as the least fixpoint of $\text{next}_{ins,A}^\#$ greater than $\widehat{\text{Init}}_A$:

$$\widehat{\text{StickyCol}}_A(P_{ins}) = \text{lfp}_{\widehat{\text{Init}}_A}^{\sqsubseteq_A} \text{next}_{ins,A}^\#, \quad (5)$$

where \sqsubseteq_A is lifted to functions as follows: $\widehat{S} \sqsubseteq_A \widehat{T} := \forall l \in \mathcal{L} : \widehat{S}(l) \sqsubseteq_A \widehat{T}(l)$.

In order for the abstract sticky trace collecting semantics to be well-defined, we require the abstract update function to be monotone in the first parameter. This guarantees that the abstract $\text{next}_{ins,A}^\#$ operator is monotone. Then, the Knaster-Tarski fixpoint theorem [9], which is reproduced in Theorem 30 in the appendix, guarantees the existence of a unique least fixpoint. Note that requiring the abstract update function to be monotone is not a restriction: the best abstract update function [4] for a given abstraction is always monotone.

If the partial order \sqsubseteq_A on abstract traces satisfies the ascending chain condition [18], i.e., if there are no infinite ascending chains of abstract traces, then $\widehat{\text{StickyCol}}_A(P_{ins})$ can effectively be computed by fixpoint iteration [3]. In Section 4.2 we recapitulate a variant of the worklist algorithm [24, 29] to more efficiently compute $\widehat{\text{StickyCol}}_A(P_{ins})$.

For the analysis results to be correct, the abstract semantics should soundly approximate its concrete counterpart. This is the case if the cache trace abstraction satisfies these three conditions, which are formalized in the following theorem: 1. The abstract initial trace needs to represent all possible concrete initial cache states. 2. The concretization function needs to be monotone in \sqsubseteq_A . 3. The abstract update function needs to overapproximate the concrete update of cache states.

⁶ Note that in a complete lattice $\langle L, \sqsubseteq \rangle$ the partial order \sqsubseteq uniquely defines the join operator \sqcup . Vice versa, a given join operator uniquely defines a corresponding partial order. Nevertheless, we explicitly provide both partial order and join operator here and in the following.

► **Theorem 3** (Soundness of Persistence Analysis*). *If the cache trace abstraction A satisfies the following conditions:*

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A), \quad (6)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\# : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \gamma_A(\widehat{S}) \subseteq \gamma_A(\widehat{T}), \quad (7)$$

$$\begin{aligned} \forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{update}_A^\#(\widehat{S}, b)). \end{aligned} \quad (8)$$

Then, its abstract semantics soundly approximates its concrete counterpart:

$$\text{StickyCol}(P_{ins}) \leq \gamma_A(\widehat{\text{StickyCol}}_A(P_{ins})), \quad (9)$$

where γ_A is lifted to functions as follows: $\gamma_A(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_A(\widehat{S}(l))$.

The proof to this theorem, and all other proofs that are not provided in the main part of the article, can be found in the appendix. Whenever a lemma or theorem is not immediately followed by its proof, the theorem's name is marked with a \star and serves as a link to the corresponding proof in the appendix. Similarly, theorems reproduced without proof in the appendix link back to their proofs in the main part.

As a consequence, the abstract sticky trace collecting semantics also soundly approximates the trace collecting semantics:

$$\text{Col}(P_{ins}) \subseteq \gamma_{ins}(\text{StickyCol}(P_{ins})) \subseteq \gamma_{ins}(\gamma_A(\widehat{\text{StickyCol}}_A(P_{ins}))).$$

The following theorem gives a condition on the persistence classification function that implies correct persistence classifications of memory blocks:

► **Theorem 4** (Soundness of Persistence Classification*). *If the cache trace abstraction A satisfies conditions (6), (7), (8) from Theorem 3, and $\text{classify}_A^\#$ satisfies*

$$\begin{aligned} \forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \text{classify}_A^\#(\widehat{S}, b) \Rightarrow \\ \forall c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \end{aligned} \quad (10)$$

then $\text{classify}_A^\#(P_{ins}, b) := \forall l \in \mathcal{L} : \text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l), b)$ implies the persistence of memory block b in program P_{ins} .

The condition on $\text{classify}_A^\#$ in the theorem above is based on the *NoEviction* persistence notion from (3). It could be replaced by weaker conditions corresponding to the *FirstMiss* or the *AtMostOneMiss* persistence notions from (1) and (2). However, all persistence analyses we are aware of can be shown correct using the *NoEviction* notion.

4.2 Computing the Abstract Sticky Trace Collecting Semantics

Algorithm 1 shows how to compute the abstract sticky trace collecting semantics of a program for a given cache trace abstraction by Kleene iteration. The algorithm computes an increasing sequence of analysis states $\widehat{S}_0 \sqsubseteq_A \widehat{S}_1 \sqsubseteq_A \dots$ starting from the initial analysis state $\widehat{S}_0 = \widehat{\text{Init}}_A$, until a fixpoint is reached. This process is guaranteed to terminate if the complete lattice of abstract traces satisfies the ascending chain condition [18].

Algorithm 2 shows a *worklist algorithm* [24, 29]. The goal of worklist algorithms is to be more efficient than the Kleene iteration by avoiding redundant recomputations of parts of the abstract next operator $\text{next}_{ins, A}^\#$. Specifically, $\text{next}_{ins, A}^\#$ involves the application of the abstract update function to each edge in the control flow graph. However, $\text{update}_A^\#(\widehat{S}(l), b)$ in (5) will only deliver a different value than in the previous iteration if $\widehat{S}(l)$ has changed in the meantime.

Algorithm 1: Kleene Iteration**Input** : Control Flow Graph $G_P = \langle \mathcal{L}, E, i \rangle$ and Cache Trace Abstraction A **Output** : Abstract Sticky Trace Collecting Semantics $\widehat{StickyCol}_A(P_{ins})$

```

1  $\widehat{S}_0 := \widehat{Init}_A$ 
2  $i := 0$ 
3 repeat
4    $\widehat{S}_{i+1} := \widehat{Init}_A \sqcup_A next_{ins,A}^\#(\widehat{S}_i)$ 
5    $i := i + 1$ 
6 until  $\widehat{S}_i = \widehat{S}_{i-1}$ 
7 return  $\widehat{S}_i$ 

```

Algorithm 2: Worklist Algorithm**Input** : Control flow Graph $G_P = \langle \mathcal{L}, E, i \rangle$ and Cache Trace Abstraction A **Output** : Abstract Sticky Trace Collecting Semantics $\widehat{StickyCol}_A(P_{ins})$

```

1  $\widehat{S} := \widehat{Init}_A$ 
2  $worklist := \{ \langle i, l \rangle \in E \}$ 
3 while exists  $\langle l, l' \rangle \in worklist$  do
4   remove  $\langle l, l' \rangle$  from  $worklist$ 
5    $t := update_A^\#(\widehat{S}(l), eff_{\mathcal{L}}(l, l'))$ 
6   if  $t \not\sqsubseteq_A \widehat{S}(l')$  then
7      $\widehat{S}(l') := t \sqcup_A \widehat{S}(l')$ 
8      $worklist := worklist \cup \{ \langle l', l'' \rangle \in E \}$ 
9   end
10 end
11 return  $\widehat{S}$ 

```

Worklist algorithms maintain a set of edges, stored in the variable `worklist` algorithm, whose source locations have been modified, and which thus have to be (re-)evaluated. Initially, only those edges emanating from the start node i of the control flow graph need to be evaluated. Thus `worklist` is initialized to edges emanating from i in line 2 of the algorithm. While there are edges to (re-)evaluate, the algorithm picks one of these edges, and removes it from `worklist` (lines 3 and 4). If the value $update_A^\#(\widehat{S}(l), eff_{\mathcal{L}}(l, l'))$ computed (line 5) for an edge is not covered (line 6) by the abstract trace $\widehat{S}(l')$ stored for its target location, then $\widehat{S}(l')$ is updated (line 7), and all edges emanating from l' need to be recomputed, and are thus added to `worklist` (line 8).

The performance of worklist algorithms depends on the *iteration strategy*. If the worklist contains multiple edges, the iteration strategy determines which edge to pick next. Nielson et al. [24, Chapter 6.1] discuss various iteration strategies and their performance characteristics.

4.3 On the Relative Precision of Different Cache Trace Abstractions

In Section 5 we introduce various basic approaches to persistence analysis as well as ways of combining basic approaches to obtain more precise combined analyses. In addition to proving these approaches correct, we also characterize their relative precision, based on the following notion of precision:

► **Definition 5** (Precision). *Given two cache trace abstractions A and B , we say that A is at least as precise as B , denoted by $A \succeq B$, if A classifies each block as persistent that B classifies as persistent:*

$$\forall P_{ins}, \forall b : \text{classify}_B^\#(P_{ins}, b) \Rightarrow \text{classify}_A^\#(P_{ins}, b).$$

We say that A is more precise than B , denoted by $A \succ B$, if $A \succeq B$, but $B \not\succeq A$. If neither $A \succeq B$ nor vice versa, we say that A and B are incomparable.

Note that \succeq is a non-strict partial order, i.e., it is reflexive, antisymmetric, and transitive. Its strict counterpart \succ is a strict partial order, i.e., it is irreflexive, asymmetric, and transitive.

One way of showing $A \succeq B$ is to show that B is a sound approximation of A , just like we show that individual domains soundly approximate the concrete trace collecting semantics. This approach yields the following two theorems, which mirror Theorems 3 and 4:

► **Theorem 6** (Approximation of Abstract Semantics*). *Given two cache trace abstractions A and B , and a function $\gamma_{B \rightarrow A} : C_B^\# \rightarrow C_A^\#$ that satisfies the following conditions:*

$$\widehat{\mathcal{I}}_A \subseteq \gamma_{B \rightarrow A}(\widehat{\mathcal{I}}_B), \quad (11)$$

$$\forall \widehat{S}, \widehat{T} \in C_B^\# : \widehat{S} \sqsubseteq_B \widehat{T} \Rightarrow \gamma_{B \rightarrow A}(\widehat{S}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{T}), \quad (12)$$

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}, b)). \quad (13)$$

Then, B 's abstract semantics soundly approximates its more concrete counterpart:

$$\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})), \quad (14)$$

where $\gamma_{B \rightarrow A}$ is lifted to the abstract sticky trace collecting semantics as follows:

$$\gamma_{B \rightarrow A}(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_{B \rightarrow A}(\widehat{S}(l)).$$

► **Theorem 7** (Precision). *Given cache trace abstractions A, B and a function $\gamma_{B \rightarrow A}$ that satisfies conditions (11), (12), and (13) from Theorem 6, and further*

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{classify}_B^\#(\widehat{S}, b) \Rightarrow \text{classify}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b), \quad (15)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\#, b \in \mathcal{B} : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \left(\text{classify}_A^\#(\widehat{T}, b) \Rightarrow \text{classify}_A^\#(\widehat{S}, b) \right). \quad (16)$$

Then, A is at least as precise as B , i.e., $A \succeq B$.

Proof. From Theorem 6 we have that $\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))$, which by definition is equivalent to $\forall l \in \mathcal{L} : \widehat{\text{StickyCol}}_A(P_{ins})(l) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l))$.

Assume $\text{classify}_B^\#(P_{ins}, b)$ for an arbitrary b :

$$\begin{aligned} \text{classify}_B^\#(P_{ins}, b) &\Leftrightarrow \forall l \in \mathcal{L} : \text{classify}_B^\#(\widehat{\text{StickyCol}}_B(P_{ins})(l), b) \\ &\stackrel{(15)}{\Rightarrow} \forall l \in \mathcal{L} : \text{classify}_A^\#(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l)), b) \\ &\stackrel{(*)}{\Rightarrow} \forall l \in \mathcal{L} : \text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l), b) \\ &\Leftrightarrow \text{classify}_A^\#(P_{ins}, b) \end{aligned}$$

(*) follows from (16) and the fact that $\forall l \in \mathcal{L} : \widehat{\text{StickyCol}}_A(P_{ins})(l) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l))$. ◀

Proving that a sound cache trace abstraction A is at least as precise as cache trace abstraction B also proves B 's soundness:

► **Theorem 8** (Soundness of Persistence Classification). *Given two cache trace abstractions A and B . If A is sound, and A is at least as precise as B , then B is also sound.*

Proof. We need to show that $\text{classify}_B^\#(P_{\text{ins}}, b) := \forall l \in \mathcal{L} : \text{classify}_B^\#(\widehat{\text{StickyCol}}_B(P_{\text{ins}})(l), b)$ implies the persistence of memory block b in program P_{ins} .

Assume $\text{classify}_B^\#(P_{\text{ins}}, b)$. Because A is at least as precise as B this implies $\text{classify}_A^\#(P_{\text{ins}}, b)$. As A is sound, this implies the persistence of memory block b in program P_{ins} . ◀

While we give independent soundness proofs for all persistence analyses introduced in Section 5, in some cases the relative precision results constitute alternative soundness proofs based on the above theorem.

5 Instantiations of the Analysis Framework: Abstractions of Cache Traces

In this section, we explain and prove correct existing and new abstractions of cache traces for cache-persistence analysis.

Paraphrasing the soundness condition from Theorem 4, a memory block is persistent, if it is guaranteed to remain in the cache *in case* it has been accessed. This suggests that persistence analyses should maintain information about memory blocks *under the condition* that the memory blocks have been accessed. All sound persistence analyses can be seen as maintaining such information as we will see below.

Before describing particular analysis domains let us characterize under which conditions a memory block is guaranteed to be cached under LRU replacement. To this end, we first define the set `LRUCACHETRACES`, which consists of all cache traces that are possible under LRU replacement, assuming an arbitrary initial cache state and an arbitrary sequence of memory access:

$$\text{LRUCACHETRACES} := \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid c_0 \in \mathcal{C} \wedge \forall i, 0 \leq i < n : b_i \in \mathcal{B} \wedge c_{i+1} = \text{update}_C^{\text{LRU}}(c_i, b_i) \wedge h_i = \text{eff}_C^{\text{LRU}}(c_i, b_i)\} \quad (17)$$

The following lemma precisely captures when a memory block is guaranteed to be cached under LRU replacement:

► **Lemma 9** (Persistence under LRU*). *Consider an arbitrary cache trace $c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$. Then $c_n(b_0) < k$, if $|\{b_i \mid 0 \leq i < n\}| \leq k$.*

In other words, after a block b is accessed, this block is guaranteed to be cached as long as less than k distinct conflicting blocks have been accessed.

In Section 5.1, we discuss basic abstractions for persistence analysis. These abstractions either bound the *number* of conflicting blocks or overapproximate the *set* of conflicting blocks for each memory block. As we will see, these two approaches are incomparable, i.e., neither of the two dominates the other in terms of precision.

In Section 5.2, we then discuss how to combine these basic abstractions in order to obtain more precise analyses. By exchanging information during analysis time, such combinations go beyond simply running two incomparable analyses in parallel. As a consequence they may classify memory blocks as persistent that none of the basic abstractions would be able to classify as persistent on its own.

5.1 Basic Abstractions

5.1.1 Global-CS: Global May-Conflict Set

If at most k distinct memory blocks may be accessed in a given cache set, then, following Lemma 9, none of these blocks may be evicted after entering the cache. Thus, the *global may-conflict set*

analysis, abbreviated to *Global-CS*, overapproximates the set of memory blocks that may have been accessed, and so its abstract traces are sets of memory blocks:

$$C_{Global-CS}^{\#} := 2^{\mathcal{B}} \quad (18)$$

An abstract trace \widehat{S} represents all concrete cache traces that may be formed by accessing blocks from the set \widehat{S} :⁷

$$\gamma_{Global-CS}(\widehat{S}) := \text{LRUCACHETRACES} \cap \{c_0 \langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S}\} \quad (19)$$

At program start, no accesses have yet been performed, and so the initial abstract trace is the empty set, which by $\gamma_{Global-CS}$ represents exactly cache traces of length 0, in other words, all initial cache states:

$$\widehat{\mathcal{I}_{Global-CS}} := \emptyset \quad (20)$$

In order to soundly approximate all memory blocks that *may* have been accessed, abstract traces are joined by taking their union:

$$\widehat{S} \sqsubseteq_{Global-CS} \widehat{T} \Leftrightarrow \widehat{S} \subseteq \widehat{T} \quad \widehat{S} \sqcup_{Global-CS} \widehat{T} := \widehat{S} \cup \widehat{T} \quad (21)$$

Upon a memory access, the accessed block is simply added to the abstract trace:

$$\text{update}_{Global-CS}^{\#}(\widehat{S}, b) := \widehat{S} \cup \{b\} \quad (22)$$

Following Lemma 9, as long as at most k memory blocks have been accessed, *any* block must still be cached *if* it has been accessed:

$$\text{classify}_{Global-CS}^{\#}(\widehat{S}, b) := b \in \widehat{S} \Rightarrow |\widehat{S}| \leq k \quad (23)$$

See Figure 1 for a small example of the *Global-CS* analysis. The figure shows the fixpoint of the set of equations determined by the update and join functions of the analysis on the given control flow graph. At any point in the loop, each of the blocks v, w , and x may have been accessed. In a cache of associativity 3 or higher these blocks would all be declared as persistent by *Global-CS*. On the other hand, while w and x are persistent even in a cache of associativity 2, the analysis is unable to detect this.

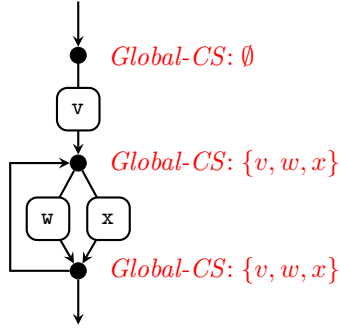
► **Theorem 10** (Soundness of Global May-Conflict Set*). *Global-CS is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{Global-CS}^{\#} = 2^{\mathcal{B}}$ is also finite, and thus it satisfies the ascending chain condition [18, 9], which guarantees termination of the fixpoint iteration to compute the abstract semantics. Following Davey and Priestley [9], we define the length of an ascending chain as the number of elements of the chain minus one. The length of a chain thus corresponds to the number of steps a fixpoint iteration takes to traverse it. The longest ascending chains in $C_{Global-CS}^{\#}$ are of length $|\mathcal{B}|$, starting from \emptyset and ending in \mathcal{B} .

For readability we limit our exposition to the analysis of fully-associative caches throughout the article. The extension to set-associative caches is straightforward: Either, a separate set of blocks should be maintained for each cache set, or the classification function $\text{classify}_{Global-CS}^{\#}(\widehat{S}, b)$ should count only those blocks mapping to the same cache set as b .

⁷ We have found two alternative approaches to formalize the cache trace abstractions discussed in this article: (1) by constraints on the memory access trace, and (2) by constraints on the resulting final cache states of the traces. The advantage of approach (1) is that, except for the persistence classification function, it can be proved correct *independently* of the employed cache replacement policy.

In approach (1) the final cache states are constrained implicitly by considering only cache traces that are compatible with the cache replacement policy. Proving correct the persistence classification function then requires invoking a property of LRU, which is condensed in Lemma 9.



■ **Figure 1** Example illustrating *Global-CS*.

5.1.2 Block-CS: Block-wise May-Conflict Set

As soon as more than k memory blocks are accessed by a program, no block can be classified persistent by *Global-CS*. In such cases, many memory blocks may actually still be persistent: Following Lemma 9, a block is persistent if at most $k - 1$ distinct other blocks are accessed between any two accesses to the block itself.

The *block-wise may-conflict set* analysis, abbreviated to *Block-CS*, maintains a separate conflict set for each memory block, rather than a single global conflict set:

$$C_{Block-CS}^{\#} := \mathcal{B} \rightarrow 2^{\mathcal{B}} \quad (24)$$

Then, an abstract trace \hat{S} represents all concrete cache traces in which, following the final access to a block, only blocks from its conflict set may have been accessed:

$$\gamma_{Block-CS}(\hat{S}) := \text{LRUCACHETRACES} \cap \{s = c_0 \langle b_0, h_0 \rangle \dots c_n \mid \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee CS_i(s) \subseteq \hat{S}(b_i)\}, \quad (25)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

Similarly to the global conflict-set case, the initial abstract trace assigns the empty conflict set to each block, which by the concretization function above exactly represents all cache traces of length 0. At joins, the union of the conflict sets is taken in order to overapproximate the conflicting blocks:

$$\widehat{\mathcal{I}_{Block-CS}} := \lambda b \in \mathcal{B}. \emptyset \quad (26)$$

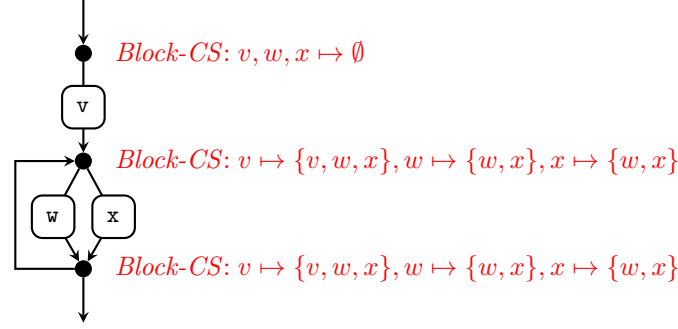
$$\hat{S} \sqsubseteq_{Block-CS} \hat{T} := \forall b \in \mathcal{B} : \hat{S}(b) \subseteq \hat{T}(b) \quad \hat{S} \sqcup_{Block-CS} \hat{T} := \lambda b \in \mathcal{B}. \hat{S}(b) \cup \hat{T}(b) \quad (27)$$

Upon a memory access, the accessed block b is added to the conflict sets of all memory blocks that may have been accessed, i.e. blocks for which $\hat{S}(b') \neq \emptyset$, and crucially b 's conflict set is reset to contain only b . This is where the analysis profits from maintaining separate conflict sets for each block.

$$\text{update}_{Block-CS}^{\#}(\hat{S}, b) := \lambda b'. \begin{cases} \emptyset & : b' \neq b \wedge \hat{S}(b') = \emptyset \\ \{b\} & : b' = b \\ \hat{S}(b') \cup \{b\} & : b' \neq b \wedge \hat{S}(b') \neq \emptyset \end{cases} \quad (28)$$

Finally, a block is locally classified as persistent, if its conflict set, which includes the block itself if it may have been accessed, is guaranteed to contain at most k blocks:

$$\text{classify}_{Block-CS}^{\#}(\hat{S}, b) := |\hat{S}(b)| \leq k \quad (29)$$



■ **Figure 2** Example illustrating *Block-CS*.

Figure 2 shows the result of running *Block-CS* on the same example program as *Global-CS* in the previous section. By tracking each block's conflict set separately – in contrast to *Global-CS* – the analysis is able to determine that w and x are persistent in a cache of associativity 2 as their conflict sets both only contain w and x .

► **Theorem 11** (Soundness of Block-wise May-Conflict Set^{*}). *Block-CS is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{Block-CS}^\# = \mathcal{B} \rightarrow 2^{\mathcal{B}}$ is also finite, and thus it satisfies the ascending chain condition [18, 9], which guarantees termination of the fixpoint iteration to compute the abstract semantics. More precisely, the longest ascending chains are of length $|\mathcal{B}|^2$: Each of the longest ascending chains begins with the bottom element, i.e., the least element of the complete lattice, $\lambda b \in \mathcal{B}. \emptyset$, mapping each block to an empty conflict set, and ends in the top element of the complete lattice, $\lambda b \in \mathcal{B}. \mathcal{B}$, mapping each block to the greatest possible conflict set, consisting of all $|\mathcal{B}|$ memory blocks. In each step of any strictly ascending chain, the conflict set of at least one of the memory blocks needs to grow by at least one block, while none of the conflict sets may shrink. Thus, any ascending chain may contain at most $|\mathcal{B}|^2$ that are greater than the bottom element.

We note that due to the classification condition in (29), it is not necessary to distinguish conflict sets that have more than k elements. Thus, for efficiency, implementations of *Block-CS* should represent all conflict sets with more than k elements by a single unique representative. This has no effect on analysis correctness or precision but reduces the maximum length of ascending chains to $(k + 1) \cdot |\mathcal{B}|$.

► **Theorem 12** (*Block-CS* vs. *Global-CS*). *Block-CS is more precise than Global-CS.*

Proof. In Section 5.1.4 we introduce the *conditional may analysis*, abbreviated to *C-May*. In the same section, in Theorems 16 and 17 we show that *Block-CS* is more precise than *C-May*, and that *C-May* is more precise than *Global-CS*. As the more-precise relation is transitive these two statements imply the theorem. ◀

We note that by Theorem 8 the above two theorems also imply the correctness of *Global-CS*.

5.1.3 C-Must: Conditional Must Analysis

The block-wise may-conflict set approach may lose precision at joins, as the union of the conflict sets needs to be taken. Instead of overapproximating the *set* of conflicting blocks, the *conditional*

must analysis, abbreviated to *C-Must*, bounds the *number* of conflicting blocks. Then it is safe to take the maximum rather than the sum of the bounds at joins.

The *conditional must analysis* thus maintains a bound on the size of the conflict sets of each memory block. A bound of 0 is used to encode that a block is guaranteed not to have been accessed so far, in which case 0 correctly bounds the size of its conflict set. Further, for the purpose of classifying a memory block as persistent, it is not useful to track the size of a block's conflict set beyond k . Therefore, all bounds greater than k are collapsed to ∞ :

$$C_{C-Must}^{\#} := \mathcal{B} \rightarrow \{0, 1, \dots, k, \infty\} \quad (30)$$

Its concretization is very similar to that of the *block-wise may-conflict set analysis*. Instead of overapproximating a block's conflict set, the size of its conflict set is bounded:

$$\gamma_{C-Must}(\widehat{S}) := \text{LRUCACHETRACES} \cap \{s = c_0 \langle b_0, h_0 \rangle \dots c_n \mid \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)\}, \quad (31)$$

where, as before, $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

In the initial abstract cache trace, each block is assigned a bound of 0. By the concretization function this represents all cache traces of length 0, i.e., traces consisting of an arbitrary initial cache state but no memory accesses.

$$\widehat{\mathcal{I}_{C-Must}} := \lambda b \in \mathcal{B}. 0 \quad (32)$$

The advantage of the *conditional must analysis* over the *block-wise may-conflict set analysis* is that the maximum of the bounds can be taken at joins, rather than their sum:

$$\widehat{S} \sqsubseteq_{C-Must} \widehat{T} \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \leq \widehat{T}(b) \quad \widehat{S} \sqcup_{C-Must} \widehat{T} := \lambda b \in \mathcal{B}. \max\{\widehat{S}(b), \widehat{T}(b)\} \quad (33)$$

Upon a memory access, the sizes of the conflict sets of all memory blocks that may have been accessed, i.e. for which $\widehat{S}(b') > 0$ holds, may increase by 1, while the accessed block's conflict set includes only itself, and so its size bound may be reset to 1.

$$\text{update}_{C-Must}^{\#}(\widehat{S}, b) := \lambda b'. \begin{cases} 0 & : b' \neq b \wedge \widehat{S}(b') = 0 \\ 1 & : b' = b \\ \widehat{S}(b') + 1 & : b' \neq b \wedge 0 < \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge k \leq \widehat{S}(b') \end{cases} \quad (34)$$

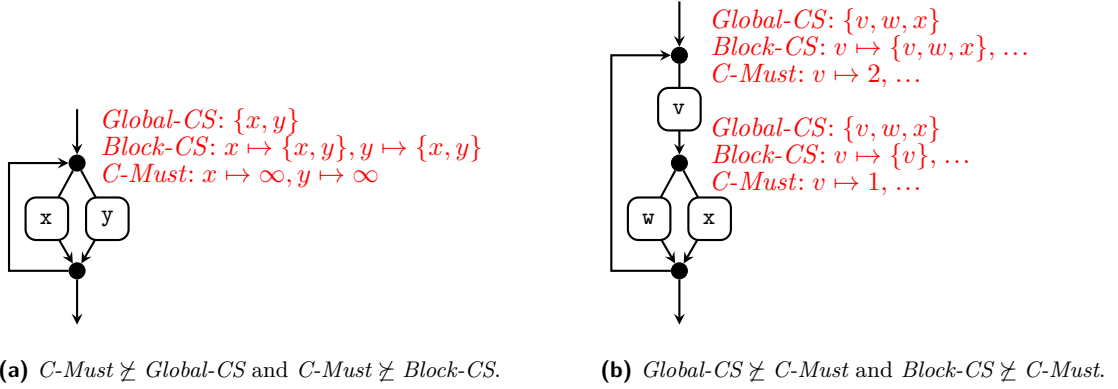
Because the conflict sets are not tracked explicitly, a single memory block may increase the bound of another block multiple times. In such scenarios the *block-wise may-conflict set analysis* may be more precise.

A memory block is locally classified as persistent, if its conflict set is guaranteed to contain less than k blocks:

$$\text{classify}_{C-Must}^{\#}(\widehat{S}, b) := \widehat{S}(b) \leq k \quad (35)$$

► **Theorem 13** (Soundness of Conditional Must*). *C-Must is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{C-Must}^{\#} = \mathcal{B} \rightarrow \{0, 1, \dots, k, \infty\}$ is also finite, and thus termination of the fixpoint iteration to compute the abstract semantics is guaranteed. The longest ascending chains are of length $(k + 1) \cdot |\mathcal{B}|$: Each of the longest ascending chains begins with the bottom element, i.e., the least element of the complete lattice, $\lambda b. 0$. In each step of any strictly ascending chain, the bound for at least one block needs to grow, while none of the bounds may shrink. As the bound of each block may grow at most $k + 1$ times, no strictly ascending chain may be of length greater than $(k + 1) \cdot |\mathcal{B}|$.



■ **Figure 3** Examples illustrating the incomparability of $C\text{-Must}$ with Global-CS and Block-CS .

► **Theorem 14** (Global-CS vs. Block-CS). $C\text{-Must}$ is incomparable to Global-CS and Block-CS .

Proof. Consider the examples in Figures 3a and 3b. Assume a cache with associativity 2. In the first example, both x and y are classified as persistent by Global-CS and Block-CS , in contrast to $C\text{-Must}$. This is because $C\text{-Must}$ may account for the same conflicting block multiple times. On the other hand, in the second example, $C\text{-Must}$ classifies v as persistent, while Block-CS and Global-CS do not. Here, unlike Global-CS and Block-CS , $C\text{-Must}$ is able to capture that in any trace either w or x conflicts with v , but never both. ◀

From the description of the $C\text{-Must}$ analysis it may not be obvious why we choose to call it the *conditional must analysis*. The reason is that it strongly resembles the original *must analysis* by Ferdinand and Wilhelm [12]. The bound on the size of the conflict set of each memory block corresponds to a bound on a memory block's age in the final state of a cache trace *under the condition* that the block has been accessed at least once.

5.1.4 C-May: Conditional May Analysis

Somewhat surprisingly it is also possible to classify memory blocks as persistent with an analysis that determines lower rather than upper bounds on the sizes of memory blocks' conflict sets. The *conditional may analysis*, abbreviated to $C\text{-May}$, maintains a lower bound on the size of the conflict set of each memory block. These lower bounds need to hold only for blocks that have been accessed at least once during program execution. In this sense the bounds are *conditional*. For the purpose of classifying memory blocks as persistent, it is not useful to track the size of a block's conflict set beyond k . Therefore, all lower bounds greater than k are collapsed to $k + 1$. In addition, ∞ is used to indicate that a block has never been accessed: in such cases, ∞ is a correct lower bound on the block's conflict set on the set of traces on which it has been accessed, which is empty.

$$C_{C\text{-May}}^\# := \mathcal{B} \rightarrow \{1, \dots, k, k + 1, \infty\} \quad (36)$$

Its concretization is very similar to that of the *conditional must analysis*. Instead of bounding the size of a block's conflict set from above, it is bounded from below:

$$\gamma_{C\text{-May}}(\hat{S}) := \text{LRUCACHETRACES} \cap \{s = c_0\langle b_0, h_0 \rangle \dots c_n \mid \forall i : 0 \leq i < n : b_i \in CS_{i+1}(s) \vee |CS_i(s)| \geq \hat{S}(b_i)\}, \quad (37)$$

where, as before, $CS_i(c_0\langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

In the initial abstract cache trace, each block is assigned a bound of ∞ . By the concretization function this represents all cache traces of length 0, i.e., traces consisting of an arbitrary initial cache state but no memory accesses.

$$\widehat{\mathcal{I}_{C-May}} := \lambda b \in \mathcal{B}. \infty \quad (38)$$

At joins the minimum of the lower bounds needs to be taken for each memory block:

$$\widehat{S} \sqsubseteq_{C-May} \widehat{T} : \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \geq \widehat{T}(b) \quad \widehat{S} \sqcup_{C-May} \widehat{T} := \lambda b \in \mathcal{B}. \min\{\widehat{S}(b), \widehat{T}(b)\} \quad (39)$$

Upon an access, the accessed block's conflict set shrinks to size 1 (case 1 in (40)). Other block's conflict sets may or may not grow (cases 2 and 3 in (40)). It is safe to increase the lower bound for memory block b' , if the previous lower bound for the *accessed block* b was at least as high (case 3 below) as its own lower bound, which can be understood by the following case distinction:

1. Either b was actually contained in b' 's conflict set before the access. Then b 's conflict set is a strict subset of b' 's conflict set, and so $\widehat{S}(b) + 1 \geq \widehat{S}(b') + 1$ is a lower bound on the size of b' 's conflict set.
2. Or b was not contained in b' 's conflict set before the access. Then b' 's conflict set grows by 1 due to the access to b and thus $\widehat{S}(b') + 1$ is a correct lower bound following the access.

Lower bounds beyond $k + 1$ are not distinguished (case 4 in (40)), and finally, blocks that are guaranteed not to have been accessed yet retain a lower bound of ∞ :

$$update_{C-May}^{\#}(\widehat{S}, b) := \lambda b'. \begin{cases} 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}(b) < \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}(b) \geq \widehat{S}(b') \wedge \widehat{S}(b') \leq k \\ k + 1 & : b' \neq b \wedge \widehat{S}(b) \geq \widehat{S}(b') \wedge \widehat{S}(b') = k + 1 \\ \infty & : b' \neq b \wedge \widehat{S}(b') = \infty \end{cases} \quad (40)$$

Maybe surprisingly⁸, it is possible to classify memory blocks as persistent using the lower bounds derived by the *conditional may analysis*. The intuition behind the classification function is the following: In any concrete cache trace, the conflict sets of the i most-recently-used memory blocks have sizes 1 to i . So only blocks with a lower bound less than or equal to i may be among these i most-recently-used blocks. If there are at most $i \leq k$ memory blocks with a lower bound less than or equal to i , then at most i blocks compete for the first i locations in the cache. So all of these blocks *must* be cached if they have previously been accessed:

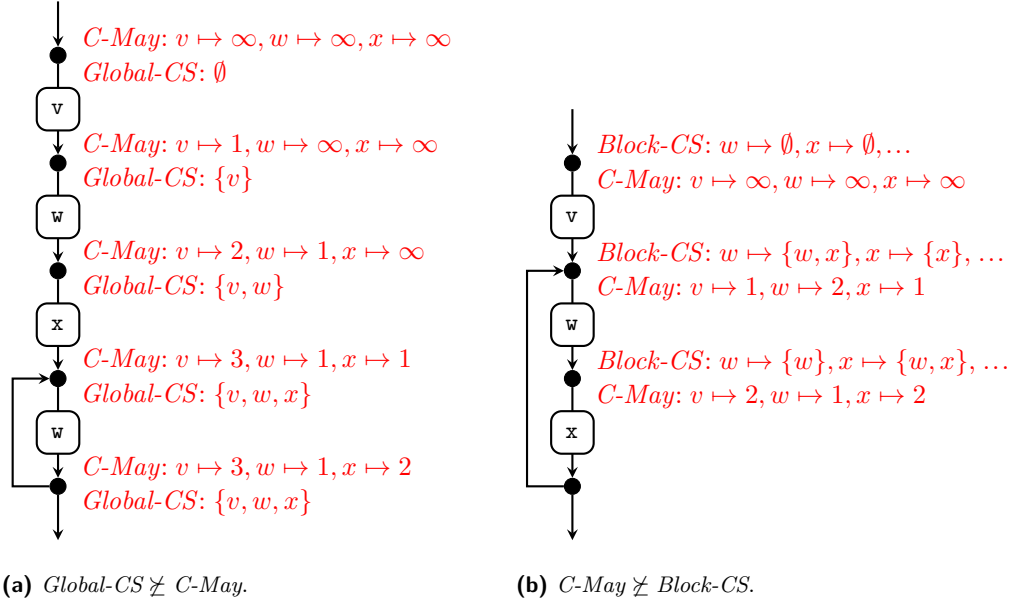
$$classify_{C-May}^{\#}(\widehat{S}, b) := (\widehat{S}(b) = \infty) \vee (\exists i \leq k : |C_i(\widehat{S}, b)| < i), \quad (41)$$

where $C_i(\widehat{S}, b) := \{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{S}(b') \leq i\}$ is the set of memory blocks with a lower bound less than or equal to i other than block b . A detailed proof of correctness is given in the proof of the following theorem:

► **Theorem 15** (Soundness of Conditional May*). *C-May is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{C-May}^{\#} = \mathcal{B} \rightarrow \{1, \dots, k, k + 1, \infty\}$ is also finite, and thus termination of the fixpoint iteration to compute the abstract semantics is guaranteed. The longest ascending chains are of length $(k + 1) \cdot |\mathcal{B}|$, which can be seen following the same train of thought as in the case of *C-Must*.

⁸ Classifying a memory block as persistent following Lemma 9 requires deriving an *upper* bound on the size of a block's conflict set. Thus it may be surprising that the *lower* bounds on blocks' conflict sets determined by *C-May* can be used for this purpose.



■ **Figure 4** Examples illustrating that $Global-CS \not\sqsubseteq C-May$ and $C-May \not\sqsubseteq Block-CS$.

Let us consider two example programs and their analysis using $C-May$ in Figure 4. On the left, in Figure 4a, $C-May$ is able to classify both w and x as persistent in a cache of associativity 2, while none of the blocks are determined persistent by $Global-CS$. For $C-May$, the figure shows the lower bounds on each block's conflict set at each program point. There are at most two blocks with a lower bound of 2 at any program point and thus these blocks are guaranteed to be cached if they have been accessed. On the right, in Figure 4b, $C-May$ is unable to classify w and x as persistent in a cache of associativity 2, while $Block-CS$ is. This is because all three blocks v , w , and x have a lower bound less than or equal to 2 within the loop.

► **Theorem 16** ($C-May$ vs. $Global-CS^*$). $C-May$ is more precise than $Global-CS$.

► **Theorem 17** ($Block-CS$ vs. $C-May^*$). $Block-CS$ is more precise than $C-May$.

By Theorem 8, Theorems 11 and 17 also imply the correctness of $C-May$. Also, due to the transitivity of the more-precise relation, Theorems 16 and 17 together imply Theorem 12, which states that $Block-CS$ is more precise than $Global-CS$.

5.2 Combinations of Basic Abstractions

We have seen four basic cache persistence abstractions: $Block-CS$, $C-May$, $Global-CS$, and $C-Must$. Among these, $Block-CS$ is more precise than $C-May$, which in turn is more precise than $Global-CS$. On the other hand, $C-Must$ is incomparable to $Block-CS$, $C-May$, and $Global-CS$.

To obtain more precise analysis results, it may be beneficial to combine incomparable cache trace abstractions with each other. In Section 5.2.1 we show how to construct the *direct product* of two arbitrary abstractions and show that the direct product of two incomparable abstractions A and B is more precise than A and B individually.

To further increase analysis precision, Section 5.2.2 introduces two ways to exchange information between two abstractions A and B , which may yield cache trace abstractions that are more precise than the direct product of A and B . In the remainder of the section, we then show how to exploit

these two ways of information exchange to build more precise analyses for various combinations of basic analyses.

5.2.1 Direct Product of Cache Trace Abstractions

The direct product of two persistence analyses corresponds to running the two analyses in parallel and classifying a block as persistent if at least one of the two analyses is able to classify the block persistent. Formally, it is defined as follows:

► **Definition 18** (Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is the tuple $A \times B = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{update}_{A \times B}^\#, \text{classify}_{A \times B}^\# \rangle$ with*

$$\begin{aligned} C_{A \times B}^\# &:= C_A^\# \times C_B^\#, \\ \gamma_{A \times B}(\widehat{S}_A, \widehat{S}_B) &:= \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B), \\ \widehat{\mathcal{I}}_{A \times B} &:= \langle \widehat{\mathcal{I}}_A, \widehat{\mathcal{I}}_B \rangle, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqsubseteq_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &\Leftrightarrow \widehat{S}_A \sqsubseteq_A \widehat{T}_A \wedge \widehat{S}_B \sqsubseteq_B \widehat{T}_B, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqcup_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &:= \langle \widehat{S}_A \sqcup_A \widehat{T}_A, \widehat{S}_B \sqcup_B \widehat{T}_B \rangle, \\ \text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \langle \text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b) \rangle, \\ \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b). \end{aligned}$$

► **Theorem 19** (Soundness of Direct Product*). *The direct product $A \times B$ of two sound persistence analyses A and B that satisfy (6), (7), (8), and (10) is a sound persistence analysis.*

In the proof in the appendix, we show that $A \times B$ satisfies the conditions of Theorems 3 and 4, i.e., (6), (7), (8), and (10), and is thus a sound persistence abstraction.

We note that if both A and B satisfy the ascending chain condition, then so does $A \times B$. Thus, persistence analysis with a direct product of two analyses terminates if both constituent analyses are guaranteed to terminate. Moreover, if the lengths of the ascending chains of A and B are bounded by l_A and l_B , then the length of $A \times B$'s longest ascending chains is bounded by $l_A + l_B$.

► **Theorem 20** (Precision of Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is at least as precise as A and B , i.e., $A \times B \succeq A$ and $A \times B \succeq B$.*

Proof. This follows from the fact that the two constituents of $A \times B$ exactly mirror A and B , respectively, and from the fact that

$$\begin{aligned} \text{classify}_A^\#(\widehat{S}_A, b) &\Rightarrow \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) = \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b), \\ \text{classify}_B^\#(\widehat{S}_B, b) &\Rightarrow \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) = \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b). \end{aligned} \quad \blacktriangleleft$$

It is not useful to construct the direct product of two analyses A and B if $A \succ B$, as the result is not going to be more precise than A . If, on the other hand, A and B are incomparable, their direct product will be more precise than both A and B :

► **Corollary 21** (Precision of Direct Product). *The direct product $A \times B$ of two incomparable persistence analyses A and B is more precise than A and B , i.e., $A \times B \succ A$ and $A \times B \succ B$.*

Proof. From Theorem 20 we already know that $A \times B \succeq A$ and $A \times B \succeq B$. Assume for a contradiction that $B \succeq A \times B$. By transitivity of \succeq this would imply $B \succeq A$, which contradicts the assumption that A and B are incomparable. Thus $B \not\succeq A \times B$ and so $A \times B \succ B$. The fact that $A \times B \succ A$ can be shown analogously. \blacktriangleleft

5.2.2 Domain Cooperation

To increase precision, it is sometimes possible for different analyses in a product to exchange information with each other. Here, we distinguish two ways in which such an information exchange can take place between two analyses A and B :

- *State reduction*: the analysis state of A is refined using the analysis state of B .
- *Cooperative update*: The abstract update function for A takes into account not only A 's analysis state but also B 's to compute a more precise successor state.

Below we state correctness conditions for state and update reductions.

► **Definition 22 (State Reduction)**. *Let A and B be persistence analyses. A reduction operator for A in the context of B is a function $red: C_A^\# \times C_B^\# \rightarrow C_A^\#$ that is reductive and that preserves concretizations, i.e., for all $\hat{S}_A \in C_A^\#, \hat{S}_B \in C_B^\#$:*

$$red(\hat{S}_A, \hat{S}_B) \sqsubseteq_A \hat{S}_A, \quad (42)$$

$$\gamma_A(red(\hat{S}_A, \hat{S}_B)) \cap \gamma_B(\hat{S}_B) = \gamma_A(\hat{S}_A) \cap \gamma_B(\hat{S}_B). \quad (43)$$

A reduction operator can be used as follows to obtain a potentially more precise reduced update for the product of A and B :

► **Theorem 23 (State Reduction*)**. *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let red be a reduction operator for A in the context of B . Let the reduced update be defined as follows:*

$$red\text{-}upd(\langle \hat{S}_A, \hat{S}_B \rangle, b) := (red(update_A^\#(\hat{S}_A, b), update_B^\#(\hat{S}_B, b)), update_B^\#(\hat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}_{A \times B}}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, red\text{-}upd, classify_{A \times B}^\# \rangle$ is a sound persistence analysis that is at least as precise as $A \times B$, i.e., $A \times B' \succeq A \times B$.

Sometimes, it is not possible to come up with a state reduction to transfer information between two domains A and B , but it is still possible to profit from the information in B during the update of A . We call such an update *cooperative*:

► **Definition 24 (Cooperative Update)**. *Let A and B be two persistence analyses. A cooperative update for A in the context of B is a function $coop\text{-}upd: (C_A^\# \times C_B^\#) \times \mathcal{B} \rightarrow C_A^\#$, such that:*

$$\begin{aligned} \forall \langle \hat{S}_A, \hat{S}_B \rangle \in C_A^\# \times C_B^\#, b \in \mathcal{B}: \\ \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\hat{S}_A) \cap \gamma_B(\hat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(coop\text{-}upd(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \end{aligned} \quad (44)$$

Given a cooperative update, it is straightforward to define the following reduced update for the product of A and B :

► **Theorem 25 (Cooperative Update)**. *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let $coop\text{-}upd$ be a cooperative update function for A in the context of B . Let the reduced update be defined as follows:*

$$red\text{-}upd(\langle \hat{S}_A, \hat{S}_B \rangle, b) := (coop\text{-}upd(\langle \hat{S}_A, \hat{S}_B \rangle, b), update_B^\#(\hat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}_{A \times B}}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, red\text{-}upd, classify_{A \times B}^\# \rangle$ is a sound persistence analysis.

Proof. We know that $A \times B$ is a sound persistence analysis from Theorem 19. The only condition from Theorem 4 that involves the update function is (8). Thus all conditions but (8) are fulfilled by $A \times B'$ as they are fulfilled by $A \times B$.

For (8), we need to show:

$$\begin{aligned} \forall \langle \hat{S}_A, \hat{S}_B \rangle \in C_{A \times B}^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{A \times B}(\hat{S}_A, \hat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_{A \times B}(\text{red-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \\ = \gamma_A(\text{coop-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \cap \gamma_B(\text{update}_B^\#(\hat{S}_B, b)) \end{aligned} \quad (45)$$

By the soundness of B and by (44) we have

$$\begin{aligned} \forall \hat{S}_A \in C_B^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_B(\hat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_B(\text{update}_B^\#(\hat{S}_B, b)) \end{aligned} \quad (46)$$

$$\begin{aligned} \forall \langle \hat{S}_A, \hat{S}_B \rangle \in C_A^\# \times C_B^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{A \times B}(\hat{S}_A, \hat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{coop-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \end{aligned} \quad (47)$$

Together, (46) and (47) imply (45), and thus (8). \blacktriangleleft

Given the definition of a cooperative update in Definition 24 it is not possible to conclude that the product $A \times B'$ from Theorem 25 is more precise than $A \times B$. However, it is relatively easy to see that this is indeed the case if $\text{coop-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b) \sqsubseteq_A \text{update}_A^\#(\hat{S}_A, b)$ for all \hat{S}_A, \hat{S}_B , and b .

5.2.3 State Reduction between C-Must and Block-CS

In terms of precision, the *block-wise may-conflict set* and the *conditional must* analyses are incomparable, as the former has more precise updates, while the latter has more precise joins. Here we show how to exchange information between the two analyses, by a state reduction, to achieve higher precision than the direct product of the two analyses would.

How can information be exchanged between the two domains? Clearly, the size of the may-conflict set of a block is also a bound on the number of conflicting blocks. Thus, we introduce the following reduction operation:

$$\text{reduce}_{C\text{-Must} \times \text{Block-CS}}(\hat{S}_{C\text{-Must}}, \hat{S}_{\text{Block-CS}}) := \lambda b \in \mathcal{B}. \min \left\{ \hat{S}_{C\text{-Must}}(b), |\hat{S}_{\text{Block-CS}}(b)| \right\} \quad (48)$$

► **Theorem 26** (Soundness of the State Reduction between C-Must and Block-CS). *The function $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is a reduction operator for C-Must in the context of Block-CS.*

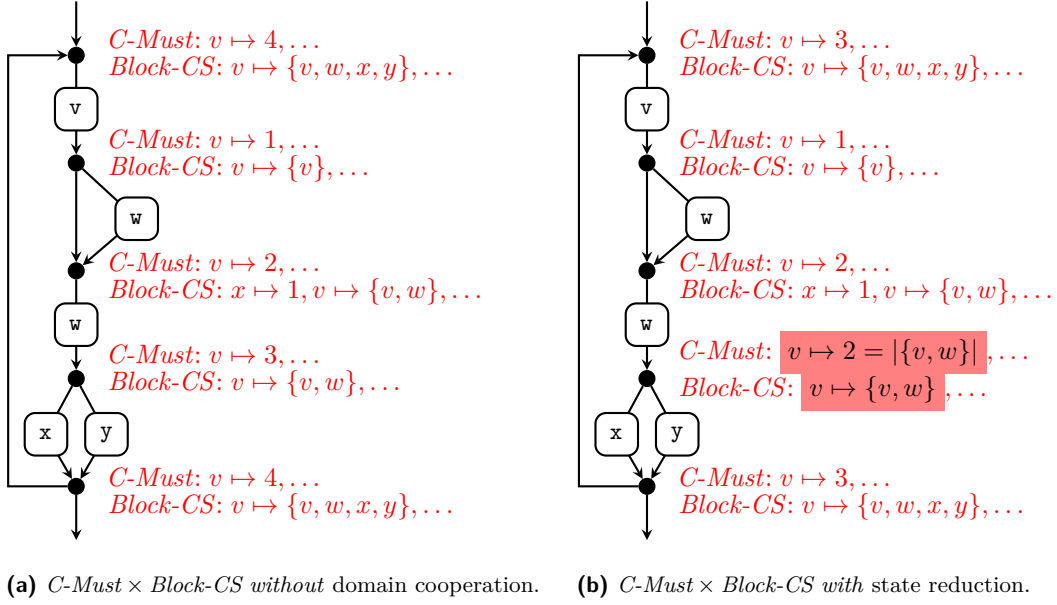
Proof. $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is reductive, as $\min \left\{ \hat{S}_{C\text{-Must}}(b), |\hat{S}_{\text{Block-CS}}(b)| \right\} \leq \hat{S}_{C\text{-Must}}(b)$.

It remains to show that for all $\hat{S}_1 \in C_{C\text{-Must}}^\#, \hat{S}_2 \in C_{\text{Block-CS}}^\#$:

$$\gamma_{C\text{-Must} \times \text{Block-CS}}(\hat{S}_1, \hat{S}_2) = \gamma_{C\text{-Must} \times \text{Block-CS}}(\text{reduce}_{C\text{-Must} \times \text{Block-CS}}(\hat{S}_1, \hat{S}_2), \hat{S}_2).$$

As $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is reductive and $\gamma_{C\text{-Must} \times \text{Block-CS}}$ is monotone, we have

$$\gamma_{C\text{-Must} \times \text{Block-CS}}(\hat{S}_1, \hat{S}_2) \supseteq \gamma_{C\text{-Must} \times \text{Block-CS}}(\text{reduce}_{C\text{-Must} \times \text{Block-CS}}(\hat{S}_1, \hat{S}_2), \hat{S}_2).$$



■ **Figure 5** Example illustrating *C-Must* × *Block-CS*.

To show that $\gamma_{C\text{-}Must \times Block\text{-}CS}(\hat{S}_1, \hat{S}_2) \subseteq \gamma_{C\text{-}Must \times Block\text{-}CS}(\text{reduce}_{C\text{-}Must \times Block\text{-}CS}(\hat{S}_1, \hat{S}_2), \hat{S}_2)$, assume for a contradiction that there is a trace $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ in $\gamma_{C\text{-}Must \times Block\text{-}CS}(\hat{S}_1, \hat{S}_2)$ that is not in $\gamma_{C\text{-}Must \times Block\text{-}CS}(\text{reduce}_{C\text{-}Must \times Block\text{-}CS}(\hat{S}_1, \hat{S}_2), \hat{S}_2)$. Then, there must be an $i, 0 \leq i < n$, such that $|CS_i| \leq \hat{S}_1(b_i)$ and $CS_i \subseteq \hat{S}_2(b_i)$, but $|CS_i| \not\leq \min\{\hat{S}_1(b_i), |\hat{S}_2(b_i)|\}$. However, observe that $CS_i \subseteq \hat{S}_2(b_i)$ implies that $|CS_i| \leq |\hat{S}_2(b_i)|$, and so $|CS_i| \leq \min\{\hat{S}_1(b_i), |\hat{S}_2(b_i)|\}$, which yields a contradiction. ◀

We have seen previously that *C-Must* and *Block-CS* are incomparable in terms of precision. See Figure 5 for an example where the state reduction described above yields a more precise analysis result than is possible with any of the two analyses in isolation. In the example, v is persistent in a cache of associativity 3, which neither *C-Must* nor *Block-CS* are able to prove on their own. The example consists of a loop whose body contains two phases, each of which can only be handled precisely by one of the two domains. In the first phase of the loop body, containing the two accesses to w , *Block-CS* is more precise, as it does not double count these two accesses in v 's conflict set. In the second phase of the loop body, *C-Must* is more precise as it accounts for a single conflict due to the potential accesses to x and y , whereas *Block-CS* accounts for two conflicts. State reduction enables *C-Must* to profit from the more precise *Block-CS* analysis in the phase of the loop body, reducing the bound for v to 2, as highlighted in the figure on the right. Due to this reduction, *C-Must* is then able to show that v is indeed persistent in a cache of associativity 3.

5.2.4 State Reduction between C-Must and C-May

Similarly to the information exchange between *C-Must* and *Block-CS*, information can also be exchanged between *C-Must* and *C-May*.

The idea of the reduction is the following: *C-May* and *C-Must* provide lower and upper bounds on the ages of memory blocks that have been accessed. A memory block b 's conflict set may only include another block c , if c 's lower bound is less than b 's upper bound. Thus b 's conflict set must be a subset of $\{b\} \cup \{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}$ and so its size is bounded by $|\{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}| + 1$.

Based on this insight, we introduce the following reduction operation:

$$\begin{aligned} \text{reduce}_{C-Must \times C-May}(\widehat{S}_{C-Must}, \widehat{S}_{C-May}) := \\ \lambda b \in \mathcal{B}. \min \left\{ \widehat{S}_{C-Must}(b), |\{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}| + 1 \right\} \end{aligned} \quad (49)$$

► **Theorem 27** (Soundness of the State Reduction between *C-Must* and *C-May*^{*}). *The operator $\text{reduce}_{C-Must \times C-May}$ is a reduction operator for *C-Must* in the context of *C-May*.*

On the example program given in Figure 5 the state reduction between *C-Must* and *C-May* yields the same analysis result as the state reduction between *C-Must* and *Block-CS* given in the previous section.

5.2.5 Must Analysis

Recall the update of the *C-Must* analysis in (34). The bound on the size of a block's conflict set is increased by 1 upon *any* access to a different block (case 3 in (34)). At first sight it may seem that the update could be improved. It is tempting to take into account the size of the conflict set of the accessed block, as we did in case of *C-May*. Unfortunately, any such attempt would be incorrect: This is because the size bounds determined by *C-Must* are *conditional*, i.e., they only hold given that a block has been accessed at all. Given this definition of *C-Must*, it is always possible that an access is the very first access to the given block, which would contribute to the conflict sets of all other blocks, which have been accessed before, necessitating the update as it is.

In order to improve the update of *C-Must*, *unconditional* bounds on the sizes of blocks' conflict sets are required. Such unconditional bounds can be determined using the original LRU must analysis by Ferdinand and Wilhelm [12]. In this section, we recapitulate this must analysis, which we simply call *Must*. In order to use it in a cooperative update of *C-Must* in the context of *Must* in Section 5.2.6, we formalize the *Must* analysis as a persistence analysis. In particular, we provide a concretization function that expresses the set of cache traces represented by a *Must* analysis state. As a stand-alone persistence analysis, *Must* is not useful at all: no memory block can be classified as persistent by a stand-alone *Must* analysis. Its utility in persistence analysis stems from the fact that it may be used to improve the precision of *C-Must* via a cooperative update, which is provided in the following section.

Must analysis maintains an upper bound on the ages of memory blocks, where age bounds greater than k are collapsed to ∞ :

$$C_{Must}^{\#} := \mathcal{B} \rightarrow \{1, \dots, k, \infty\} \quad (50)$$

The original formalization of *Must* in [12] is not based on a *trace* collecting semantics. There, an abstract state corresponds to a set of concrete cache states. To fit into our persistence analysis framework, we here give an alternative formalization that captures the set of cache traces represented by an abstract cache trace⁹. The resulting concretization is quite similar to the one

⁹ We use the term *abstract trace* rather than *abstract state* as we interpret it to represent a set of cache traces.

for $C\text{-}Must$. The difference is that $C_{Must}^\#(b)$ must be ∞ for blocks that have not been accessed:

$$\begin{aligned} \gamma_{Must}(\widehat{S}) &:= \text{LRUCACHETRACES} \cap \\ &\{s = c_0\langle b_0, h_0 \rangle \dots c_n \mid (\forall b \in \mathcal{B} : (\forall i, 0 \leq i < n : b_i \neq b) \Rightarrow \widehat{S}(b) = \infty) \\ &\quad \wedge \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)\}, \end{aligned} \quad (51)$$

where, as before, $CS_i(c_0\langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

The concretization function given in the original formalization by Ferdinand and Wilhelm [12] is an abstraction of the one given above. It captures the final cache states of the cache traces determined by $\gamma_{Must}(\widehat{S})$, which is sufficient to classify memory accesses as guaranteed hits.

In the initial abstract cache trace, each block is assigned a bound of ∞ . By the concretization function this represents all possible cache traces, in particular those of length 0, i.e., traces consisting of an arbitrary initial cache state:

$$\widehat{\mathcal{I}}_{Must} := \lambda b \in \mathcal{B}. \infty \quad (52)$$

As in $C\text{-}Must$ the maximum of the bounds is taken at joins:

$$\widehat{S} \sqsubseteq_{Must} \widehat{T} : \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \leq \widehat{T}(b) \quad \widehat{S} \sqcup_{Must} \widehat{T} := \lambda b \in \mathcal{B}. \max\{\widehat{S}(b), \widehat{T}(b)\} \quad (53)$$

Upon a memory access, the accessed block's bound is reduced to 1, as its conflict set will only contain the block itself (case 1, below). Other blocks' bounds are increased only if the accessed block's bound is greater than their bound (cases 2 and 3). This is sound because the bounds are unconditional in the must analysis.

$$update_{Must}^\#(\widehat{S}, b) := \lambda b'. \begin{cases} 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}(b) \leq \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}(b) > \widehat{S}(b') \wedge \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge \widehat{S}(b) > \widehat{S}(b') \wedge \widehat{S}(b') = k \end{cases} \quad (54)$$

For completeness, we provide the following classification function. A memory block is locally classified as persistent, if its bound is less than or equal to k , which implies that the block *must* be cached:

$$classify_{Must}^\#(\widehat{S}, b) := \widehat{S}(b) \leq k \quad (55)$$

As the bounds are initialized to ∞ , prior to the first access to a block, no block can be classified as persistent. This is why *Must* is not useful as a stand-alone persistence analysis.

► **Theorem 28** (Soundness of Must Analysis*). *Must is a sound persistence analysis.*

5.2.6 Cooperative Update for C-Must in the Context of Must

The unconditional bounds computed by *Must* can be used to improve the update of $C\text{-}Must$. A cooperative update for $C\text{-}Must$ in the context of *Must* is given below:

$$coop\text{-}upd_{C\text{-}Must \times Must}(\widehat{S}, \widehat{S}_{Must}, b) := \lambda b'. \begin{cases} 0 & : b' \neq b \wedge \widehat{S}(b') = 0 \\ 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}_{Must}(b) \leq \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}_{Must}(b) > \widehat{S}(b') \wedge 0 < \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge \widehat{S}_{Must}(b) > \widehat{S}(b') \wedge k \leq \widehat{S}(b') \end{cases} \quad (56)$$

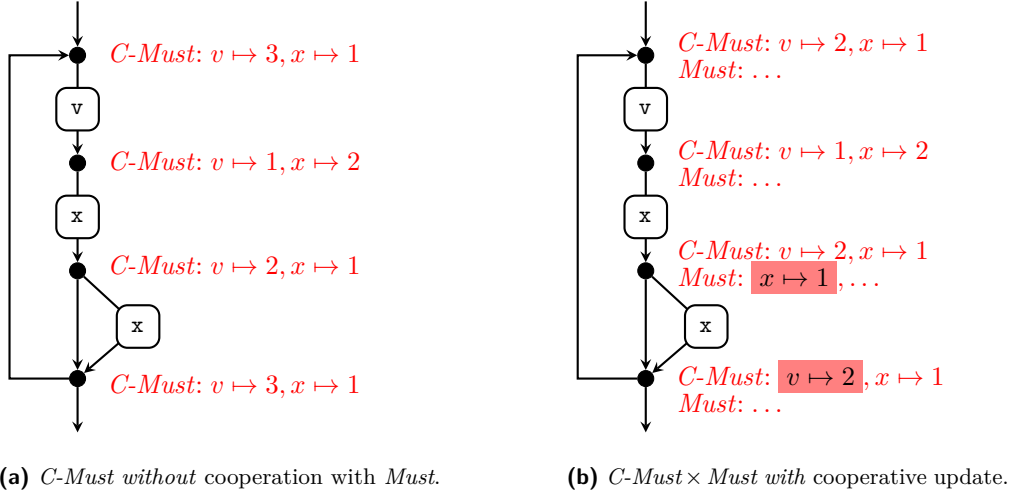


Figure 6 Example illustrating $C-Must \times Must$.

The update differs from the stand-alone update for $C-Must$ in the third case, where the bound for b' is not increased even though another block is accessed. The correctness of all other cases follows from the correctness of the original update for $C-Must$. Let's consider the third case more carefully. It occurs under the condition that $b' \neq b \wedge \hat{S}_{Must}(b) \leq \hat{S}(b')$. If $\hat{S}(b') = \infty$ then the update is trivially correct. So assume $\hat{S}(b') \leq k$ and thus also $\hat{S}_{Must}(b) \leq k$. In this case, the correctness of the update can be understood by the following case distinction:

1. Either b was actually contained in b' 's conflict set before the access. Then b' 's conflict set does not grow due to the access to b and keeping the previous bound on its size is correct.
2. Or b was not contained in b' 's conflict set before the access. Crucially, b must have been accessed before, as $\hat{S}_{Must}(b) \leq k$, and so b 's conflict set must contain b' 's conflict set. Further, b 's conflict set additionally contains b itself. As a consequence, b 's conflict set before the access contains b' 's conflict set after the access, and thus its bound, $\hat{S}_{Must}(b) \leq \hat{S}(b')$ is a correct bound for b' 's conflict set after the access.

A more formal and detailed correctness argument is given in the proof of the following theorem:

► **Theorem 29** (Soundness of Cooperative Update^{*}). *The function $coop-upd_{C-Must \times Must}$ is a cooperative update for $C-Must$ in the context of $Must$.*

Let's consider a small example illustrating the benefit of the cooperative update for $C-Must$ in the context of $Must$. Figures 6a and 6b show the analysis results of $C-Must$ with and without cooperation with $Must$ on a loop containing a conditional. Without cooperation, $C-Must$ is unable to prove that v is persistent in a cache of associativity 2. This is because, the two accesses to x both cause the bound on the size of v 's conflict set to increase by 1, even though only the first access to x may actually increase its size. In contrast, $C-Must \times Must$ with a cooperative update is able to prove that v is persistent in a cache of associativity 2, as illustrated in Figure 6b. Here, we only show the *relevant* information that $Must$ provides for the cooperative update: Right before the potential second access to x in the loop, $Must$ determines an *unconditional* bound on the size of x 's conflict set of 1. This information is then exploited in the cooperative update to determine that this second access may not increase the size of v 's conflict set, and so 2 is a correct bound on v 's conflict set at the end of the loop body.

5.3 Summary: The Landscape of Persistence Abstractions

In this section, we summarize the results obtained in Sections 5.1 and 5.2. In Section 5.1 we have seen abstractions following these two general approaches to persistence analysis:

1. Overapproximating the *set* of conflicting blocks
2. Overapproximating the *number* of conflicting blocks

Global-CS, *C-May*, and *Block-CS* all follow the first of these two approaches. Further, these three abstractions are totally ordered in terms of precision, with *Block-CS* strictly dominating *C-May*, and *C-May* strictly dominating *Global-CS*.

The only basic abstraction following the second approach is *C-Must*, which is incomparable to *Global-CS*, *C-May*, and *Block-CS*, i.e., there are cases where *C-Must* is more precise than *Block-CS*, but there are also cases where even *Global-CS* is more precise than *C-Must*.

In Section 5.2 we have then seen how to combine two incomparable basic abstractions into their so-called *direct product*, which is more precise than its constituents. To further increase analysis precision, we have also introduced two general mechanisms to exchange information between two abstractions:

- *state reduction* – where the analysis state of one abstraction is refined based on the analysis state of another abstraction, and
- *update reduction* – where the update of one abstraction takes into account information provided by another abstraction.

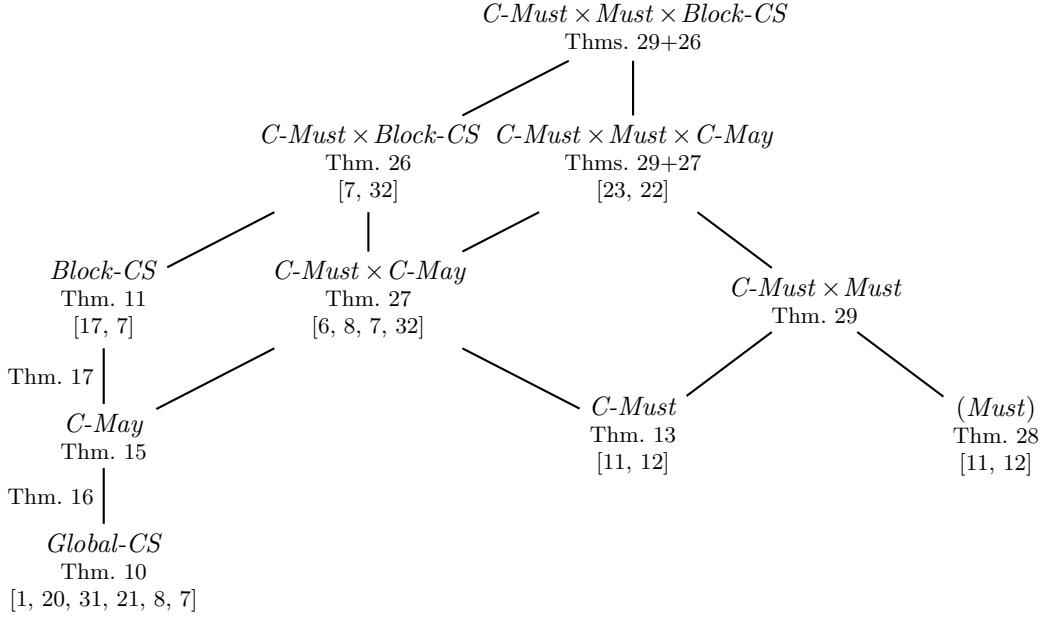
Then, we have seen three concrete instantiations of these mechanisms: state reductions between *C-Must* and *Block-CS* and between *C-Must* and *C-May*, as well as an update reduction between *C-Must* and a regular must cache analysis, which we simply call *Must* here. To facilitate the update reduction with *C-Must*, we have formalized *Must* as a persistence analysis, i.e., as an abstraction of sets of cache traces rather than sets of cache states.

We note that a state reduction between *C-Must* and *Global-CS* could be defined easily, similarly to the state reduction between *C-Must* and *Block-CS*. We do not provide this reduction here, because we believe that it would have little practical value: We have shown *C-May* to strictly dominate *Global-CS*, while it is hardly more expensive in terms of analysis time and memory consumption than *Global-CS*.

Figure 7 illustrates the landscape of persistence abstractions formalized in this article in the form of a Hasse diagram. Two comparable abstractions are connected by an edge, with the more precise abstraction drawn higher up in the diagram. Transitive relations are omitted for readability. Each abstraction is annotated with the corresponding theorem that shows its soundness. Similarly, relations between abstractions that are not the consequence of a product constructions are annotated with the corresponding dominance theorem. We also annotate abstractions with papers from the literature, which are based on the given abstraction. We discuss the related work in more detail in the following section.

6 Related Work and How It Maps Into the Landscape of Persistence Abstractions

In 1994, Mueller et al. [20, 1] introduced the “first miss” persistence notion and a corresponding persistence analysis for direct-mapped instruction caches. Later they extended their analysis to set-associative data [31] and instruction caches [21] with LRU replacement. The basic idea behind their analysis for set-associative caches is to collect all conflicting blocks in a given cache set within a loop. If all conflicting blocks fit into the cache together, then these blocks are classified as “first miss”. This corresponds to *Global-CS* applied separately to each loop in the program.



■ **Figure 7** Hasse diagram illustrating the relative precision of different persistence abstractions. The *Must* domain is in parentheses because it is not suitable to prove persistence of memory blocks on its own, but it may be useful in conjunction with other domains.

Ferdinand and Wilhelm [11, 12] introduced the “no eviction” persistence notion and a persistence analysis for set-associative caches. They characterized their analysis [12] as computing “the *maximal* position (relative age) for all memory blocks that *may* be in the cache.” Intuitively, their analysis thus corresponds to the *C-Must* analysis defined in this article. However, while their analysis employs the same join function as in the *C-Must* analysis, its update function differs; it is identical to the update function of the *Must* analysis: upon an access to memory block b only the ages of younger blocks are incremented. Unfortunately, this is unsound. To our knowledge, Hugues Cassé was the first to point this out. Given the concretization function of *C-Must* defined in this article, it is apparent why the update function is incorrect: *C-Must* bounds the age (the size of its conflict set) of a block only in case the block has previously been accessed. If the accessed block has not been accessed previously, it increases the ages of all other previously accessed blocks (the sizes of their conflict sets, respectively). Thus, without any additional information, upon an access to block b , a sound update function has to increase the bounds of all blocks, other than b , that have potentially been accessed before. The original *Must* analysis may provide such additional information, as it provides unconditional bounds on the maximum ages of blocks. The cooperative update for *C-Must* in the context of *Must* defined in Section 5.2.6 shows how to exploit this information for a more precise update of *C-Must*.

Aiming to solve the soundness issue of Ferdinand’s analysis, Cullmann [6] proposed an analysis combining Ferdinand’s persistence analysis with a slightly modified version of the *may* analysis from [12]. This combination corresponds to the product of *C-Must* and *C-May*, however with a less precise reduction than the one given in (49).

In a different approach to fix the soundness issues of the original persistence analysis, Huynh et al. [17] proposed a scope-aware persistence analysis for set-associative data caches. Their analysis tracks a *younger set* for each memory block, which corresponds to the *Block-CS* analysis in this article. To increase precision in the analysis of data caches, *temporal scopes* are used to distinguish different loop iterations in which array accesses in a loop touch different memory blocks.

■ **Listing 2** Input- and loop-iteration-dependent data accesses.

```
for (int i=0; i<N; i++) {
    k = read_sensor();
    sum[k] = sum[k] + arr[i];
}
```

Later, Cullmann [8] proposed “set-wise conflict counting”, which corresponds to *Global-CS* in this article and is similar to Mueller et al.’s approach. In his dissertation [7], Cullmann discusses two further analyses:

1. *Element-wise conflict counting*, which corresponds to the younger-set analysis by Huynh et al. [17] and *Block-CS* in this article.
2. *Age-tracking conflict counting*, which corresponds to the direct product of *C-Must* and *Block-CS* in this article, however, without the state reduction given in (48).

Nagar and Srikant [23, 22] show how to improve the precision of must, may, and persistence analysis by exchanging information between the analyses via what we call reductions in this article. Along the way they also identify and correct the soundness issue of Ferdinand’s persistence analysis. In case of persistence analysis, their approach corresponds to $C\text{-Must} \times \text{Must} \times C\text{-May}$ with the update and state reductions given in Theorems 29 and 27.

Similarly to Nagar and Srikant, Zhang and Koutsoukos [32] show how to combine *C-Must* and *C-May* using an update reduction. Their update improves upon Cullmann’s update in the combination of *C-Must* and *C-May*, which only uses the information from *C-May* to exclude the eviction of memory blocks from *C-Must*. Zhang and Koutsoukos also note that *Block-CS* is incomparable to $C\text{-Must} \times C\text{-May}$. They propose to combine $C\text{-Must} \times C\text{-May}$ and *Block-CS* in a single analysis to achieve higher precision than any of its constituents. As *Block-CS* dominates *C-May*, the resulting analysis is equivalent to $C\text{-Must} \times \text{Block-CS}$ in our framework (with the appropriate state reduction), and one might wonder why it could be useful to run the two analyses together with *C-May*. However, Zhang and Koutsoukos show how to derive “younger sets” from $C\text{-Must} \times C\text{-May}$. Whenever the derived younger set is equal to the one maintained by *Block-CS* it is not necessary to explicitly represent the younger set in *Block-CS*. In this way, the memory consumption of the analysis can be significantly reduced.

Ballabriga and Cassé [2] note that different blocks can be persistent within different scopes. For example, while an inner loop may entirely fit into the cache, its enclosing loop might not. In such a case, a sound persistence analysis would not be able to declare any of the loop’s memory blocks as persistent. Still, during any execution of the inner loop, each of its memory blocks may miss the cache at most once. Ballabriga and Cassé thus propose “multi-level” persistence analysis, which determines for each loop nesting level, whether blocks are persistent within the execution of the loop at that nesting level. This idea was later also applied to “temporal scopes” by Huynh et al. [17] as discussed earlier.

7 Extension to Data Caches

In the preceding sections we have focused on persistence analysis for instruction caches. Persistence analysis for data caches faces the additional challenge that an individual load or store instruction may result in different data memory accesses depending on the program’s inputs or the loop iteration the instruction is executed in. Consider the example in Listing 2. Within the loop, the access to the array `arr` depends on the loop iteration, and the accesses to the array `sum` depend on external sensor inputs.

It is possible to employ a control flow abstraction similar to the one described in Section 3.1 to such programs. However, due to input- and loop-iteration-dependent data accesses, some transitions in the control flow graph will have to be annotated with a set of possible memory blocks rather than an single one. The abstract trace update function then needs to be lifted to sets, which can be done in a generic manner as follows:

$$\text{update}_A^\#(\widehat{S}, B) := \bigsqcup_{b \in B} \text{update}_A^\#(\widehat{S}, b), \quad (57)$$

where B is a set of memory blocks.

In this way, all the persistence analyses discussed in this article can also be applied to the analysis of data caches. However, this generic approach comes with two drawbacks:

1. *Reduced efficiency*: Implementing (57) literally, the update and join functions need to be applied $|B|$ and $|B| - 1$ times, respectively. So if the set of potentially-accessed blocks B is large the analysis may become quite costly. However, in most cases, it is fairly straightforward to derive an expression for $\text{update}_A^\#(\widehat{S}, B)$ that does not involve applying the original update and join functions that often. For example, $\text{update}_{Global-CS}^\#(\widehat{S}, B) = \bigsqcup_{b \in B} \text{update}_{Global-CS}^\#(\widehat{S}, b)$ can be simplified to $\text{update}_{Global-CS}^\#(\widehat{S}, B) = \widehat{S} \cup B$. Nagar and Srikant [23, 22] describe such simplifications for their persistence analysis.
2. *Limited precision*: Due to uncertainty about the accessed memory blocks the analysis may be imprecise. In case of loop-iteration-dependent data accesses, more precise persistence classifications could be derived by performing the analysis on a more fine-grained abstraction of the program than its control flow abstraction. For instance, to increase analysis precision, Huynh et al. [17] introduce *temporal scopes* to distinguish different loop iterations in which array accesses in a loop touch different memory blocks.

8 Conclusions and Future Work

Our main goal has been to put persistence analysis on a more solid semantic foundation. We have argued that persistence is a property of cache traces rather than cache states. Accordingly, we introduced a trace-based semantics to formally capture varying persistence notions and to enable rigorous correctness proofs of persistence analyses.

Section 5 demonstrates that persistence analyses can be defined and proved correct as abstractions of a trace collecting semantics; we believe rather elegantly. Such formalizations also contribute to a better understanding of how and why an analysis works, simply by requiring its designer to precisely capture the meaning of the abstraction that the analysis is based upon. To our own surprise, it is possible to explain the essence of all prior persistence abstractions as combinations of just a few rather basic abstractions.

We note that our focus has been on the underlying abstractions and not on their efficient implementation. Different implementations of the same abstraction will deliver the same persistence classifications, but may exhibit different performance characteristics, in particular in terms of space consumption. This is, for example, demonstrated by Zhang and Koutsoukous [32], who show how to implement *Block-CS* more efficiently than a straightforward implementation that directly matches its logical definition.

In this article, we have only considered private single-level caches with LRU replacement. Future work should consider replacement policies other than LRU, which have received some attention in classifying cache analysis [26, 27, 25, 13, 14] and in the broader scope of quantitative cache analysis [16, 15], but which have so far received very little attention in persistence analysis. It may also be interesting to study persistence analysis for shared caches in multi cores. Such

shared caches are usually second- or third-level caches and thus any step in this direction would also require the analysis of multi-level caches. The lattice of abstractions in Figure 7 may be a good starting point for a rigorous experimental evaluation of the various persistence analyses that have been proposed to date. All abstractions studied in this article are sound but incomplete. It is conceivable to design a sound and complete persistence analysis along the lines of the recent work of Touzeau et al. [30].

Acknowledgments. I would like to sincerely thank the anonymous reviewers for their help in improving this paper.

References

- Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*, pages 172–181. IEEE Computer Society, 1994. doi:10.1109/REAL.1994.342718.
- Clément Ballabriga and Hugues Cassé. Improving the first-miss computation in set-associative instruction caches. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 341–350. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.34.
- Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.
- Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.
- Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, volume 156 of *IFIP*, pages 359–366. Kluwer/Springer, 2004. doi:10.1007/978-1-4020-8157-6_27.
- Christoph Cullmann. Cache persistence analysis: a novel approach theory and practice. In Jan Vitek and Bjorn De Sutter, editors, *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 121–130. ACM, 2011. doi:10.1145/1967677.1967695.
- Christoph Cullmann. *Cache persistence analysis for embedded real-time systems*. PhD thesis, Saarland University, Saarbrücken, Germany, 2013. URL: <http://d-nb.info/1052779867>.
- Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embedded Comput. Syst.*, 12(1s):40:1–40:25, 2013. doi:10.1145/2435227.2435236.
- B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.
- Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, 2015. doi:10.1145/2756550.
- Christian Ferdinand. *Cache behavior prediction for real-time systems*. PhD thesis, Saarland University, Saarbrücken, Germany, 1997. URL: <http://d-nb.info/953983706>.
- Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999. doi:10.1023/A:1008186323068.
- Daniel Grund and Jan Reineke. Precise and efficient fifo-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 155–164. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.8.
- Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 23–35. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASICS.WCET.2010.23.
- Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU cache: Challenging LRU for predictability. *ACM Trans. Embedded Comput. Syst.*, 13(4s):123:1–123:26, 2014. doi:10.1145/2584655.
- Nan Guan, Xiping Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.073.
- Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*,

- RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011, pages 203–212. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.27.
- 18 Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973. doi:10.1145/512927.512945.
 - 19 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *LITES*, 3(1):05:1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.
 - 20 Frank Mueller. *Static cache simulation and its applications*. PhD thesis, Florida State University, Tallahassee, United States, 1994. URL: http://www.cs.fsu.edu/~whalley/papers/mueller_diss94.pdf.
 - 21 Frank Müller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, 2000. doi:10.1023/A:1008145215849.
 - 22 Kartik Nagar. Cache analysis for multi-level data caches. Master’s thesis, Indian Institute of Science, Bangalore, India, 2012.
 - 23 Kartik Nagar and Y. N. Srikant. Interdependent cache analyses for better precision and safety. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEM-CODE 2012, Arlington, VA, USA, July 16-17, 2012*, pages 99–108. IEEE, 2012. doi:10.1109/MEMCOD.2012.6292306.
 - 24 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
 - 25 Jens Palsberg and Zhendong Su, editors. *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-03237-0.
 - 26 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008. URL: <http://rw4.cs.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalysis.pdf>.
 - 27 Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In Krisztián Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’08), Tucson, AZ, USA, June 12-13, 2008*, pages 51–60. ACM, 2008. doi:10.1145/1375657.1375665.
 - 28 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007. doi:10.1145/1275497.1275501.
 - 29 Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design - Analysis and Transformation*. Springer, 2012. doi:10.1007/978-3-642-17548-0.
 - 30 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. doi:10.1007/978-3-319-63390-9_2.
 - 31 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium, RTAS ’97, Montreal, Canada, June 9-11, 1997*, pages 192–202. IEEE Computer Society, 1997. doi:10.1109/RTTAS.1997.601358.
 - 32 Zhenkai Zhang and Xenofon D. Koutsoukos. Improving the precision of abstract interpretation based cache persistence analysis. In Sam H. Noh, Sebastian Fischmeister, and Jason Xue, editors, *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2015, CD-ROM, Portland, OR, USA, June 18 - 19, 2015*, pages 10:1–10:10. ACM, 2015. doi:10.1145/2670529.2754967.

A Proofs

► **Definition 1** (Persistence in a Program). *Memory block b is persistent in program P , if*

$$\forall \tau \in \text{Col}(P) : \text{AtMostOneMiss}(\tau, b).$$

► **Definition 2** (Cache Trace Abstraction). *A cache trace abstraction is a tuple*

$$A = \langle C_A^\#, \gamma_A, \widehat{\mathcal{I}}_A, \sqsubseteq_A, \sqcup_A, \text{update}_A^\#, \text{classify}_A^\# \rangle,$$

consisting of the following components:

1. $C_A^\#$, a set of abstract traces,
2. $\gamma_A : C_A^\# \rightarrow 2^{\text{CacheTraces}}$, a concretization function, which specifies the set of concrete cache traces represented by each abstract trace,

3. $\widehat{\mathcal{I}}_A \in C_A^\#$, an abstract initial trace that represents all possible initial cache states,
4. \sqsubseteq_A , a partial order on $C_A^\#$, such that $\langle C_A^\#, \sqsubseteq_A \rangle$ is a complete lattice [9],
5. \sqcup_A , a join operator on abstract traces¹⁰,
6. $\text{update}_A^\# : C_A^\# \times \mathcal{B} \rightarrow C_A^\#$, an abstract update function,
7. $\text{classify}_A^\# : C_A^\# \times \mathcal{B} \rightarrow \mathbb{B}$, a persistence classification function.

In the proof of the following theorem, we will make use of Knaster-Tarski's fixpoint theorem. Many variants of Knaster-Tarski's fixpoint theorem can be found in the literature. Below, we reproduce one such variant and its proof from [9], adapted to the terminology used in this article:

► **Theorem 30 (Knaster-Tarski Fixpoint Theorem).** *Let (L, \leq) be a complete lattice and $F : L \rightarrow L$ a monotone function. Let $\bigwedge A$ denote the greatest lower bound of $A \subseteq L$. Then,*

$$\alpha := \bigwedge \{x \in L \mid F(x) \leq x\}$$

is a fixpoint of F . Further, α is the least fixpoint of F .

Proof. Let $H = \{x \in L \mid F(x) \leq x\}$. For all $x \in H$, we have $\alpha \leq x$, so $F(\alpha) \leq F(x) \leq x$. Thus $F(\alpha)$ is a lower bound of H , and so $F(\alpha) \leq \alpha$, as α is the greatest lower bound of H .

Since F is monotone, $F(F(\alpha)) \leq F(\alpha)$, and so $F(\alpha) \in H$, and thus $\alpha \leq F(\alpha)$. Thus we have established that α is a fixpoint of F .

If β is any fixpoint of F , then $\beta \in H$, and so $\alpha \leq \beta$. Thus α is the least fixpoint of F . ◀

► **Theorem 3 (Soundness of Persistence Analysis).** *If the cache trace abstraction A satisfies the following conditions:*

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A), \tag{6}$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\# : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \gamma_A(\widehat{S}) \subseteq \gamma_A(\widehat{T}), \tag{7}$$

$$\begin{aligned} \forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{update}_A^\#(\widehat{S}, b)). \end{aligned} \tag{8}$$

Then, its abstract semantics soundly approximates its concrete counterpart:

$$\text{StickyCol}(P_{\text{ins}}) \leq \gamma_A(\widehat{\text{StickyCol}}_A(P_{\text{ins}})), \tag{9}$$

where γ_A is lifted to functions as follows: $\gamma_A(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_A(\widehat{S}(l))$.

Proof. We first show that (8) implies the “local consistency” of $\text{next}_{\text{ins}, A}^\#$ relative to next_{ins} , i.e., $\text{next}_{\text{ins}}(\gamma_A(\widehat{S})) \leq \gamma_A(\text{next}_{\text{ins}, A}^\#(\widehat{S}))$.

Choose an arbitrary $l' \in \mathcal{L}$. Then:

$$\begin{aligned} \text{next}_{\text{ins}}(\gamma_A(\widehat{S}))(l') &\stackrel{\text{Def.}}{=} \bigcup_{\langle l, l' \rangle \in E} \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S})(l) \\ &\quad \wedge b = \text{eff}_C(l, l') \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ &\stackrel{(8)}{\subseteq} \bigcup_{\langle l, l' \rangle \in E} \gamma_A(\text{update}_A^\#(\widehat{S}(l), \text{eff}_C(l, l'))) \\ &\stackrel{(7)}{\subseteq} \gamma_A(\bigsqcup_{\langle l, l' \rangle \in E} \text{update}_A^\#(\widehat{S}(l), \text{eff}_C(l, l'))) \\ &\stackrel{\text{Def.}}{=} \gamma_A(\text{next}_{\text{ins}, A}^\#(\widehat{S})(l')) \end{aligned}$$

¹⁰ Note that in a complete lattice $\langle L, \sqsubseteq \rangle$ the partial order \sqsubseteq uniquely defines the join operator \sqcup . Vice versa, a given join operator uniquely defines a corresponding partial order. Nevertheless, we explicitly provide both partial order and join operator here and in the following.

From this it follows that $\gamma_A(\widehat{StickyCol}_A(P_{ins}))$ is a post fixpoint of $next_{ins}$:

$$\begin{aligned}
\gamma_A(\widehat{StickyCol}_A(P_{ins})) & \stackrel{\text{fixpoint}}{=} \gamma_A(\widehat{Init}_A \sqcup_A next_{ins,A}^\#(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{(7)}{\geq} \gamma_A(\widehat{Init}) \vee \gamma_A(next_{ins,A}^\#(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{\text{"local consistency"}}{\geq} \gamma_A(\widehat{Init}) \vee next_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{(6)}{\geq} Init \vee next_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins})))
\end{aligned}$$

In order to apply Knaster-Tarski's fixpoint theorem, we need to show that the domain of the sticky cache trace collecting semantics $(\mathcal{L} \rightarrow 2^{CacheTraces}, \leq)$ is a complete lattice, and that $next_{ins}$ is a monotone function. The power set 2^A of any set A is a complete lattice with respect to the subset relation \subseteq . Thus $(2^{CacheTraces}, \subseteq)$ is a complete lattice. Also, the total function space $A \rightarrow B$ between a set A and a complete lattice (B, \leq) is a complete lattice w.r.t. to the pointwise ordering $f \leq g : \Leftrightarrow \forall a \in A : f(a) \leq g(a)$. Thus $\mathcal{L} \rightarrow 2^{CacheTraces}$ is a complete lattice w.r.t. to \leq , as it is defined in Section 3.2.

To see that $next_{ins}$ is monotone w.r.t. \leq , first observe that the lifting $F(X) := \{f(x) \mid x \in X\}$ of any function f to sets is a monotone function w.r.t. \subseteq , i.e., if $X \subseteq Y$, then also $F(X) \subseteq F(Y)$. Thus $F(l, l') := \{t.c(b, h)c' \mid t.c \in X \wedge b = eff_{\mathcal{L}}(l, l') \wedge h = eff_C(c, b) \wedge c' = update_C(c, b)\}$ is monotone w.r.t. \subseteq for any l, l' . Also, the union of multiple monotone functions w.r.t. \subseteq is monotone w.r.t. \subseteq . Finally, the pointwise application of monotone functions w.r.t. \leq is monotone w.r.t. its pointwise extension \leq , as defined in Section 3.2, and so $next_{ins}$ is monotone. Further, any constant function is monotone w.r.t. to any order. Thus, the pointwise union of the constant function $Init$ and $next_{ins}$ is monotone as well.

Applying Knaster-Tarski's fixpoint theorem to the complete lattice $(\mathcal{L} \rightarrow 2^{CacheTraces}, \leq)$ and the monotone function $Init \vee next_{ins}$, we get that its post fixpoint $\gamma_A(\widehat{StickyCol}_A(P_{ins}))$ is greater than or equal to its least fixpoint

$$\begin{aligned}
StickyCol(P_{ins}) & \stackrel{\text{Def.}}{=} lfp_{Init}^{\leq} next_{ins} \\
& = lfp^{\leq}(Init \vee next_{ins}) \\
& \stackrel{\text{Knaster-Tarski}}{=} \bigwedge \{x \mid x \geq Init \vee next_{ins}(x)\}.
\end{aligned}$$

► **Theorem 4** (Soundness of Persistence Classification). *If the cache trace abstraction A satisfies conditions (6), (7), (8) from Theorem 3, and $classify_A^\#$ satisfies*

$$\begin{aligned}
\forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : classify_A^\#(\widehat{S}, b) \Rightarrow \\
\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \quad (10)
\end{aligned}$$

then $classify_A^\#(P_{ins}, b) := \forall l \in \mathcal{L} : classify_A^\#(\widehat{StickyCol}_A(P_{ins})(l), b)$ implies the persistence of memory block b in program P_{ins} .

Proof. Proof by contradiction. Assume that $\forall l \in \mathcal{L} : classify_A^\#(\widehat{StickyCol}_A(P_{ins})(l), b)$ holds for some memory block b , but b is not persistent in P_{ins} according to Definition 1. Then, there must be a trace $\tau = \langle l_0, c_0 \rangle e_0 \langle l_1, c_1 \rangle e_1 \dots e_{n-1} \langle l_n, c_n \rangle \in Col(P_{ins})$, such that $AtMostOneMiss(\tau, b)$ does not hold. Let i and j be such that $e_i = e_j = \langle b, miss \rangle$ and $i < j$.

By conditions (6), (7), and (8), Theorem 3 holds, and so

$$Col(P_{ins}) \subseteq \gamma_{ins}(StickyCol(P_{ins})) \subseteq \gamma_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins}))).$$

So $\tau \in \gamma_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins})))$. By (4), this implies that

$$c_0 e_0 \dots c_j \in \gamma_A(\widehat{StickyCol}_A(P_{ins}))(l_j) = \gamma_A(\widehat{StickyCol}_A(P_{ins}))(l_j).$$

By assumption $\text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l_j), b)$ holds, and so due to (10), we have

$$b \in c_j \vee (\forall i, 0 \leq i < j : b_i \neq b).$$

As $e_i = \langle b, \text{miss} \rangle$, we can exclude the second part of the disjunction. However, $b \in c_j$ contradicts $e_j = \langle b, \text{miss} \rangle$, which concludes the proof. \blacktriangleleft

► **Definition 5** (Precision). *Given two cache trace abstractions A and B , we say that A is at least as precise as B , denoted by $A \succeq B$, if A classifies each block as persistent that B classifies as persistent:*

$$\forall P_{ins}, \forall b : \text{classify}_B^\#(P_{ins}, b) \Rightarrow \text{classify}_A^\#(P_{ins}, b).$$

We say that A is more precise than B , denoted by $A \succ B$, if $A \succeq B$, but $B \not\succeq A$. If neither $A \succeq B$ nor vice versa, we say that A and B are incomparable.

► **Theorem 6** (Approximation of Abstract Semantics). *Given two cache trace abstractions A and B , and a function $\gamma_{B \rightarrow A} : C_B^\# \rightarrow C_A^\#$ that satisfies the following conditions:*

$$\widehat{\mathcal{I}}_A \subseteq \gamma_{B \rightarrow A}(\widehat{\mathcal{I}}_B), \quad (11)$$

$$\forall \widehat{S}, \widehat{T} \in C_B^\# : \widehat{S} \sqsubseteq_B \widehat{T} \Rightarrow \gamma_{B \rightarrow A}(\widehat{S}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{T}), \quad (12)$$

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}, b)). \quad (13)$$

Then, B 's abstract semantics soundly approximates its more concrete counterpart:

$$\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})), \quad (14)$$

where $\gamma_{B \rightarrow A}$ is lifted to the abstract sticky trace collecting semantics as follows:

$$\gamma_{B \rightarrow A}(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_{B \rightarrow A}(\widehat{S}(l)).$$

Proof. We first show that (13) implies the “local consistency” of $\text{next}_{ins,B}^\#$ relative $\text{next}_{ins,A}^\#$, i.e., $\text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{S})) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{S}))$.

Choose an arbitrary $l' \in \mathcal{L}$. Then:

$$\begin{aligned} \text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{S}))(l') &\stackrel{\text{Def.}}{=} \bigsqcup_{(l,l') \in E} \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}(l)), \text{eff}_{\mathcal{L}}(l, l')) \\ &\stackrel{(13)}{\sqsubseteq_A} \bigsqcup_{(l,l') \in E} \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}(l), \text{eff}_{\mathcal{L}}(l, l'))) \\ &\stackrel{(12)}{\sqsubseteq_A} \gamma_{B \rightarrow A}(\bigsqcup_{(l,l') \in E} \text{update}_B^\#(\widehat{S}(l), \text{eff}_{\mathcal{L}}(l, l'))) \\ &\stackrel{\text{Def.}}{=} \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{S})(l')) \end{aligned}$$

From this it follows that $\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))$ is a post fixpoint of $\text{next}_{ins,A}^\#$:

$$\begin{aligned} \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})) &\stackrel{\text{fixpoint}}{=} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B \sqcup_B \text{next}_{ins,B}^\#(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{(12)}{\sqsupseteq_A} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B) \sqcup_A \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{\text{“local consistency”}}{\sqsupseteq_A} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B) \sqcup_A \text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{(11)}{\sqsupseteq_A} \widehat{\text{Init}}_A \sqcup_A \text{next}_{ins}(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))) \end{aligned}$$

In order to apply Knaster-Tarski's fixpoint theorem, we need to show that the domain of abstraction A , $(\mathcal{L} \rightarrow C_A^\#, \sqsubseteq_A)$ is a complete lattice, and that $\text{next}_{ins,A}^\#$ is a monotone function. The total function space $A \rightarrow B$ between a set A and a complete lattice (B, \leq) is a complete lattice w.r.t.

to the pointwise ordering $f \leq g \Leftrightarrow \forall a \in A : f(a) \leq g(a)$. Thus $\mathcal{L} \rightarrow C_A^\#$ is a complete lattice w.r.t. to \sqsubseteq_A , as it is defined in Section 4.1.

To see that $next_{ins,A}^\#$ is monotone w.r.t. \sqsubseteq_A , observe that by assumption $update_A^\#$ is monotone in its first parameter. Thus, $F_{l,l'}(X) := update_A^\#(\widehat{S}(l), eff_{\mathcal{L}}(l, l'))$ is monotone in \widehat{S} for any l, l' . Also, the least upper bound of multiple monotone functions is a monotone function, and so $F(l') := \bigsqcup_{(l,l') \in E} \{update_A^\#(\widehat{S}(l), b) \mid b = eff_{\mathcal{L}}(l, l')\}$ is monotone in \widehat{S} . Finally, the pointwise application of monotone functions w.r.t. \leq is monotone w.r.t. its pointwise extension \leq , and so $next_{ins,A}^\# = \lambda l' \in \mathcal{L}. F(l')$ is monotone. Further, any constant function is monotone w.r.t. to any order. Thus, the pointwise union of the constant function \widehat{Init}_A and $next_{ins,A}^\#$ is monotone as well.

Applying Knaster-Tarski's fixpoint theorem to the complete lattice $(\mathcal{L} \rightarrow C_A^\#, \sqsubseteq_A)$ and the monotone function $\widehat{Init}_A \sqcup_A next_{ins,A}^\#$, we get that its post fixpoint $\gamma_{B \rightarrow A}(\widehat{StickyCol}_B(P_{ins}))$ is greater than or equal to its least fixpoint:

$$\begin{aligned} \widehat{StickyCol}_A(P_{ins}) &\stackrel{\text{Def.}}{=} lfp_{\widehat{Init}_A}^{\sqsubseteq_A} next_{ins,A}^\# \\ &= lfp^{\sqsubseteq_A} \widehat{Init}_A \sqcup_A next_{ins,A}^\# \\ &\stackrel{\text{Knaster-Tarski}}{=} \bigsqcap_A \{x \mid x \sqsubseteq_A \widehat{Init}_A \sqcup_A next_{ins,A}^\#(x)\}. \end{aligned} \quad \blacktriangleleft$$

Whenever the proof of a theorem is in the main part of the article, the name of the theorem is marked with a \star and serves as a link to the corresponding proof. The first example of such a case is the following theorem:

► **Theorem 7 (Precision \star).** *Given cache trace abstractions A, B and a function $\gamma_{B \rightarrow A}$ that satisfies conditions (11), (12), and (13) from Theorem 6, and further*

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : classify_B^\#(\widehat{S}, b) \Rightarrow classify_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b), \quad (15)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\#, b \in \mathcal{B} : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow (classify_A^\#(\widehat{T}, b) \Rightarrow classify_A^\#(\widehat{S}, b)). \quad (16)$$

Then, A is at least as precise as B , i.e., $A \succeq B$.

► **Theorem 8 (Soundness of Persistence Classification \star).** *Given two cache trace abstractions A and B . If A is sound, and A is at least as precise as B , then B is also sound.*

► **Lemma 31 (Monotonicity of LRU).** *Consider an arbitrary cache trace $c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$. Assume that $c_0(b) > c_0(b')$ and $b \notin \{b_0, b_1, \dots, b_{n-1}\}$. Then:*

$$\forall i, 0 \leq i \leq n : c_i(b) > c_i(b').$$

Proof. Proof by induction over i :

■ Base case ($i = 0$):

$c_0(b) > c_0(b')$ holds by assumption.

■ Inductive step:

We must show that $c_{i+1}(b) > c_{i+1}(b')$.

By the inductive hypothesis (I.H.) we have $c_i(b) > c_i(b')$.

We distinguish two cases:

1. $c_i(b) > c_i(b') + 1$:

By the definition of $update_{\mathcal{C}}^{LRU}$ we have $c_i(b') + 1 \geq c_{i+1}(b')$.

As by assumption $b \neq b_i$, it also follows from the definition of $update_{\mathcal{C}}^{LRU}$ that $c_{i+1}(b) \geq c_i(b)$.

Thus, $c_{i+1}(b) \geq c_i(b) > c_i(b') + 1 \geq c_{i+1}(b')$.

2. $c_i(b) = c_i(b') + 1$:

We distinguish four cases based on the value of $c_i(b_i)$:

- $c_i(b_i) < c_i(b')$:

By the definition of $update_C^{LRU}$ (third case), $c_{i+1}(b') = c_i(b')$ and $c_{i+1}(b) = c_i(b)$.

And so $c_{i+1}(b) = c_i(b) \stackrel{\text{I.H.}}{>} c_i(b') = c_{i+1}(b')$.

- $c_i(b_i) = c_i(b')$:

This implies that $b_i = b'$. Thus, by the definition of $update_C^{LRU}$, $c_{i+1}(b') \stackrel{\text{first case}}{=} 0 < c_i(b') + 1 = c_{i+1}(b) \stackrel{\text{third case}}{=} c_i(b)$.

- $c_i(b_i) = c_i(b)$:

This implies that $b_i = b$, which contradicts our assumption that $b \notin \{b_0, b_1, \dots, b_{n-1}\}$.

- $c_i(b_i) > c_i(b')$:

By the definition of $update_C^{LRU}$, $c_{i+1}(b') \stackrel{\text{second case}}{=} c_i(b') + 1$ and $c_{i+1}(b) \stackrel{\text{second case}}{=} c_i(b) + 1$.

And so $c_{i+1}(b) = c_i(b) + 1 \stackrel{\text{I.H.}}{>} c_i(b') + 1 = c_{i+1}(b')$. ◀

► **Lemma 9** (Persistence under LRU). *Consider an arbitrary cache trace $c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$. Then $c_n(b_0) < k$, if $|\{b_i \mid 0 \leq i < n\}| \leq k$.*

Proof. Let $s = c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$ be an arbitrary cache trace and assume that $B = \{b_i \mid 0 \leq i < n\}$ with $|B| \leq k$. We need to show that $c_n(b_0) < k$.

Let j be the index of the last occurrence of b_0 in s , i.e., $b_j = b_0$ and $\forall l > j : b_l \neq b_0$. Observe that $c_{j+1}(b_0) = 0$, because of the preceding access to $b_0 = b_j$. Let $B_j = \{b_i \mid j < i < n\}$. By construction, $b_0 \notin B_j$. As $b_0 \in B$ and $B_j \subseteq B$, we have $|B_j| < |B| \leq k$ and thus $|B_j| < k$.

Each memory block in B_j occurs one or more times in the suffix $s_j = c_{j+1}\langle b_{j+1}, h_{j+1} \rangle \dots c_n$. Let I be the set of indices of the first occurrences of the blocks in B_j in s_j , i.e.,

$$I = \{i \mid j < i < n \wedge \forall l, j < l < i : b_l \neq b_i\}.$$

Let I^C be the complement of I , i.e., $I^C = \{j + 1, \dots, n - 1\} \setminus I$. We claim that

1. $c_{i+1}(b_0) \leq c_i(b_0) + 1$ for all $i \in I$, and
2. $c_{t+1}(b_0) = c_t(b_0)$ for all $t \in I^C$.

These two claims imply that $c_n(b_0) \leq c_{j+1}(b_0) + |I| = |I|$. As $|I| = |B_j| < k$, $c_n(b_0) = |I| < k$, and it only remains to show the two claims:

1. The fact that $c_{i+1}(b_0) \leq c_i(b_0) + 1$ follows immediately from the definition of $update_C^{LRU}$.
2. Let t be an arbitrary index in I^C and let v be the greatest index smaller than t such that $b_v = b_t$. As $t \in I^C$ there must be such a $v > j$ due to the definitions of I and I^C .

Observe that $c_{v+1}(b_t) = c_{v+1}(b_v) = 0$ and $c_{v+1}(b_0) > 0$. Applying Lemma 31 to the subsequence $c_{v+1}\langle b_{v+1}, h_{v+1} \rangle \dots c_t$ with $b = b_0$ and $b' = b_t$ yields that $c_t(b_0) > c_t(b_t)$.

As $c_t(b_0) > c_t(b_t)$, the third case in $update_C^{LRU}$ applies and we get $c_{t+1}(b_0) = c_t(b_0)$. ◀

► **Theorem 10** (Soundness of Global May-Conflict Set). *Global-CS is a sound persistence analysis.*

Proof. We show that *Global-CS* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 are unconstrained in the definition of $\gamma_{\text{Global-CS}}$.
- Let $\hat{S} \subseteq_{\text{Global-CS}} \hat{T}$. Let $s = c_0\langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{\text{Global-CS}}(\hat{S})$. Then $\{b_i \mid 0 \leq i < n\} \subseteq \hat{S} \subseteq \hat{T}$ and so by definition of $\gamma_{\text{Global-CS}}$, $s \in \gamma_{\text{Global-CS}}(\hat{T})$, which shows that (7) is satisfied.

- Let $\hat{S} \in C_{Global-CS}^\#$, $b \in \mathcal{B}$, and $t.c \in \gamma_{Global-CS}(\hat{S})$ be arbitrary.
 To show that (8) is satisfied, we have to show that $t.c\langle b, h \rangle c'$ with $h = \text{eff}_C^{LRU}(c, b)$ and $c' = \text{update}_C^{LRU}(c, b)$ is an element of $\gamma_{Global-CS}(\text{update}_{Global-CS}^\#(\hat{S}, b))$.
 By definition of $\gamma_{Global-CS}$ and $\text{update}_{Global-CS}^\#$ we have

$$\begin{aligned} & \gamma_{Global-CS}(\text{update}_{Global-CS}^\#(\hat{S}, b)) \\ & \stackrel{\text{Def. } \text{update}_{Global-CS}^\#}{=} \gamma_{Global-CS}(\hat{S} \cup \{b\}) \\ & \stackrel{\text{Def. } \gamma_{Global-CS}}{=} \text{LRUCACHETRACES} \cap \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \hat{S} \cup \{b\}\} \end{aligned}$$
 Because $t.c \in \gamma_{Global-CS}(\hat{S})$, we have that $t.c \in \text{LRUCACHETRACES}$. From $h = \text{eff}_C^{LRU}(c, b)$ and $c' = \text{update}_C^{LRU}(c, b)$, it follows that $t.c\langle b, h \rangle c' \in \text{LRUCACHETRACES}$.
 It remains to show that $t.c\langle b, h \rangle c' \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \hat{S} \cup \{b\}\}$.
 As $t.c \in \gamma_{Global-CS}(\hat{S})$, we have that $t.c \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \hat{S}\}$. So $t.c\langle b, h \rangle c' \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \hat{S} \cup \{b\}\}$.
- Let $\hat{S} \in C_{Global-CS}^\#$ and $b \in \mathcal{B}$ be arbitrary.
 To show that (10) is satisfied, we consider two cases: 1. $b \notin \hat{S}$ and 2. $b \in \hat{S}$.
Case 1: If $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\hat{S})$, then $\{b_i \mid 0 \leq i < n\} \subseteq \hat{S}$ by the definition of $\gamma_{Global-CS}$. As $b \notin \hat{S}$, the second disjunct in (10) holds: $\forall i, 0 \leq i < n : b_i \neq b$.
Case 2: Let $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\hat{S})$. Assume $b_i = b$ for some i . Otherwise the second disjunct of (10) holds. As $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\hat{S})$, in particular $\{b_j \mid i \leq j < n\} \subseteq \hat{S}$. As $|\hat{S}| \leq k$ and $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the trace $c_i\langle b_i, h_i \rangle c_{i+1} \dots c_n$ to conclude that $b \in c_n$. ◀

► **Theorem 11** (Soundness of Block-wise May-Conflict Set). *Block-CS is a sound persistence analysis.*

Proof. We show that *Block-CS* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 are unconstrained in the definition of $\gamma_{Block-CS}$.
- Let $\hat{S} \sqsubseteq_{Block-CS} \hat{T}$. Let $s = c_0\langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{Block-CS}(\hat{S})$. Because $\hat{S}(b_i) \subseteq \hat{T}(b_i)$ for all i , and \subseteq is transitive, s is also an element of $\gamma_{Block-CS}(\hat{T})$, which shows that (7) is satisfied.
- Let $\hat{S} \in C_{Block-CS}^\#$, $b \in \mathcal{B}$, and $s = c_0\langle b_0, h_0 \rangle \dots c_n \in \gamma_{Block-CS}(\hat{S})$ be arbitrary.
 To show that (8) is satisfied, we have to show that $t = c_0\langle b_0, h_0 \rangle \dots c_n\langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{Block-CS}(\hat{T})$, with $\hat{T} = \text{update}_{Block-CS}^\#(\hat{S}, b_n)$.
 Because $s \in \gamma_{Block-CS}(\hat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.\langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .
 It remains to show that the second constraint in (25) holds¹¹, i.e.,

$$\forall i, 0 \leq i < n+1 : b_i \in CS_{i+1}(t) \vee CS_i(t) \subseteq \hat{T}(b_i), \quad (58)$$

where $CS_i(c_0\langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n+1\}$.

In order to show that (58) holds, we distinguish two cases based on the value of i :

¹¹ The constraint below accounts for the fact that t contains $n+1$ accesses, where n is the number of accesses in s .

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

Due to the second case in the definition of $update_{Block-CS}^\#$, $\widehat{T}(b_n) = \{b_n\}$, and so

$$CS_n(t) = \{b_n\} \subseteq \{b_n\} = \widehat{T}(b_n).$$

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee CS_i(s) \subseteq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{Block-CS}(\widehat{S})$.

We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $CS_i(s) \subseteq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

Then $CS_i(t) = CS_i(s) \cup \{b_n\} \subseteq \widehat{S}(b_i) \cup \{b_n\} = \widehat{T}(b_i)$ as the third case in $update_{Block-CS}^\#$ applies:

$CS_i(s) \neq \emptyset$ and thus $\widehat{S}(b_i) \neq \emptyset$ and $b_i \neq b_n$.

ii. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_i\}$.

■ Let $\widehat{S} \in C_{Block-CS}^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{Block-CS}^\#(\widehat{S}, b)$ holds and thus $|\widehat{S}(b)| \leq k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary cache trace in $\gamma_{Block-CS}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $CS_i(s) \subseteq \widehat{S}(b)$. As $|\widehat{S}(b)| \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 12** (*Block-CS vs. Global-CS**). *Block-CS is more precise than Global-CS.*

► **Theorem 13** (*Soundness of Conditional Must*). *C-Must is a sound persistence analysis.*

Proof. We show that *C-Must* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

■ Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 only are unconstrained in the definition of γ_{C-Must} .

■ Let $\widehat{S} \sqsubseteq_{C-Must} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{C-Must}(\widehat{S})$. Because $\widehat{S}(b_i) \leq \widehat{T}(b_i)$ for all i , and \leq is transitive, s is also an element of $\gamma_{C-Must}(\widehat{T})$, which shows that (7) is satisfied.

■ Let $\widehat{S} \in C_{C-Must}^\#$, $b \in \mathcal{B}$, and $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S})$ be arbitrary.

To show that (8) is satisfied, we have to show that $t = c_0 \langle b_0, h_0 \rangle \dots c_n \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C-Must}(\widehat{T})$, with $\widehat{T} = \text{update}_{C-Must}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{C-Must}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s. \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (31) holds, i.e.,

$$\forall i, 0 \leq i < n+1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \widehat{T}(b_i), \quad (59)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n+1\}$.

In order to show that (59) holds, we distinguish two cases based on the value of i :

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

Due to the second case in the definition of $update_{C-Must}^\#$, $\widehat{T}(b_n) = 1$, and so

$$|CS_n(t)| = |\{b_n\}| = 1 \leq 1 = \widehat{T}(b_n).$$

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S})$.

We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

Then $CS_i(t) = CS_i(s) \cup \{b_n\}$.

Because $1 \leq |CS_i(s)| \leq \widehat{S}(b_i)$ and $b_i \neq b_n$, the third or fourth case in $update_{C-Must}^\#$ applies. Thus $|CS_i(t)| = |CS_i(s) \cup \{b_n\}| \leq \widehat{S}(b_i) + 1 \leq \widehat{T}(b_i)$.

ii. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_i\}$.

■ Let $\widehat{S} \in C-Must^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{C-Must}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) < k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{C-Must}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $|CS_i(s)| \leq \widehat{S}(b)$. As $\widehat{S}(b) \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 14** (*Global-CS vs. Block-CS**). *C-Must is incomparable to Global-CS and Block-CS.*

► **Theorem 15** (*Soundness of Conditional May*). *C-May is a sound persistence analysis.*

Proof. We show that *C-May* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

■ Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 only are unconstrained in the definition of γ_{C-May} .

■ Let $\widehat{S} \sqsubseteq_{C-May} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{C-May}(\widehat{S})$. Because $\widehat{S}(b_i) \geq \widehat{T}(b_i)$ for all i , and \geq is transitive, s is also an element of $\gamma_{C-May}(\widehat{T})$, which shows that (7) is satisfied.

■ Let $\widehat{S} \in C-May^\#$, $b \in \mathcal{B}$, and $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-May}(\widehat{S})$ be arbitrary.

To show that (8) is satisfied, we have to show that $t = c_0 \langle b_0, h_0 \rangle \dots c_n \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C-May}(\widehat{T})$, with $\widehat{T} = \text{update}_{C-May}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{C-May}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s \cdot \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (37) holds, i.e.,

$$\forall i : 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \geq \widehat{T}(b_i), \quad (60)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n + 1\}$.

In order to show that (60) holds, we distinguish two cases based on the value of i :

1. $i = n$:

$CS_n(t) = \{b_n\}$. Due to the first case in the definition of $update_{C-May}^\#$, $\widehat{T}(b_n) = 1$, and so $|CS_n(t)| \geq \widehat{T}(b_n)$.

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \geq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-May}(\widehat{S})$. We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \geq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

We distinguish two cases:

A. $\widehat{S}(b_n) < \widehat{S}(b_i)$:

Then $|CS_i(t)| \geq |CS_i(s)| \geq \widehat{S}(b_i) = \widehat{T}(b_i)$, as the second case of $update_{C-May}^\#$ applies.

B. $\widehat{S}(b_n) \geq \widehat{S}(b_i)$:

Then, by the definition of $update_{C-May}^\#$, $\widehat{T}(b_i) \leq \widehat{S}(b_i) + 1$.

Let j be the index of the last occurrence of b_n in s . We distinguish two cases:

– $i > j$:

Then $b_j \notin CS_i(s)$. Thus

$$|CS_i(t)| = |CS_i(s) \dot{\cup} \{b_n\}| = |CS_i(s)| + 1 \geq \widehat{S}(b_i) + 1 \geq \widehat{T}(b_i).$$

– $i < j$:

Then $b_i, b_j \in CS_i(s)$ and $b_i \notin CS_j(s)$ and $CS_i(s) \supseteq CS_j(s)$. Thus

$$|CS_i(t)| = |CS_i(s) \cup \{b_j\}| \geq |CS_j(s)| + 1 \geq \widehat{S}(b_j) + 1 = \widehat{S}(b_n) + 1 \geq \widehat{T}(b_i).$$

ii. $b_i = b_n$:

Then $b_i \in CS_i(t) = CS_i(s) \cup \{b_i\}$.

– Let $\widehat{S} \in C_{C-May}^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{C-May}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) = \infty$ or $|C_i(\widehat{S}, b)| < i$ for some $i \leq k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{C-May}(\widehat{S})$. Let i' be the index of the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. If $\widehat{S}(b) = \infty$ holds, then b is guaranteed not to occur in the trace s .

Otherwise, $|C_i(\widehat{S}, b)| = |\{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{S}(b') \leq i\}| < i$ and $b_{i'} \notin CS_{i'+1}(s)$.

We will show that $|CS_{i'}(s)| \leq i$.

Assume for a contradiction that $|CS_{i'}(s)| > i$. Let $j > i'$ be an index such that $|CS_j(s)| = i$, which must then exist as $|CS_{i'}(s)| > i$ and $CS_i(s)$ is monotonically decreasing in i and eventually reaches $|CS_n(s)| = 1$.

For each element b' of $CS_j(s)$, we must have $\widehat{S}(b') \leq i$ as $s \in \gamma_{C-May}(\widehat{S})$. So $CS_j(s) \subseteq C_i(\widehat{S}, b)$. Thus $i = |CS_j(s)| \leq |C_i(\widehat{S}, b)|$, which contradicts the fact that $|C_i(\widehat{S}, b)| < i$.

Thus $|CS_{i'}(s)| \leq i$. As $i \leq k$ and $s \in \text{LRUCACHE TRACES}$, we can apply Lemma 9 to the suffix $c_{i'} \langle b_{i'}, h_{i'} \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 16 (C-May vs. Global-CS).** *C-May is more precise than Global-CS.*

Proof. We will show this by making use of Theorem 7. To this end, we need to define a function $\gamma_{CS \rightarrow May} : C_{Global-CS}^{\#} \rightarrow C_{C-May}^{\#}$ that satisfies conditions (11), (12), (13), (15), and (16).

We define $\gamma_{CS \rightarrow May}$ as follows:

$$\gamma_{CS \rightarrow May}(\widehat{S}) := \lambda b. \begin{cases} \infty & : b \notin \widehat{S} \\ 1 & : b \in \widehat{S} \end{cases} \quad (61)$$

The rationale is that if $b \notin \widehat{S}$ then it has not yet been accessed and thus ∞ is a sound lower bound on the size of b 's conflict set. On the other hand, if $b \in \widehat{S}$, and thus may have been accessed, then 1 is the best sound lower bound on the size of b 's conflict set that can be given, as the access to b may have been the final one in the cache trace.

- Proof of satisfaction of (11): $\gamma_{CS \rightarrow May}(\widehat{\mathcal{I}_{Global-CS}}) = \gamma_{CS \rightarrow May}(\emptyset) = \lambda b. \infty = \widehat{\mathcal{I}_{C-May}}$.
- Proof of satisfaction of (12): Let \widehat{S}, \widehat{T} be arbitrary abstract traces from $C_{Global-CS}^{\#}$. Assume $\widehat{S} \sqsubseteq_{Global-CS} \widehat{T}$, i.e., $\widehat{S} \subseteq \widehat{T}$.

Then $\forall b \in \widehat{S} : \gamma_{CS \rightarrow May}(\widehat{S})(b) = 1 = \gamma_{CS \rightarrow May}(\widehat{T})(b)$ and

$$\forall b \notin \widehat{S} : \gamma_{CS \rightarrow May}(\widehat{S})(b) = \infty \geq \gamma_{CS \rightarrow May}(\widehat{T})(b),$$

which implies

$$\forall b : \gamma_{CS \rightarrow May}(\widehat{S})(b) \geq \gamma_{CS \rightarrow May}(\widehat{T})(b), \text{ i.e., } \gamma_{CS \rightarrow May}(\widehat{S}) \sqsubseteq_{C-May} \gamma_{CS \rightarrow May}(\widehat{T}),$$

which shows (12).

- Proof of satisfaction of (13): We need to show that

$$\begin{aligned} \forall \widehat{S} \in C_{Global-CS}^{\#}, b \in \mathcal{B} : \\ \text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b) \sqsubseteq_{C-May} \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b)). \end{aligned}$$

Let $\widehat{S} \in C_{Global-CS}^{\#}$ and $b \in \mathcal{B}$ be arbitrary.

Due to the definition of \sqsubseteq_{C-May} , we need to show

$$\forall b' \in \mathcal{B} : \text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') \geq \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

To prove this, let $b' \in \mathcal{B}$ be arbitrary.

We distinguish two cases:

1. $b' \in \text{update}_{Global-CS}^{\#}(\widehat{S}, b)$:
Then $\gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b') = 1$, which is the smallest value that a block may be assigned to in $C_{C-May}^{\#}$, and so

$$\text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') \geq 1 = \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

2. $b' \notin \text{update}_{Global-CS}^{\#}(\widehat{S}, b)$:
Then, $b' \neq b$ and $b' \notin \widehat{S}$. Thus, $\gamma_{CS \rightarrow May}(\widehat{S})(b') = \infty$ and the fifth case in the definition of $\text{update}_{C-May}^{\#}$ applies, so

$$\text{update}_{Global-CS}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') = \infty \geq \infty = \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

- Proof of (15): Let $\widehat{S} \in C_{Global-CS}^{\#}$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{Global-CS}^{\#}(\widehat{S}, b)$ holds. Then, either $b \notin \widehat{S}$ or $|\widehat{S}| \leq k$:

- If $b \notin \hat{S}$, then $\gamma_{CS \rightarrow May}(\hat{S})(b) = \infty$ and by definition, $classify_{C-May}^\#(\gamma_{CS \rightarrow May}(\hat{S}), b)$ holds as well.
- If $b \in \hat{S}$ and thus $|\hat{S}| \leq k$ we have exactly $|\hat{S}|$ blocks b' for which $\gamma_{CS \rightarrow May}(\hat{S})(b) \leq k$. Thus, the second disjunct of $classify_{C-May}^\#(\gamma_{CS \rightarrow May}(\hat{S}), b)$ holds.
- Proof of satisfaction of (16): Let $\hat{S}, \hat{T} \in C_{C-May}^\#$ with $\hat{S} \sqsubseteq_{C-May} \hat{T}$ and $b \in \mathcal{B}$ be arbitrary. Assume $classify_{C-May}^\#(\hat{T}, b)$ holds.
 - If $\hat{T}(b) = \infty$, then $\hat{S}(b) = \infty$ as $\hat{S}(b) \geq \hat{T}(b)$. Then, $classify_{C-May}^\#(\hat{S}, b)$ holds as well.
 - If $\hat{T}(b) \leq k + 1$, then there is an $i \leq k$, such that $|C_i(\hat{T}, b)| < i$. As $\hat{S} \sqsubseteq_{C-May} \hat{T}$ we have $\hat{S}(b') \geq \hat{T}(b')$ for all $b' \in \mathcal{B}$. So $C_i(\hat{S}, b) = \{b' \in \mathcal{B} \mid b' \neq b \wedge \hat{T}(b') \leq \hat{S}(b') \leq i\} \subseteq C_i(\hat{T}, b)$, and thus $|C_i(\hat{S}, b)| \leq |C_i(\hat{T}, b)| < i$, which implies $classify_{C-May}^\#(\hat{S}, b)$.

To see that $C-May$ is more precise than $Global-CS$, consider the example in Figure 4a. Here, x is classified as persistent by $C-May$, but not by $Global-CS$. ◀

► **Theorem 17** (*Block-CS vs. C-May*). *Block-CS is more precise than C-May.*

Proof. We will show this by making use of Theorem 7. To this end, we need to define a function $\gamma_{May \rightarrow CS} : C_{C-May}^\# \rightarrow C_{Block-CS}^\#$ that satisfies conditions (11), (12), (13), (15), and (16).

We define $\gamma_{May \rightarrow CS}$ as follows:

$$\gamma_{May \rightarrow CS}(\hat{S}) := \lambda b. \begin{cases} \emptyset & : \hat{S}(b) = \infty \\ \{b\} \cup C_n(\hat{S}, b) & : \hat{S}(b) \neq \infty \wedge n = \min\{i \in \mathbb{N} \mid |C_i(\hat{S}, b)| < i\} \end{cases} \quad (62)$$

where $C_i(\hat{S}, b) := \{b' \in \mathcal{B} \mid b' \neq b \wedge \hat{S}(b') \leq i\}$.

- Proof of satisfaction of (11): $\gamma_{May \rightarrow CS}(\widehat{\mathcal{I}_{C-May}}) = \gamma_{May \rightarrow CS}(\lambda b. \infty) = \lambda b. \emptyset = \widehat{\mathcal{I}_{Block-CS}}$.
- Proof of satisfaction of (12): Let \hat{S}, \hat{T} be arbitrary abstract traces from $C_{C-May}^\#$. Assume $\hat{S} \sqsubseteq_{C-May} \hat{T}$, i.e., $\forall b : \hat{S}(b) \geq \hat{T}(b)$.

We need to show that

$$\gamma_{May \rightarrow CS}(\hat{S}) \sqsubseteq_{Block-CS} \gamma_{May \rightarrow CS}(\hat{T}) \Leftrightarrow \forall b : \gamma_{May \rightarrow CS}(\hat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\hat{T})(b).$$

Let b be arbitrary. We will show $\gamma_{May \rightarrow CS}(\hat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\hat{T})(b)$ by the following case distinction:

- $\hat{S}(b) = \infty$: Then $\gamma_{May \rightarrow CS}(\hat{S})(b) = \emptyset$, which is a subset of any set, in particular $\gamma_{May \rightarrow CS}(\hat{T})(b)$.
- $\hat{S}(b) \leq k + 1$: Let $n = \min\{i \in \mathbb{N} \mid |C_i(\hat{T}, b)| < i\}$ and thus $\gamma_{May \rightarrow CS}(\hat{T})(b) = \{b\} \cup C_n(\hat{T}, b)$. As $\forall b : \hat{S}(b) \geq \hat{T}(b)$, $C_i(\hat{S}, b) \subseteq C_i(\hat{T}, b)$ and so $|C_i(\hat{S}, b)| \leq |C_i(\hat{T}, b)| < i$. Thus, $\gamma_{May \rightarrow CS}(\hat{S})(b) = \{b\} \cup C_{n'}(\hat{S}, b)$, with $n' = \min\{i \in \mathbb{N} \mid |C_i(\hat{S}, b)| < i\} \leq n$. As CS_i is monotone in i and $CS_i(\hat{S}, b) \subseteq CS_i(\hat{T}, b)$, we have $\gamma_{May \rightarrow CS}(\hat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\hat{T})(b)$.

- Proof of satisfaction of (13): We need to show that

$$\forall \hat{S} \in C_{C-May}^\#, b \in \mathcal{B} : update_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\hat{S}), b) \sqsubseteq_{Block-CS} \gamma_{May \rightarrow CS}(update_{C-May}^\#(\hat{S}, b)).$$

Let \hat{S} and b be arbitrary, and let $\hat{T} = update_{C-May}^\#(\hat{S}, b)$. Then, we need to show for all $b' \in \mathcal{B}$:

$$update_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\hat{S}), b)(b') \subseteq \gamma_{May \rightarrow CS}(update_{C-May}^\#(\hat{S}, b))(b') = \gamma_{May \rightarrow CS}(\hat{T})(b').$$

To prove this we distinguish two cases:

1. $b' = b$:

Then $update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \{b'\}$ and as $update_{C-May}^{\#}(\widehat{S}, b)(b') \neq \infty$, we have $\gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b') \supseteq \{b'\}$, and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \{b'\} \subseteq \gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b').$$

2. $b' \neq b$:

We further distinguish two cases:

a. $\widehat{S}(b') = \infty$:

Then, it is easy to see that

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \emptyset = \gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b').$$

b. $\widehat{S}(b') \leq k + 1$:

Let $n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b')| < i\}$ and $n' = \min\{i \in \mathbb{N} \mid |C_i(\widehat{T}, b')| < i\}$.

We further distinguish three cases:

i. $\widehat{S}(b) < n$:

Then $b \in \gamma_{May \rightarrow CS}(\widehat{S})(b')$ and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \gamma_{May \rightarrow CS}(\widehat{S})(b').$$

Observe that

- $C_1(\widehat{T}, b') = \{b\}$,
- $C_j(\widehat{T}, b') = C_{j-1}(\widehat{S}, b') \dot{\cup} \{b\}$ for $j \in \{2, \dots, \widehat{S}(b)\}$, and
- $C_j(\widehat{T}, b') = C_j(\widehat{S}, b')$ for $j \in \{\widehat{S}(b) + 1, \dots, n\}$.

Due to the definition of n , we have that $|C_i(\widehat{S}, b')| \geq i$ for all $i < n$, and so:

- $|C_1(\widehat{T}, b')| \geq 1$,
- $|C_j(\widehat{T}, b')| \geq |C_{j-1}(\widehat{S}, b')| + 1 \geq j - 1 + 1 = j$ for $j \in \{2, \dots, \widehat{S}(b)\}$, and
- $|C_j(\widehat{T}, b')| = |C_j(\widehat{S}, b')| \geq j$ for $j \in \{\widehat{S}(b) + 1, \dots, n\}$.

As a consequence $n' \geq n$ and thus

$$\gamma_{May \rightarrow CS}(\widehat{T})(b') \supseteq \gamma_{May \rightarrow CS}(\widehat{S})(b') = update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b').$$

ii. $\widehat{S}(b) = n$:

This case is impossible:

As $n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b')| < i\}$, we have $|C_{n-1}(\widehat{S}, b)| \geq n - 1$.

However, $C_n(\widehat{S}, b') \supseteq C_{n-1}(\widehat{S}, b) \dot{\cup} \{b\}$, if $\widehat{S}(b) = n$, which implies $|C_i(\widehat{S}, b')| \geq n$, which contradicts of our definition of n , which implies $|C_n(\widehat{S}, b')| < n$.

iii. $\widehat{S}(b) > n$:

Then $b \notin \gamma_{May \rightarrow CS}(\widehat{S})(b')$ and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \gamma_{May \rightarrow CS}(\widehat{S})(b') \dot{\cup} \{b\}.$$

Observe that

- $C_1(\widehat{T}, b') = \{b\}$, and
- $C_j(\widehat{T}, b') = C_{j-1}(\widehat{S}, b') \dot{\cup} \{b\}$ for $j \in \{2, \dots, n\}$.

Due to the definition of n , we have that $|C_i(\widehat{S}, b')| \geq i$ for all $i < n$, and so:

- $|C_1(\widehat{T}, b')| \geq 1$, and
- $|C_j(\widehat{T}, b')| \geq |C_{j-1}(\widehat{S}, b')| + 1 \geq j - 1 + 1 = j$ for $j \in \{2, \dots, n\}$.

Thus $n' \geq n + 1$. Also, $C_{n+1}(\widehat{T}, b') \supseteq C_n(\widehat{S}, b') \dot{\cup} \{b\}$ and thus

$$\begin{aligned} \gamma_{May \rightarrow CS}(\widehat{T})(b') &\supseteq C_n(\widehat{S}, b') \dot{\cup} \{b\} \\ &= \gamma_{May \rightarrow CS}(\widehat{S})(b') \cup \{b\} = update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b'). \end{aligned}$$

- Proof of satisfaction of (15): Let $\hat{S} \in C_{C-May}^\#$ and $b \in \mathcal{B}$ be arbitrary. Assume $classify_{C-May}^\#(\hat{S}, b)$ holds.

Then either $\hat{S}(b) = \infty$ or $\exists i \leq k : |C_i(\hat{S}, b)| < i$.

In the first case, $\gamma_{May \rightarrow CS}(\hat{S})(b) = \emptyset$ and so $classify_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\hat{S}), b)$ holds.

In the second case, $\gamma_{May \rightarrow CS}(\hat{S})(b) = \{b\} \cup C_n(\hat{S}, b)$ with $|C_n(\hat{S}, b)| < k$ and thus

$|\gamma_{May \rightarrow CS}(\hat{S})(b)| \leq k$, which implies that $classify_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\hat{S}), b)$ holds as well.

- Proof of satisfaction of (16): Let $\hat{S}, \hat{T} \in C_{Block-CS}^\#$ with $\hat{S} \sqsubseteq_{Block-CS} \hat{T}$ and $b \in \mathcal{B}$ be arbitrary. Assume $classify_{Block-CS}^\#(\hat{T}, b)$ holds. Then, $|\hat{T}(b)| \leq k$. As $\hat{S}(b) \subseteq \hat{T}(b)$ this implies $|\hat{S}(b)| \leq k$ and so $classify_{Block-CS}^\#(\hat{S}, b)$ holds as well.

To see that *Block-CS* is more precise than *C-May*, consider the example in Figure 4b. Here, w and x are classified as persistent by *Block-CS*, but not by *C-May*. ◀

► **Definition 18** (Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is the tuple $A \times B = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}_{A \times B}}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, update_{A \times B}^\#, classify_{A \times B}^\# \rangle$ with*

$$\begin{aligned} C_{A \times B}^\# &:= C_A^\# \times C_B^\#, \\ \gamma_{A \times B}(\hat{S}_A, \hat{S}_B) &:= \gamma_A(\hat{S}_A) \cap \gamma_B(\hat{S}_B), \\ \widehat{\mathcal{I}_{A \times B}} &:= \langle \widehat{\mathcal{I}_A}, \widehat{\mathcal{I}_B} \rangle, \\ \langle \hat{S}_A, \hat{S}_B \rangle \sqsubseteq_{A \times B} \langle \hat{T}_A, \hat{T}_B \rangle &:\Leftrightarrow \hat{S}_A \sqsubseteq_A \hat{T}_A \wedge \hat{S}_B \sqsubseteq_B \hat{T}_B, \\ \langle \hat{S}_A, \hat{S}_B \rangle \sqcup_{A \times B} \langle \hat{T}_A, \hat{T}_B \rangle &:= \langle \hat{S}_A \sqcup_A \hat{T}_A, \hat{S}_B \sqcup_B \hat{T}_B \rangle, \\ update_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b) &:= \langle update_A^\#(\hat{S}_A, b), update_B^\#(\hat{S}_B, b) \rangle, \\ classify_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b) &:= classify_A^\#(\hat{S}_A, b) \vee classify_B^\#(\hat{S}_B, b). \end{aligned}$$

► **Theorem 19** (Soundness of Direct Product). *The direct product $A \times B$ of two sound persistence analyses A and B that satisfy (6), (7), (8), and (10) is a sound persistence analysis.*

Proof. We show that $A \times B$ satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- As A and B satisfy (6), we have $\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}_A})$ and $\mathcal{I}_C \subseteq \gamma_B(\widehat{\mathcal{I}_B})$. Thus,

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}_A}) \cap \gamma_B(\widehat{\mathcal{I}_B}) \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\widehat{\mathcal{I}_A}, \widehat{\mathcal{I}_B}) \stackrel{\text{Def. } \widehat{\mathcal{I}_{A \times B}}}{=} \gamma_{A \times B}(\widehat{\mathcal{I}_{A \times B}}),$$

and so $A \times B$ satisfies (6).

- For (7), we have to show that

$$\forall \hat{S}, \hat{T} \in C_{A \times B}^\# : \hat{S} \sqsubseteq_{A \times B} \hat{T} \Rightarrow \gamma_{A \times B}(\hat{S}) \subseteq \gamma_{A \times B}(\hat{T}).$$

Let $\hat{S} = \langle \hat{S}_A, \hat{S}_B \rangle$ and $\hat{T} = \langle \hat{T}_A, \hat{T}_B \rangle$ be arbitrary. Assume that $\hat{S} \sqsubseteq_{A \times B} \hat{T}$, otherwise the implication trivially holds. Then, we have $\hat{S}_A \sqsubseteq_A \hat{T}_A$ and $\hat{S}_B \sqsubseteq_B \hat{T}_B$ by definition of $\sqsubseteq_{A \times B}$. As A and B satisfy (7) this implies $\gamma_A(\hat{S}_A) \subseteq \gamma_A(\hat{T}_A)$ and $\gamma_B(\hat{S}_B) \subseteq \gamma_B(\hat{T}_B)$, and so we have

$$\begin{aligned} \gamma_{A \times B}(\hat{S}) &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_A(\hat{S}_A) \cap \gamma_B(\hat{S}_B) \\ &\subseteq \gamma_A(\hat{T}_A) \cap \gamma_B(\hat{T}_B) \\ &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\hat{T}). \end{aligned}$$

- For (8), we have to show that

$$\begin{aligned} \forall \hat{S} \in C_{A \times B}^\#, b \in \mathcal{B} : \{t.c(b, h)c' \mid t.c \in \gamma_{A \times B}(\hat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_{A \times B}(\text{update}_{A \times B}^\#(\hat{S}, b)). \end{aligned}$$

Let $\widehat{S} = \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$ be arbitrary. Further, let $t.c \in \gamma_{A \times B}(\widehat{S})$ be arbitrary. We will show that $t.c \langle b, h \rangle c' \in \gamma_{A \times B}(\text{update}_{A \times B}^\#(\widehat{S}, b))$, with $h = \text{eff}_C(c, b)$ and $c' = \text{update}_C(c, b)$. By definition of $\gamma_{A \times B}$, $t.c \in \gamma_A(\widehat{S}_A)$ and $t.c \in \gamma_B(\widehat{S}_B)$. Because A and B satisfy (8), we have both $t.c \langle b, h \rangle c' \in \gamma_A(\text{update}_A^\#(\widehat{S}_A, b))$ and $t.c \langle b, h \rangle c' \in \gamma_B(\text{update}_B^\#(\widehat{S}_B, b))$, and thus:

$$\begin{aligned} t.c \langle b, h \rangle c' &\in \gamma_A(\text{update}_A^\#(\widehat{S}_A, b)) \cap \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \\ &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)) \\ &\stackrel{\text{Def. } \text{update}_{A \times B}^\#}{=} \gamma_{A \times B}(\text{update}_{A \times B}^\#(\widehat{S}, b)). \end{aligned}$$

■ For (10), we have to show that

$$\begin{aligned} \forall \widehat{S} \in C_{A \times B}^\#, b \in \mathcal{B} : \text{classify}_{A \times B}^\#(\widehat{S}, b) \Rightarrow \\ \forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_{A \times B}(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \end{aligned} \quad (63)$$

Let $\widehat{S} = \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{A \times B}^\#(\widehat{S}, b)$ holds, otherwise the implication holds trivially.

By the definition of $\text{classify}_{A \times B}^\#$, $\text{classify}_A^\#(\widehat{S}_A, b)$ holds or $\text{classify}_B^\#(\widehat{S}_B, b)$ holds. Assume $\text{classify}_A^\#(\widehat{S}_A, b)$ holds. The case that $\text{classify}_B^\#(\widehat{S}_B, b)$ is analogous.

As A satisfies (10), we have $\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}_A) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b)$. As by definition, $\gamma_{A \times B}(\widehat{S}) \subseteq \gamma_A(\widehat{S}_A)$, we also have: $\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_{A \times B}(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b)$. ◀

► **Theorem 20** (Precision of Direct Product*). *The direct product $A \times B$ of two persistence analyses A and B is at least as precise as A and B , i.e., $A \times B \succeq A$ and $A \times B \succeq B$.*

► **Corollary 21** (Precision of Direct Product*). *The direct product $A \times B$ of two incomparable persistence analyses A and B is more precise than A and B , i.e., $A \times B \succ A$ and $A \times B \succ B$.*

► **Definition 22** (State Reduction). *Let A and B be persistence analyses. A reduction operator for A in the context of B is a function $\text{red} : C_A^\# \times C_B^\# \rightarrow C_A^\#$ that is reductive and that preserves concretizations, i.e., for all $\widehat{S}_A \in C_A^\#, \widehat{S}_B \in C_B^\#$:*

$$\text{red}(\widehat{S}_A, \widehat{S}_B) \sqsubseteq_A \widehat{S}_A, \quad (42)$$

$$\gamma_A(\text{red}(\widehat{S}_A, \widehat{S}_B)) \cap \gamma_B(\widehat{S}_B) = \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B). \quad (43)$$

► **Theorem 23** (State Reduction). *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let red be a reduction operator for A in the context of B . Let the reduced update be defined as follows:*

$$\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) := (\text{red}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)), \text{update}_B^\#(\widehat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}_{A \times B}}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{red-upd}, \text{classify}_{A \times B}^\# \rangle$ is a sound persistence analysis that is at least as precise as $A \times B$, i.e., $A \times B' \succeq A \times B$.

Proof. We know that $A \times B$ is a sound persistence analysis from Theorem 19 that satisfies (6), (7), (8), and (10). The only condition from Theorem 4 that involves the update function is (8). Thus all conditions but (8) are fulfilled by $A \times B'$ as they are fulfilled by $A \times B$.

To show that (8) is satisfied, we argue that $\gamma_{A \times B}(\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) =$

$\gamma_{A \times B}(\text{update}_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b))$ for all $\langle \hat{S}_A, \hat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$:

$$\begin{aligned}
& \gamma_{A \times B}(\text{red-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \\
& \stackrel{\text{Def. red-upd}}{=} \gamma_{A \times B}(\text{red}(\text{update}_A^\#(\hat{S}_A, b), \text{update}_B^\#(\hat{S}_B, b)), \text{update}_B^\#(\hat{S}_B, b)) \\
& \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_A(\text{red}(\text{update}_A^\#(\hat{S}_A, b), \text{update}_B^\#(\hat{S}_B, b))) \cap \gamma_B(\text{update}_B^\#(\hat{S}_B, b)) \\
& \stackrel{(43)}{=} \gamma_A(\text{update}_A^\#(\hat{S}_A, b)) \cap \gamma_B(\text{update}_B^\#(\hat{S}_B, b)) \\
& \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\text{update}_A^\#(\hat{S}_A, b), \text{update}_B^\#(\hat{S}_B, b)) \\
& \stackrel{\text{Def. update}_{A \times B}^\#}{=} \gamma_{A \times B}(\text{update}_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b))
\end{aligned}$$

We can easily show that $A \times B'$ is at least as precise as $A \times B$ by making use of Theorem 7. To this end, we need to define a function $\gamma_{A \times B' \rightarrow A \times B}$ that satisfies conditions (11), (12), (13), (15), and (16). We define $\gamma_{A \times B' \rightarrow A \times B}$ to be the identity function. Conditions (11), (12), (15), and (16) trivially hold as the left and right hand sides of these inequalities are the same. Finally (13) reduces to $\forall \langle \hat{S}_A, \hat{S}_B \rangle \in C_{A \times B}^\#, b \in \mathcal{B} : \text{red-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b) \sqsubseteq_{A \times B} \text{update}_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b)$, which follows from (42):

$$\begin{aligned}
\text{red-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b) &= (\text{red}(\text{update}_A^\#(\hat{S}_A, b), \text{update}_B^\#(\hat{S}_B, b)), \text{update}_B^\#(\hat{S}_B, b)) \\
&\stackrel{(42)}{\sqsubseteq_{A \times B}} (\text{update}_A^\#(\hat{S}_A, b), \text{update}_B^\#(\hat{S}_B, b)) \\
&= (\text{update}_{A \times B}^\#(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \quad \blacktriangleleft
\end{aligned}$$

► **Definition 24** (Cooperative Update). *Let A and B be two persistence analyses. A cooperative update for A in the context of B is a function $\text{coop-upd} : (C_A^\# \times C_B^\#) \times \mathcal{B} \rightarrow C_A^\#$, such that:*

$$\begin{aligned}
& \forall \langle \hat{S}_A, \hat{S}_B \rangle \in C_A^\# \times C_B^\#, b \in \mathcal{B} : \\
& \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\hat{S}_A) \cap \gamma_B(\hat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\
& \qquad \qquad \qquad \subseteq \gamma_A(\text{coop-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b)) \quad (44)
\end{aligned}$$

► **Theorem 25** (Cooperative Update*). *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let coop-upd be a cooperative update function for A in the context of B . Let the reduced update be defined as follows:*

$$\text{red-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b) := (\text{coop-upd}(\langle \hat{S}_A, \hat{S}_B \rangle, b), \text{update}_B^\#(\hat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}_{A \times B}}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{red-upd}, \text{classify}_{A \times B}^\# \rangle$ is a sound persistence analysis.

► **Theorem 26** (Soundness of the State Reduction between C -Must and Block-CS^*). *The function $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is a reduction operator for C -Must in the context of Block-CS .*

► **Theorem 27** (Soundness of the State Reduction between C -Must and C -May). *The operator $\text{reduce}_{C\text{-Must} \times C\text{-May}}$ is a reduction operator for C -Must in the context of C -May.*

Proof. $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is reductive, as $\min \{ \hat{S}_{C\text{-Must}}(b), \dots \} \leq \hat{S}_{C\text{-Must}}(b)$.

It remains to show that for all $\hat{S}_1 \in C_{C\text{-Must}}^\#, \hat{S}_2 \in C_{C\text{-May}}^\#$:

$$\gamma_{C\text{-Must} + C\text{-May}}(\hat{S}_1, \hat{S}_2) = \gamma_{C\text{-Must} + C\text{-May}}(\text{reduce}_{C\text{-Must} \times C\text{-May}}(\hat{S}_1, \hat{S}_2), \hat{S}_2). \quad (64)$$

If we can show

$$\gamma_{C-Must}(\widehat{S}_1) \cap \gamma_{C-May}(\widehat{S}_2) = \gamma_{C-Must}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2)) \cap \gamma_{C-May}(\widehat{S}_2), \quad (65)$$

then (64) can be shown as follows:

$$\begin{aligned} \gamma_{C-Must+C-May}(\widehat{S}_1, \widehat{S}_2) &\stackrel{\text{Def. } \gamma_{C-Must+C-May}}{=} \gamma_{C-Must}(\widehat{S}_1) \cap \gamma_{C-May}(\widehat{S}_2) \\ &\stackrel{(65)}{=} \gamma_{C-Must}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2)) \cap \gamma_{C-May}(\widehat{S}_2) \\ &\stackrel{\text{Def. } \gamma_{C-Must+C-May}}{=} \gamma_{C-Must+C-May}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2) \end{aligned}$$

Let us now show that (65) holds:

As $\text{reduce}_{C-Must \times \text{Block-CS}}$ is reductive and γ_{C-Must} is monotone, we know that

$$\gamma_{C-Must}(\widehat{S}_1) \cap \gamma_{C-May}(\widehat{S}_2) \supseteq \gamma_{C-Must}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2)) \cap \gamma_{C-May}(\widehat{S}_2).$$

To prove that

$$\gamma_{C-Must}(\widehat{S}_1) \cap \gamma_{C-May}(\widehat{S}_2) \subseteq \gamma_{C-Must}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2)) \cap \gamma_{C-May}(\widehat{S}_2),$$

assume for a contradiction that there is a trace $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ in $\gamma_{C-Must}(\widehat{S}_1) \cap \gamma_{C-May}(\widehat{S}_2)$ that is not in $\gamma_{C-Must+C-May}(\text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2)$.

Then, there must be an i , such that $b_i \notin CS_{i+1}$ and $|CS_i| \leq \widehat{S}_1(b_i)$, but not

$$\begin{aligned} |CS_i| &\leq \text{reduce}_{C-Must \times C-May}(\widehat{S}_1, \widehat{S}_2)(b_i) \\ &= \min\{\widehat{S}_1(b_i) | \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\} \mid + 1\} \\ &\leq |\{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\}| + 1. \end{aligned}$$

To reach a contradiction, we will show that $CS_i \setminus \{b_i\} \subseteq \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') \leq \widehat{S}_1(b_i)\}$.

Let c be an arbitrary element of $CS_i \setminus \{b_i\}$ and let j denote the index of the last occurrence of c in the trace s . As $c \in CS_i \setminus \{b_i\}$, j must be greater than i . Thus $j \geq i + 1$, and so $CS_j \subseteq CS_{i+1}$.

As $b_i \notin CS_{i+1}$, we also have $b_i \notin CS_j$. So $|CS_i| \leq \widehat{S}_1(b_i)$ implies $|CS_j| < \widehat{S}_1(b_i)$.

As the trace s is in $\gamma_{C-May}(\widehat{S}_2)$, we have $|CS_j| \geq \widehat{S}_2(c)$.

Thus $\widehat{S}_2(c) \leq |CS_j| < \widehat{S}_1(b_i)$, which shows that $c \in \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\}$. ◀

► **Theorem 28** (Soundness of Must Analysis). *Must is a sound persistence analysis.*

Proof. We show that *Must* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is satisfied, because $\gamma_{Must}(\widehat{\mathcal{I}_{Must}})$ represents all possible sequences.
- Let $\widehat{S} \sqsubseteq_{Must} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{Must}(\widehat{S})$. Because $\widehat{S}(b_i) \leq \widehat{T}(b_i)$ for all i , and \leq is transitive, s is also an element of $\gamma_{Must}(\widehat{T})$, which shows that (7) is satisfied.
- Let $\widehat{S} \in C_{Must}^\#$, $b \in \mathcal{B}$, and $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{Must}(\widehat{S})$ be arbitrary. To show that (8) is satisfied, we have to show that $t = c_0 \langle b_0, h_0 \rangle \dots c_n \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{Must}(\widehat{T})$, with $\widehat{T} = \text{update}_{Must}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{Must}(\hat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.\langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second and third constraint in (51) hold¹², i.e.,

$$(\forall b \in \mathcal{B} : (\forall i, 0 \leq i < n+1 : b_i \neq b) \Rightarrow \hat{T}(b) = \infty) \quad (66)$$

$$\wedge \quad \forall i, 0 \leq i < n+1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \hat{T}(b_i) \quad (67)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n+1\}$.

- Let us consider the constraint (66) first.

Let $b \in \mathcal{B}$ be arbitrary. We distinguish two cases:

1. $\hat{S}(b) \neq \infty$:

Then, as $s \in \gamma_{Must}(\hat{S})$, there must be an $i, 0 \leq i < n$, such that $b_i = b$. As t is an extension of s , b_i also is part of t and thus $(\forall i, 0 \leq i < n+1 : b_i \neq b)$ is false for t as well.

2. $\hat{S}(b) = \infty$:

Then, we further distinguish two cases:

- a. $b_n = b$:

Then, the constraints holds for t , because $(\forall i : b_i \neq b)$ is false.

- b. $b_n \neq b$:

Then, $\hat{T}(b) = \infty$ due to the definition of $\text{update}_{Must}^\#$, where the second case applies.

With $\hat{T}(b) = \infty$ the constraint holds trivially.

- Let us now consider the constraint (67).

Let i be arbitrary. We distinguish two cases based on i 's value:

1. $i = n$:

$CS_n(t) = \{b_n\}$. Due to the first case in the definition of $\text{update}_{Must}^\#$, $\hat{T}(b_n) = 1$, and so $|CS_n(t)| \leq \hat{T}(b_n)$.

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \hat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{Must}(\hat{S})$.

Observe that $CS_i(t) = CS_i(s) \cup \{b_n\}$ for all $i, 0 \leq i < n$.

We distinguish two cases:

- a. $b_i \in CS_{i+1}(s)$:

As $CS_i(t) \supseteq CS_i(s)$, we have $b_i \in CS_{i+1}(t)$.

- b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \hat{S}(b_i)$:

We distinguish two further cases:

- i. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_n\} = CS_{i+1}(s) \cup \{b_i\}$.

- ii. $b_i \neq b_n$:

We distinguish three cases based on which case in $\text{update}_{Must}^\#$ applies:

- A. First case in $\text{update}_{Must}^\#$ applies:

This is impossible as $b_i \neq b_n$.

- B. Second case in $\text{update}_{Must}^\#$ applies:

Thus, $\hat{S}(b_n) \leq \hat{S}(b_i)$ and $\hat{T}(b_i) = \hat{S}(b_i)$.

We distinguish two further cases:

- * $\hat{S}(b_i) = \infty$:

Then, $\hat{T}(b_i) = \infty$ and trivially $|CS_i(t)| \leq \hat{T}(b_i)$.

¹²The constraints below account for the fact that t contains $n+1$ accesses, where n is the number of accesses in s .

* $\widehat{S}(b_i) < \infty$:

As $\widehat{S}(b_n) \leq \widehat{S}(b_i)$, we also have $\widehat{S}(b_n) < \infty$.

As a consequence, due to the second constraint in (51), b_n must occur in s .

Let j be the index of the last occurrence of b_n in s .

If $i < j$ then $CS_i(t) = CS_i(s)$ and so $|CS_i(t)| = |CS_i(s)| \leq \widehat{S}(b_i) = \widehat{T}(b_i)$.

Otherwise, if $j < i$ then $CS_i(t) = CS_i(s) \cup \{b_n\} \subseteq CS_j(s)$.

As $|CS_j(s)| \leq \widehat{S}(b_n)$ and $\widehat{S}(b_n) \leq \widehat{S}(b_i) = \widehat{T}(b_i)$, we have $|CS_i(t)| \leq \widehat{T}(b_i)$.

C. Third or fourth case in $update_{Must}^\#$ applies:

Then $|CS_i(t)| \leq |CS_i(s)| + 1 \leq \widehat{S}(b_i) + 1 \leq \widehat{T}(b_i)$

■ Let $\widehat{S} \in C_{Must}^\#$ and $b \in \mathcal{B}$ be arbitrary.

Assume $classify_{Must}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) < k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{Must}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) is satisfied by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $|CS_i(s)| \leq \widehat{S}(b)$. As $\widehat{S}(b) \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 29** (Soundness of Cooperative Update). *The function $coop-upd_{C-Must \times Must}$ is a cooperative update for $C-Must$ in the context of $Must$.*

Proof. We need to show that $coop-upd_{C-Must \times Must}$ satisfies (44), i.e.

$$\begin{aligned} \forall (\widehat{S}, \widehat{S}_{Must}) \in C_{C-Must}^\# \times C_{Must}^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{C-Must}(\widehat{S}) \cap \gamma_{Must}(\widehat{S}_{Must}) \wedge h = \text{eff}_C^{LRU}(c, b) \wedge c' = \text{update}_C^{LRU}(c, b)\} \\ \subseteq \gamma_{C-Must}(coop-upd_{C-Must \times Must}(\widehat{S}, \widehat{S}_{Must}, b)) \end{aligned} \quad (68)$$

Let $(\widehat{S}, \widehat{S}_{Must}) \in C_{C-Must}^\# \times C_{Must}^\#$ and $b \in \mathcal{B}$ be arbitrary.

Pick an arbitrary $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S}) \cap \gamma_{Must}(\widehat{S}_{Must})$. We have to show that $t = s.c \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C-Must}(\widehat{T})$, with $\widehat{T} = \text{update}_{C-Must}^\#(coop-upd_{C-Must \times Must}(\widehat{S}, \widehat{S}_{Must}, b_n))$ for all $b_n \in \mathcal{B}$.

Because $s \in \gamma_{C-Must}(\widehat{S})$, $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.c \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (31) holds, i.e.:

$$\forall i, 0 \leq i < n+1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \widehat{S}(b_i).$$

Let i be arbitrary. We distinguish two cases based on its value:

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

The second case in the definition of $coop-upd_{C-Must \times Must}$ applies, and so $\widehat{T}(b_n) = 1$.

Thus, $|CS_n(t)| \leq \widehat{T}(b_n)$.

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S})$.

Clearly, $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

So the case where $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \widehat{S}(b_i)$ remains.

Then, the first case in the update may not apply, because $|CS_i(s)| > 0$ and thus $\widehat{S}(b_i) > 0$.

We distinguish two cases:

a. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_n\} = CS_{i+1}(s) \cup \{b_i\}$.

b. $b_i \neq b_n$:

Because $b_i \neq b_n$, the second case in the update may not apply. So only the three final cases in $coop-upd_{C-Must \times Must}$ are possible.

Observe that $CS_i(t) = CS_i(s) \cup \{b_n\}$.

We distinguish two cases:

i. $b_i \in CS_i(s)$:

Then $|CS_i(t)| = |CS_i(s)| \leq \widehat{S}(b_i) \leq \widehat{T}(b_i)$ regardless of which of the three possible final cases in $coop-upd_{C-Must \times Must}$ applies to b_i .

ii. $b_n \notin CS_i(s)$:

Then $|CS_i(t)| = |CS_i(s)| + 1$.

We apply a case distinction based on the three possible final cases in $coop-upd_{C-Must \times Must}$:

A. If the fourth case in $coop-upd_{C-Must \times Must}$ applies to b_i , then

$$|CS_i(t)| = |CS_i(s)| + 1 \leq \widehat{S}(b_i) + 1 = \widehat{T}(b_i).$$

B. If the fifth case in the update applies, then $|CS_i(t)| \leq \infty = \widehat{T}(b_i)$.

C. It remains to show that $|CS_i(t)| \leq \widehat{T}(b_i)$ even if the third case in the update applies, which is where the update profits from the information provided by the must analysis.

If b_n does not occur in s , then by the definition of γ_{Must} , $\widehat{S}_{Must}(b_n) = \infty$, and so $\widehat{T}(b_n) = \widehat{S}(b_n) = \infty > |CS_i(t)|$.

Otherwise, let j be the index of the last occurrence of b_n in s .

As $b_n \notin CS_i(s)$, $j < i$, and $CS_j(s) \supseteq CS_i(s) \cup \{b_j\} = CS_i(s) \cup \{b_n\} = CS_i(t)$.

By the definition of γ_{Must} , $|CS_j(s)| \leq \widehat{S}_{Must}(b_n)$.


Under the assumption that the third case in the update applies, $\widehat{S}_{Must}(b_n) \leq \widehat{S}(b_i)$ and thus $|CS_i(t)| \leq |CS_j(s)| \leq \widehat{S}_{Must}(b_n) \leq \widehat{S}(b_i) = \widehat{T}(b_i)$. \blacktriangleleft

A Static Analysis for the Minimization of Voters in Fault-Tolerant Circuits

Dmitry Burlyaeв

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France


dmitry.burlyaeв@inria.fr

 <https://orcid.org/0000-0001-8685-6923>

Pascal Fradet

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France


pascal.fradet@inria.fr

 <https://orcid.org/0000-0003-4961-9923>

Alain Girault

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

alain.girault@inria.fr

 <https://orcid.org/0000-0001-7500-1655>

Abstract

We present a formal approach to minimize the number of voters in triple-modular redundant (TMR) sequential circuits. Our technique actually works on a single copy of the TMR circuit and considers a large class of fault models of the form “at most one Single-Event Upset (SEU) or Single-Event Transient (SET) every k clock cycles”. Verification-based voter minimization guarantees that the resulting TMR circuit (i) is fault tolerant to the soft-errors defined by the fault model and (ii) is functionally equivalent to the initial TMR circuit. Our approach operates at the logic level and takes into account the input and output interface specifications of the circuit. Its implementation makes use of graph traversal algorithms, fixed-point iterations, and binary decision diagrams (BDD). Experimental results on the

ITC’99 benchmark suite indicate that our method significantly decreases the number of inserted voters, yielding a hardware reduction of up to 55% and a clock frequency increase of up to 35% compared to full TMR. As our experiments show, if the SEU fault-model is replaced with the stricter fault-model of SET, it has a minor impact on the number of removed voters. On the other hand, BDD-based modelling of SET effects represents a more complex task than the modelling of an SEU as a bit-flip. We propose solutions for this task and explain the nature of encountered problems. We address scalability issues arising from formal verification with approximations and assess their efficiency and precision.

2012 ACM Subject Classification Hardware → Model checking, Hardware → Redundancy

Keywords and Phrases Digital Circuits, Fault-tolerance, Optimization, Static Analysis, Triple Modular Redundancy.

Digital Object Identifier 10.4230/LITES-v005-i001-a004

Received 2017-01-11 **Accepted** 2018-06-21 **Published** 2018-10-15

1 Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors is an important research topic. As technology shrinks, the risk of system failures due to soft errors increases, which is especially dangerous in safety-critical industries (*e.g.*, space, transport, nuclear, *etc.*). Natural radiation, such as neutrons of cosmic rays and alpha particles of packing or solder materials, is a common source of soft errors [20, 34, 42, 47]. There are two main types of soft errors: Single-Event Upsets (SEUs) (*i.e.*, bit-flips in Flip-Flops (FFs)) and Single-Event Transients (SETs) (*i.e.* glitches



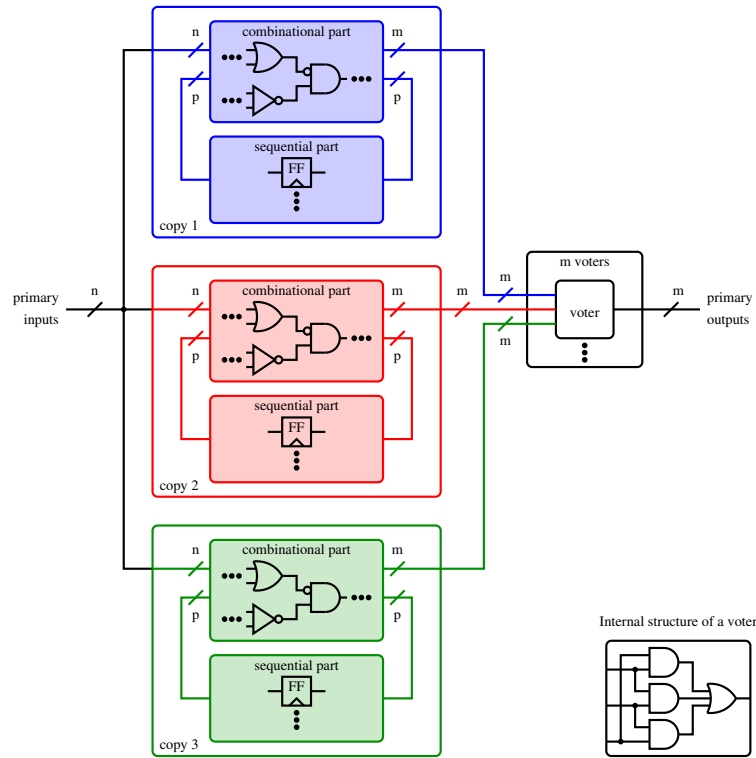
© Dmitry Burlyaeв, Pascal Fradet, and Alain Girault;
licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)

Leibniz Transactions on Embedded Systems, Vol. 5, Issue 1, Article No. 4, pp. 04:1–04:26



Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** TMR scheme proposed by von Neumann, with n primary inputs, m primary outputs, and a voter after each primary output.

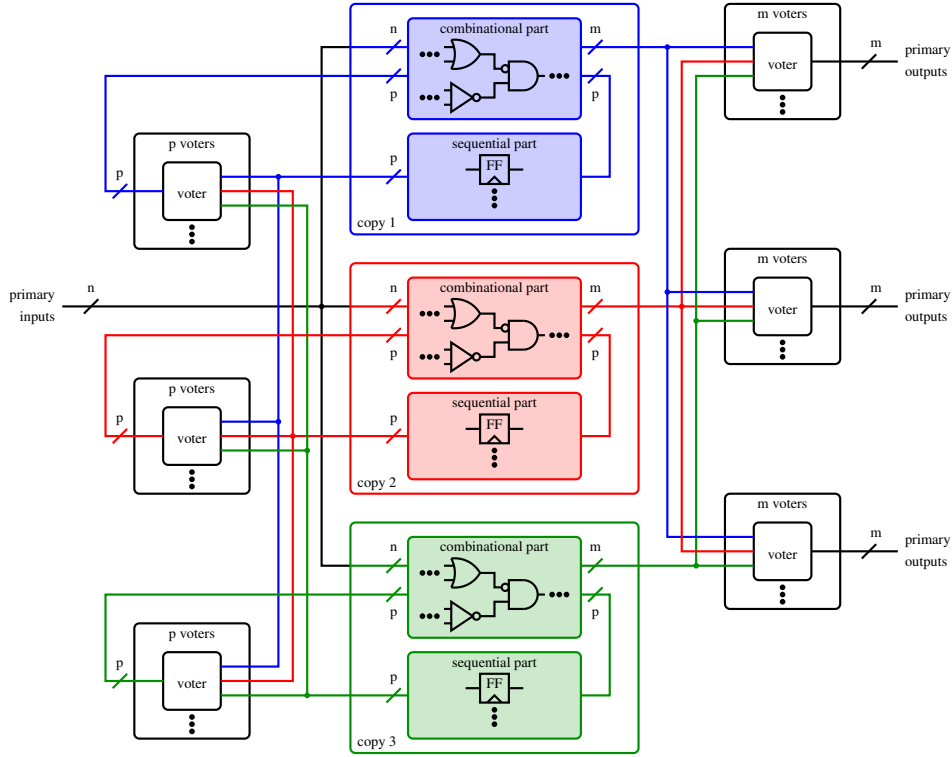
propagating in the combinational circuit). Since an SET may lead to several bit-flips, SETs are more general than SEUs.

Triple-Modular Redundancy (TMR) proposed by von Neumann [45] remains the most popular fault tolerance technique in Field-Programmable Gate Arrays (FPGAs) to mask both types of soft-errors. In its original form, TMR relies on three redundant copies of an original system receiving the same inputs, as shown in Figure 1. A majority voter is inserted after each of the triplicated primary outputs. If at most one redundant module returns incorrect values, the voter will return the correct result, therefore masking one possible error. An implementation example of the majority voter is also depicted in Figure 1. The voter always returns the majority bit among its three inputs provided that a fault does not occur in its own logic.

Manual introduction of TMR [24] into a circuit design is often a tedious and error-prone process. Hence, several CAD tools automatically implement TMR for fault tolerant FPGA designs [6, 19, 37, 44, 46].

In a triplicated sequential circuit, adding voters at the primary outputs is not sufficient in general. Indeed, an error may remain in a memory cell long enough until another error corrupts a different redundant copy of the circuit. In that case, the final vote may produce an incorrect output. Moreover, single voters cannot mask errors occurring within the voting logic itself. A solution to these problems is to insert a triplicated voter after each memory cell (as depicted in Figure 2). This is sufficient to mask any SET in the combinational circuit (even within voters) and to prevent errors from remaining in cells ¹. However, full TMR greatly increases both the

¹ Of course, two upsets occurring at the same cycle in two different copies may not be masked; triplication does not provide enough redundancy for multiple upsets.



■ **Figure 2** Full TMR with triplicated voters after the p internal FFs and the m primary outputs.

hardware overhead and the critical path, which directly influences the circuit performance. Thus, the overall TMR throughput is degraded whereas it should be the main advantage of TMR over time-redundant fault-tolerance techniques.

From the functional point of view, introducing a voter per cell is excessive in most cases. Intuitively, this is because some voters are useless, either because faults at this stage will be captured by another voter “later” in the circuit, or because some faults are naturally masked by the logic. But, to the best of our knowledge, there is no tool dedicated to voter minimization in TMR that guarantees fault-tolerance according to a user-defined fault model. The main existing research trends in TMR have been providing probabilistic solutions and not absolute ones (see Section 8).

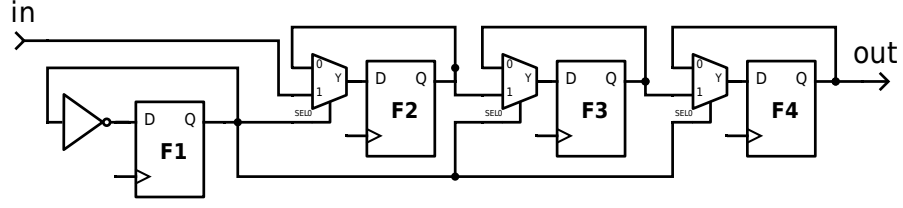
In this paper, we propose an *automatic* and *optimized* transformation process for TMR on digital circuits. Our transformation inserts as few voters as possible, while guaranteeing to mask all errors of the considered fault-model.

We consider circuits described at the gate level (*i.e.*, netlists of AND, OR, NOT gates plus FFs – also called memory cells). This level has two main advantages:

- gate level netlists can be described by an elementary language, which simplifies correctness proofs;
- it is easier to prevent synthesis tools from optimizing (undoing) our transformation at this late design stage.

Since the main contributors to Soft-Error Rate (SER) at frequencies below 1 GHz are the FFs [27], we focus first on errors caused by SEUs (*i.e.*, bit-flips in FFs). We consider fault models of the form “at most one bit-flip within K cycles” denoted by $SEU(1, K)$.

However, SETs in high-speed Integrated Circuits (ICs) have become a growing concern [8,25,35].



■ **Figure 3** Use-case of the semantic analysis: voters are needed only at F1 and the primary output out.

An SET is a voltage pulse (glitch) caused by a particle. It may propagate through the combinational logic provided that it is not logically masked by the circuit functionality. As a result, the outputs of the combinational circuit might be glitched and be incorrectly latched by memory cells. Due to the non-deterministic nature of the propagated glitch, it can be latched by none, some, or all memory cells it reaches. Thus, since an SET may lead to several corrupted memory cells (bit-flips), SETs subsume SEUs. In response, we expand our approach to fault-models of the form “at most one SET within K clock cycles” denoted by $SET(1, K)$.

The proposed voter-minimization methodology is based on a static analysis that checks whether an error in a single copy of the TMR circuit may remain after K cycles. If not, protecting the primary outputs with voters is sufficient to mask the error. If, for instance, the circuit is a pipeline without feedback loops, then any bit-flip will propagate to the outputs and will thus disappear before K cycles, where K is the length of the longest path. But if the state of the circuit is still erroneous after K cycles (in the form of an incorrect value stored in one of its memory cells), then there is a potential error accumulation since, according to the $SEU/SET(1, K)$ models, another soft-error may occur in another copy of the circuit. It may lead to two incorrect redundant modules of the TMR circuit and the loss of its fault-tolerance properties. In this case, additional voters are needed to prevent an error accumulation and mask all errors circulating inside one redundant module before the next soft-error may occur.

Our static analysis consists of four steps. The first step, described in Section 2, is purely syntactic and finds all loops in the circuit. Error accumulation can be prevented by keeping enough voters to cut all loops.

In many cases, a digital circuit resets (or overwrites) some memory cells, which may mask errors. Detecting such cases allows further useless voters to be removed. This second step is performed by a semantic analysis (Section 3) taking into account the logic of the circuit.

Figure 3 shows a simple example where the previous syntactic analysis is not able to suppress any voter because each FF is in a self-loop. Our semantic analysis can catch the behavior of $F1$ that changes its value every cycle. Since $F1$ controls the multiplexers before $F2$, $F3$, and $F4$, all these FFs will be rewritten each other cycle. Consequently, even if a fault occurs in one of these three FFs, it will be eventually re-written with new correct data coming from the primary input in . The semantic analysis indicates that it is sufficient to protect $F1$ and the primary output with majority voters.

Circuits are also often supposed to be used in a specific context. For instance, a circuit specification may assume that a **start** signal occurs every x cycles and outputs are only read y cycles after each **start**. When such assumptions exist, taking them into account makes the semantic analysis more effective. Section 4 and Section 5 explain how to integrate such input and output specifications respectively.

Our analysis has been implemented based on graph algorithms and fixed point iterations using Binary Decision Diagrams (BDDs). We have tested several safe approximations and trade-offs between cost and precision. The implementation and experiments are presented in Section 7.

Related work on TMR and voter insertion strategies are reviewed in Section 8. We summarize our contributions and sketch a few extensions in Section 9.

This article extends and revises the work presented in [10]. Section 6, presenting the extension of the approach to SET, is new. Sections 3 and 7 present and assess respectively a new abstract domain; explanations and examples have been added throughout.

2 Syntactic Analysis

We consider a triplicated circuit with voters but we actually work on a *single copy* of this circuit that abstracts the triplicated version. The effect of insertion or removal of voters can be represented and analyzed on such a single copy of the TMR circuit. We model a sequential circuit C as a directed graph G_C where each vertex represents a FF (memory cell or latch) and an edge $x \rightarrow y$ exists whenever there is at least one combinational path between the two FFs x and y in C . An error in a cell x may propagate, in the next clock cycle, to all cells connected to x by an edge in this graph. Note that this is an over-approximation since the error may actually be masked by some logical operation.

Under the fault model $SEU(1, K)$, error accumulation is the situation where an error remains in the circuit K clock cycles after the SEU that caused it. Any circuit C without feedback loop will return, after an SEU, to a correct state before K clock cycles, provided that K is larger than the maximal length of the paths in G_C . In environments with high levels of ionizing radiations (*e.g.*, space, particle accelerators), K is bigger than 10^{10} [5]. For comparison, Soft-Error Rate (SER) can be as small as 10^{-10} bit-upset/day for Virtex FPGAs in terrestrial conditions [7]. So, even if our approach can deal with any K , we can safely assume that K is larger than the max length of all paths in G_C . It follows that error accumulation can only be caused by cycles in G_C , which must therefore be cut by removing vertices. Removing a vertex in G_C amounts to protecting the corresponding memory cells with a voter in the triplicated circuit.

The best solution to cut all cycles in G_C is to find the Minimum Vertex Feedback Set (MVFS), *i.e.*, the smallest set of vertices whose removal leaves G_C without cycles. This standard graph problem is NP-hard [32]. While there exist good polynomial time approximations [22], the exact algorithm was efficient enough to be used in all our experiments with relatively small circuits (less than 200 FFs).

Having a voter after each cell belonging to the MVFS prevents error accumulation. This simple graph-based analysis is very effective with some classes of circuits. In particular, it is sufficient to remove all internal voters in pipelined architectures such as logarithm units and floating-point multipliers (see Table 1).

However, this approach is not effective for many circuits due to the extensive use of loops in circuit synthesis from Mealy machine representation. In such circuits, most cells are in self-loops (*e.g.*, D-type flip-flops with Enable input). This entails many voters if the syntactic analysis is used alone. However, if the circuit functionality is taken into account, we can discover that such memory cells may not lead to erroneous outputs. Detecting such cases requires to analyze the logic (semantics) of the circuit. We address this issue in the following section.

3 Semantic Analysis

The semantic analysis first computes the Reachable State Set (RSS) of the circuit with a voter inserted after each memory cell in the MVFS. Then, for each cell $m \in \text{MVFS}$, it checks whether its voter is necessary: (i) First, the voter is removed and all possible errors (modeled by the chosen fault-model in each state of RSS) are considered; (ii) If such an error leads to error accumulation, then the voter is needed and kept.

■ **Table 1** Voter Minimization, Syntactic Analysis Step.

	Circuit	FFs	Syntactic
Data Flow I.	Pipelined FP multiplier 8x8 [21]	121	0
	Pipelined logarithm unit [21]	41	0
	Shift/Add multiplier 8x8 [33]	28	28
	ITC'99 [16](subset)		
Control Flow Intensive	b01 Flows comparator	5	3
	b02 BCD recognizer	4	3
	b03 Resource arbiter	30	29
	b06 Interrupt handler	9	3
	b08 Inclusions detector	21	21
	b09 Serial converter	28	21

3.1 The precise logic domain D_1

Correct and erroneous values are represented by the four-value logic domain D_1 :

$$D_1 = \{0, 1, \bar{0}, \bar{1}\}$$

where $\bar{0}$ and $\bar{1}$ represent erroneous 0 and 1, respectively. The truth tables of standard operations in this four-value logic are given in Table 2. The operators AND and OR gates can mask errors:

$$\bar{x} \vee 1 = 1 \quad \bar{x} \wedge 0 = 0 \quad \bar{0} \wedge \bar{1} = 0 \quad \bar{1} \vee \bar{0} = 1$$

Since we work on circuits that abstracts their TMR version, a $\bar{0}$ (resp. $\bar{1}$) in a cell means that one of its three copies may have the incorrect value 1 (resp. 0). A 0 (resp. 1) means that all three memory cells have the same correct value 0 (resp. 1). This interpretation assumes that any SET affects only one copy of the TMR circuit.

The **err** function models bit-flips and represents a bit corruption in one of three copies in the TMR circuit:

$$\mathbf{err}(0) = \bar{1} \quad \mathbf{err}(1) = \bar{0}$$

The **vot** function models the effect of a voter on the TMR version of the circuit and corrects an error:

$$\mathbf{vot}(\bar{1}) = 0 \quad \mathbf{vot}(\bar{0}) = 1$$

It corresponds, in the TMR version that it abstracts, to the majority voter depicted in Figure 2. Finally, for any $x \in \{0, 1\}$, $\mathbf{vot}(\mathbf{err}(x)) = x$.

3.2 Semantic analysis with D_1

A sequential synchronous circuit with M memory cells and I primary inputs is formalized as a discrete-time transition system with the transition relation $\delta : \{0, 1\}^M \times \{0, 1\}^I \mapsto \{0, 1\}^M$. We abuse the notation and use M (resp. I) to denote both the number and the set of memory cells (resp. inputs) of the circuit. The state of a circuit is the values of its cells and the initial state s_0

■ **Table 2** Operators for the 4-value logic domain D_1 .

OR	0	1	$\bar{1}$	$\bar{0}$
0	0	1	$\bar{1}$	$\bar{0}$
1	1	1	1	1
$\bar{1}$	$\bar{1}$	1	$\bar{1}$	1
$\bar{0}$	$\bar{0}$	1	1	$\bar{0}$

AND	0	1	$\bar{1}$	$\bar{0}$
0	0	0	0	0
1	0	1	$\bar{1}$	$\bar{0}$
$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0
$\bar{0}$	0	$\bar{0}$	0	$\bar{0}$

	NOT	err	vot
0	1	$\bar{1}$	0
1	0	$\bar{0}$	1
$\bar{1}$	$\bar{0}$	0	0
$\bar{0}$	$\bar{1}$	1	1

is obtained after the circuit reset. $\Delta(S)$ denotes the function returning the set of states obtained from the set S after one clock cycle. Formally

$$\Delta(S) = \{s' \mid \exists i. \exists s \in S. \delta(s, i) = s'\}$$

Δ applies the transition function δ to all states of its argument set and all possible inputs. The set of reachable states RSS is defined by the following iteration:

$$\begin{aligned} S_0 &= \{s_0\} \\ S_{i+1} &= S_i \cup \Delta(S_i) \end{aligned} \tag{1}$$

Starting from the initial state, we compute the set of reachable states by accumulating states obtained by applying Δ iteratively. The set of possible states being finite, the iteration reaches a fixed point equal to the RSS and denoted² by $\{s_0\}_\Delta^*$.

The second phase is to check whether the suppression of voters may lead to an error accumulation under the chosen fault-model. Let δ_V be the transition relation of a circuit equipped with a voter after each cell in a given set V , and let Δ_V be its extension to sets. δ_V is defined as:

$$\begin{aligned} \delta_V((m_1, \dots, m_M), i) &= \delta((m'_1, \dots, m'_M), i) \\ \text{where } \forall 1 \leq j \leq M, m'_j &= \begin{cases} \mathbf{vot}(m_j) & \text{if } m_j \in V \\ m_j & \text{otherwise} \end{cases} \end{aligned}$$

This checking process is described by Algorithm 1.

We start with the circuit equipped with a voter after each cell in the MVFS (line 1). For each such cell m , we check whether its voter suppression entails error accumulation. Bit-flips are introduced in all possible cells and states of RSS according to the fault-model (line 5):

$$\bigcup_{m_i \in M} \text{RSS}[m_i \leftarrow \mathbf{err}(m_i)]$$

The transition function corresponding to the circuit with the current set of voters (V) is applied K times (Δ_V^K), where K is the number of clock cycles in the fault model ($SEU(1, K)$).

² We will use this notation with other initial states and transition functions.

Algorithm 1 Semantic Analysis – Main Loop.

```

    Input :  $MVFS$ ; // The minimum vertex feedback set.
            $\Delta$ ;    // The circuit transition function.
            $s_0$ ;    // The initial state.
    Output :  $V$ ;    // The subset of vertices (i.e., memory cells) after which a voter is needed.
1:  $V := MVFS$ ;
2:  $RSS := \{s_0\}_{\Delta}^*$ ;
3: forall  $m \in MVFS$ 
4:    $V := V \setminus \{m\}$ ;
5:    $S := \Delta_V^K(\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)])$ ;
6:   if  $ErrAcc(S)$  then
7:      $V := V \cup \{m\}$ ;
8: return  $V$ 

```

The resulting set of states shows error accumulation if there exists an erroneous cell in at least one state of this set, which we capture with the predicate $ErrAcc$ in line 6. $ErrAcc$ is defined as:

$$ErrAcc(S) \Leftrightarrow \exists s \in S. \exists m \in s. m = \bar{0} \vee m = \bar{1}$$

If the set S does not show error accumulation, the voter is useless and can indeed be suppressed. Otherwise the voter is re-introduced (line 7).

In practice, Δ is applied a small number of times dictated by the circuit functionality and available analysis time. It is always safe to stop the iterative computation before reaching K ; the only drawback would be to infer an error accumulation when there is none. The number of Δ applications can be also adjusted to the available analysis time. In our experiments, the analysis time limit was set to 20 minutes and K to 50. Furthermore, the iteration is stopped:

- if the current set of states is errorless, then there cannot be error accumulation (no error can reappear);
- or, if the erroneous current set is the same as the previous one, a fixed point is reached and there is an error accumulation.

The order in which the cells in the MVFS are analyzed (line 2 in Algorithm 1) may influence the number of removed voters. It follows that the result of removing voters one-by-one is not unique, it depends on the order the voters are chosen. We use the following heuristic to chose the ordering of voter selection: starting from the MVFS of memory cells with voters, we first sort it according to the number of successive memory cells that each cell has in the netlist (the number of successors in G_C). Then, we consider primarily the removal of voters that lead to the corruption of the smallest number of cells in the next clock cycle. The voters whose removal may lead to a large number of corrupted cells are considered last. We found out that following this ordering, we are able to suppress more voters than with a random ordering or with the ordering relying on the number of preceding memory cells in the netlist.

3.3 More Abstract Logic Domains

The aforementioned method is precise but costly since it considers all possible inputs. In general, keeping track of the relations between indeterminate inputs is not very useful. Fortunately, our technique can be used as it is with other, more abstract, logic domains. There are several domains that retain enough precision and allow larger circuits to be analyzed.

■ **Table 3** Operators for the 4-value logic domain D_2 .

OR	0	1	U	\bar{U}	AND	0	1	U	\bar{U}
0	0	1	U	\bar{U}	0	0	0	0	0
1	1	1	1	1	1	0	1	U	\bar{U}
U	U	1	U	\bar{U}	U	0	U	U	\bar{U}
\bar{U}	\bar{U}	1	\bar{U}	\bar{U}	\bar{U}	0	\bar{U}	\bar{U}	\bar{U}

x	NOT	err(x)	vot(x)
0	1	\bar{U}	0
1	0	\bar{U}	1
U	U	\bar{U}	U
\bar{U}	\bar{U}	\bar{U}	U

The 4-value logic domain D_2 decreases the state space explosion that occurs with D_1 :

$$D_2 = \{0, 1, U, \bar{U}\}$$

The abstract value U represents a correct value (either 0 or 1) and \bar{U} represents any (possibly erroneous) value (*i.e.*, 0, 1, $\bar{0}$, or $\bar{1}$). A vector of n inputs is represented as a unique vector (U, \dots, U) with D_2 whereas 2^n vectors had to be considered with D_1 . The truth tables of standard operations in D_2 are given in Table 3.

In contrast with D_1 , a gate with two erroneous values cannot produce a correct one. Logical masking of errors can only occur with two operations: $0 \wedge \bar{U}$ and $1 \vee \bar{U}$. This is sufficient to take into account the masking performed by explicit signals (*e.g.*, resets).

Typical examples where the semantic analysis with D_2 is more effective are circuits that use D-type FFs with an **enable** input driven by a Finite State Machine (FSM) encoded in the circuit. The syntactic approach would keep a voter for each such cell (they are in self-loops). The semantic analysis can detect that such cells are regularly overwritten by fresh inputs. For example, the resource arbiter *b03* in Section 7 is such a circuit. After initialization, its finite state machine forces 12 cells (*fu1-fu4*, *ru1-ru4*, *grant_o[3:0]*) to be overwritten with fresh values every other cycle. The semantic analysis (using D_1 or D_2) is able to show that those cells, although in self loops, do not need to be protected by voters.

Another approximate logic domain is the 16-values logic domain D_3 , where a memory cell is encoded as a subset of its four possible values. It is defined as the powerset of D_1 :

$$D_3 = \mathcal{P}(\{0, 1, \bar{0}, \bar{1}\})$$

A value A in D_3 is the set of all possible values that its memory cell can take at this stage of the analysis. For example, a fully determinate value is represented by a singleton (*e.g.*, $\{0\}$ for a correct 0 or $\{\bar{0}\}$ for a bit-flipped 1), an unknown but uncorrupted value by the set $\{0, 1\}$, and a completely unknown value by the set $\{0, 1, \bar{0}, \bar{1}\}$.

The operators of D_3 are the power set extensions of the operators of D_1 .

$$\begin{aligned}
A \wedge_3 B &= \{x \mid x = a \wedge_1 b, a \in A, b \in B\} \\
A \vee_3 B &= \{x \mid x = a \vee_1 b, a \in A, b \in B\} \\
\neg_3 A &= \{x \mid x = \neg_1 a, a \in A\} \\
err_3(A) &= \{x \mid x = err_1(a), a \in A\} \\
vot_3(A) &= \{x \mid x = vot_1(a), a \in A\}
\end{aligned}$$

where \wedge_1 , \vee_1 , \neg_1 , err_1 , and vot_1 denote the *and*, *or*, *not*, *err*, and *vot* operators of D_1 as defined in Table 2.

This domain is a trade-off in terms of precision between D_1 and D_2 . The main advantage of D_3 over D_1 is the prevention of state explosion, since a vector of n unknown and uncorrupted inputs is represented as a unique vector $(\{0, 1\}, \dots, \{0, 1\})$. Contrary to D_2 , D_3 remains able to represent logical masking such as $\{\bar{0}\} \wedge_3 \{0, \bar{1}\} = \{0\}$ or $\{\bar{1}\} \vee_3 \{1, \bar{0}\} = \{1\}$. D_3 can be seen as retaining precise information about the possible values and corruptions but ignoring the relationships between different inputs.

3.4 Summary

We have presented a semantic analysis to minimize the number of voters in TMR circuits. Using several logic domains, we can represent the internal circuit state with various levels of precision. The functional behavior of a circuit under random inputs is taken into account to analyse how an error may propagate through it. Algorithm 1 and domain encodings have been implemented in Ocaml using the BDD library CUDD [43] and MLCuddIDL [30]. Transition systems and sets of states are expressed as BDD formulae [15]. Section 7 provides more details about the implementation and experimental results. In the two next sections, we improve the analysis by specifying and considering assumptions on inputs and outputs. This permits us to better express real-life circuit behavior and to improve the precision of the analysis.

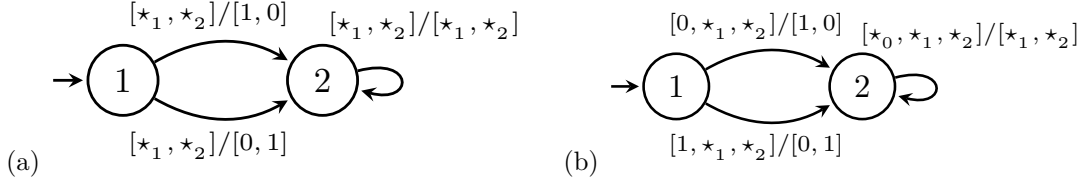
4 Inputs Specification

Circuits are often designed to be used in a specific context where some input signals must occur at definite timings. Taking into account assumptions about the context may make the semantical analysis much more precise, in particular, when the logical masking of corrupted cells depends on specific inputs (*e.g.*, a **start** control signal). Our approach is to translate these specifications into an interface circuit feeding the original circuit with the specified inputs. The analysis of the previous section can be applied to the resulting combined circuit. As a consequence, error accumulation is checked with the method described in Section 3.2, but under the constraints specified by the interface. The only small adjustment needed in Algorithm 1 is to make sure that errors are introduced only in the cells of the original circuit and not in the cells of the interface circuit. We use ω -regular expressions to specify circuit interfaces. An ω -regular expression specifies constraints using vectors of $\{0, 1, \star\}$, which replace primary inputs by 0, 1, or leave them unchanged (\star being the wild card). Consider, for instance, a circuit with two primary inputs $[i_1, i_2]$, then the expression $([1, 0] + [0, 1]).[\star, \star]^\omega$ specifies that the circuit first reads either $i_1 = 0$ and $i_2 = 1$, or $i_1 = 1$ and $i_2 = 0$, and then proceeds with no further constraints.

In general, specifications need non-determinism to describe a partially specified or a non-deterministic context. Hence, the aforementioned ω -regular expression can also be seen as a Non-deterministic Büchi Automaton (NBA) that reads inputs and replaces them by 0, 1, or leaves them unchanged (\star). Such a translation to NBA can be performed in linear time.

For instance, the expression $([1, 0] + [0, 1]).[\star, \star]^\omega$ can be represented as the two-state NBA of Figure 4 (a): in the first state, it reads inputs and returns either the outputs $[1, 0]$ or $[0, 1]$ (regardless of the inputs). Then, the automaton goes (and stays) in the second state where inputs are read and produced as outputs. The indices in \star_1 and \star_2 allow to identify the inputs according to their position.

To generate a circuit from an ω -regular expression, we first convert the corresponding NBA into a deterministic automaton as follows. Each nondeterministic edge is made deterministic using new inputs (sometimes referred to as oracles). If a vertex has n nondeterministic outgoing edges, adding



■ **Figure 4** Input interface as a NBA (a) and its deterministic version (b).

$\log_2(n)$ new inputs is sufficient. For example, the specification $([1, 0] + [0, 1]).[\star, \star]^\omega$ can be made deterministic by adding a single additional input i . The automaton (see Figure 4 (b)) now reads three inputs: if i is 0 (resp. 1) it produces $[1, 0]$ (resp. $[0, 1]$). The resulting deterministic automaton is then translated into an interface circuit using standard logic synthesis techniques [17, p.118]. If the original circuit has I inputs, the resulting interface circuit will have $I + a$ (a new inputs to make it deterministic) inputs and I outputs. It is then plugged by connecting its outputs to the inputs of the circuit to be analyzed.

A typical example where an input specification is useful is the circuit *b08* of Section 7. Such a circuit has a **start** input signal and 8-bit data input. Its input interface specification can be expressed as the following ω -regular expression:

$$([1, \star, \star, \star, \star, \star, \star, \star, \star]. [0, \star, \star, \star, \star, \star, \star, \star, \star])^{17} \omega \quad (2)$$

A **start** signal is first raised and the input data is read. For the next 17 cycles, data is processed and **start** is kept to 0. This process is repeated over and over. Since **start** is raised every 18 clock cycles, the internal data registers are rewritten periodically with new data, as they can keep erroneous data only until the next **start** signal. The circuit also has an internal FSM which can be corrupted but the periodic **start** ensures that it returns to its initial state every 18 cycles. Consequently, error accumulation is impossible for any $K > 18$, and no voters (except implicit voters at primary outputs) need to be inserted.

5 Outputs Specification

Consider another example, similar to the previous one, with 2 inputs, 1 output, and where some waiting can occur before raising the **start** signal. Formally, the input interface would be:

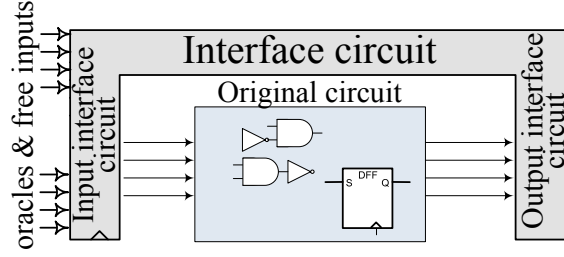
$$([0, \star]^* . [1, \star] . [0, \star])^{17} \omega \quad (3)$$

This interface does not guarantee that **start** will be raised before K clock cycles. Since the analysis must consider the case where **start** is not raised, it may detect error accumulation even though **start** would ensure logical masking. However, if it is known that the primary outputs are not read before some useful computation triggered by the **start** signal completes, a better analysis can be performed.

We specify the output interface by adding to each vector of the input interface a vector of $\{0, 1\}$ indicating whether the corresponding outputs are read (1) or not read (0). For instance, the output interface of the previous example, where the single bit output is read only after **start** is raised, can be specified as

$$(([0, \star] : [0])^* . ([1, \star] : [0]) . ([0, \star] : [1]))^{17} \omega \quad (4)$$

It states that the output is not read ($[0]$) until the **start** signal is raised. Then, the output is read ($[1]$) during 17 cycles.



■ **Figure 5** Original circuit with the surrounding interface circuit.

The extended ω -regular expression is translated into an NBA as in Section 4, then made deterministic, and finally translated into a sequential circuit. The corresponding interface circuit will additionally produce 0 or 1 signals to filter the useless and needed outputs respectively. Each such signal is connected using an AND gate to the corresponding primary output of the original circuit. The final configuration with the surrounding interface circuit is shown in Figure 5.

The property to check must now be refined to allow error accumulation as long as no error propagates to the filtered primary outputs. Recall that when an error occurs, it is allowed to propagate to outputs (or final voters) within the next K clock cycles since no additional soft-error can occur during that time. If there is an error accumulation, the analysis must further ensure that no error can propagate to outputs after the K cycles *i.e.*, when additional errors occur which could not be masked by final voters.

The refined property check discussed in the previous paragraph is performed by lines 6-15 of Algorithm 2. If an error accumulation is detected in the reached state set S , K cycles after a fault occurrence (line 6), then we calculate all states $S_{\Delta_V}^*$ that can be reached after these K cycles (line 7). Then, we iteratively simulate the occurrences of additional errors (line 9-12) separated by at least K steps. E_0 (line 7) represents the circuit reachable state space with only one fault. E_i represents the reachable state space after at most $i + 1$ errors separated from one another by at least K clock cycles. The global fixpoint E_i (line 13) represents the set of all possible states that can be reached after all possible sequence of errors allowed by the fault model. It can now be checked that none of these states leads to the propagation of an error to the (filtered) primary outputs (line 13).

Since this computation is done assuming that voters operate correctly, we must ensure that no error accumulate in a cell followed by a voter. Indeed, in that case, if a similar error occurs in a second copy of the circuit, the voter would fail to mask it. The function *ErrProp* (line 13) detects if there is a reachable state where a memory cell with a voter or a primary output is corrupted and prevents the voter under consideration (m) to be removed. We assume that each primary output is represented by a new memory cell. Let *out*, *vot* and *cor* be predicates denoting whether a cell represents an output, a cell protected by a voter or is corrupted respectively, then *ErrProp* is defined as:

$$ErrProp(E_i) \Leftrightarrow \exists s \in E_i. \exists m \in s. (out(m) \vee vot(m)) \wedge (cor(m))$$

These criteria are safe but sometimes too strict. Consider, for instance, a circuit with a sequence of two enabled flip-flops (*i.e.*, with self loops) that produce significant output only two cycles after the enable signal is set. Both cells may be protected by voters to break self loops and prevent error accumulation. However, no voter is necessary since error accumulation can occur only when no significant output is produced. Indeed, when the enable signal is set, new input and intermediate results will overwrite the (possibly corrupted) cells and a correct output will be produced. If we first try to remove the first voter, our algorithm will detect that an error can

Algorithm 2 Semantic Analysis with Output Specification.

```

  Input :  $MVFS$ ; // The minimum vertex feedback set.
           $\Delta$ ; // The circuit transition function.
           $s_0$ ; // The initial state.
  Output :  $V$ ; // The subset of vertices (i.e., memory cells) after which a voter is needed.
1:  $V := MVFS$ ;
2:  $RSS := \{s_0\}_{\Delta}^*$ ;
3: forall  $m \in MVFS$ 
4:    $V := V \setminus \{m\}$ ;
5:    $S := \Delta_V^K(\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)])$ ;
6:   if  $ErrAcc(S)$  then
7:      $E_0 := \{S\}_{\Delta_V}^*$ ;
8:      $i := 0$ ;
9:     repeat
10:       $i++$ ;
11:       $E_i := E_{i-1} \cup (\Delta^K(\bigcup_{m_i \in M} E_{i-1}[m_i \leftarrow \mathbf{err}(m_i)]))_{\Delta_V}^*$ ;
12:    until  $E_i = E_{i-1}$ ;
13:    if  $ErrProp(E_i)$  then
14:       $V := V \cup \{m\}$ ;
15: return  $V$ ;

```

remain in the first cell after K steps. That cell will in turn corrupt the second one still protected by a voter. Hence, the condition $ErrProp$ will prevent removing the first voter whereas starting with the second or removing both voters would have been possible. Therefore, a useful refinement of Algorithm 2 is, whenever $ErrProp$ is true only because of error accumulation before some voters (and no error propagates to the output), to iterate and check whether all these voters can be removed.

Output interfaces are especially useful for circuits whose outputs are not read before some input signal is raised and some computation is completed. For instance, the shift/add multiplier (see Section 7) waits for a **start** signal. During that time, errors may accumulate in internal registers and propagate to the outputs, which are not read. When **start** occurs, fresh input data is read and written to internal registers (which are thus reset). The outputs are read only after the multiplication is completed and a **done** signal is raised.

Note that output interfaces allow to model Transient Error Tolerance (TET) where all errors at primary outputs are not necessarily critical. For instance, if erroneous outputs are considered non-critical within a specified number of cycles, output interfaces can express it and allow further optimizations. In this case, the optimized TMR configuration is tuned to particular system requirements. Such quality guided optimizations are investigated on MPEG decoding in [26, 36] to select gates whose hardening maximize fault-tolerance.

6 Extension to Single-Event Transients

In the previous sections, we considered single event upsets and the corresponding fault-models $SEU(1, K)$, corresponding to “at most one bit-flip every K cycles”. Hereafter, we extend our approach to single event transients, in particular, the fault model $SET(1, K)$ which can be read as “at most one SET within K clock cycles”.

An SET occurs when a high-energy particle strikes a combinational logic element [25]. Such a particle causes a transient voltage disturbance, which can propagate on wires and possibly be latched by several memory cells. Due to the non-deterministic nature of the propagated glitch, it can be latched by none, some, or all memory cells it reaches. Consequently, an SET can potentially lead to several corrupted memory cells (*i.e.*, several SEUs). In this section, we present the extension of our previous analysis to SET.

6.1 Precise modeling of SETs

As opposed to an SEU, the effect of an SET depends on the logical propagation (and possible logical masking) of the signal perturbation through the combinational part. Such signal perturbation or glitch is latched in a non-deterministic manner. From now on, a signal can take 3 values: a logical one, a logical zero, or a glitch written ζ .

$$\text{Signal} := 0 \mid 1 \mid \zeta$$

A glitch can be masked in a combinatorial circuit by $\text{OR}(\zeta, 1) = 1$ or $\text{AND}(\zeta, 0) = 0$. The precise modelling of a glitched signal in a TMR circuit requires the knowledge of its correct value (present in the corresponding signals of the two other redundant modules). Consequently, the precise domain D_1 is extended as D_t to model a glitch propagation in a combinatorial circuit of one redundant module:

$$D_t = \{0, 1, \bar{0}, \bar{1}, 0^\zeta, 1^\zeta\}$$

where 0^ζ and 1^ζ represent respectively a glitched 0 and 1. That is, 0^ζ represents a glitch at one point of the circuit such that the value in the two other redundant copies is 0. A glitch on an incorrect signal with the value $\bar{0}$ (resp. $\bar{1}$) will be represented by the signal value 1^ζ (resp. 0^ζ). One example that illustrates the difference between a glitch and a corrupted value is:

$$D_1 : \bar{0} \vee_1 \bar{1} = 1 \quad D_t : 0^\zeta \vee_t 1^\zeta = 1^\zeta$$

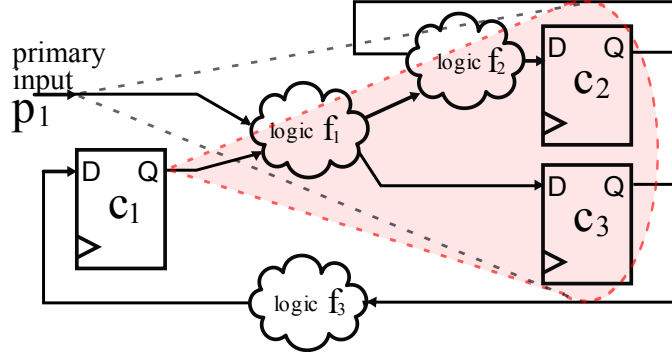
While in the first case, an *or* gate with corrupted but stable signals returns a correct value, in the second case, the glitch propagates.

While the precise domain D_1 requires the aforementioned extension to D_t , the domains D_2 and D_3 can overapproximate such glitch behavior with no extension. In particular, a glitched signal, as well as any possibly wrong stable signal, takes the value \bar{U} in D_2 . A glitched 1 (resp. 0) can be represented as $\{1, \bar{0}\}$ (resp. $\{0, \bar{1}\}$) in D_3 .

A glitch propagated to a memory cell is non-deterministically latched as true or false. It follows that the precise glitch modelling in D_t implies that any glitched signal 0^ζ (resp. 1^ζ) is non-deterministically latched as a correct 0 or as an incorrect $\bar{1}$ (resp. as a correct 1 or as an incorrect $\bar{0}$). This non determinism may lead to a significant state space growth in D_1 . The domains D_2 and D_3 avoid this inconvenience since glitched signals are expressed in the same logic as the latched values.

To take into consideration all possible effects of an SET, it is necessary to calculate the set of reachable states for all cases of SET injections. These cases include a fault injection either at the output of a logical gate/a memory cell or the mutually exclusive corruption of branches of a wire split. The union of the state spaces that can be reached in each of these corruption cases forms the reachable state set.

The precise SET modeling in D_t imposes significant computational overhead. Its two important bottlenecks are the need to consider all possible SET injection points and all possible non deterministic choices when a glitch is latched. Both points can be taken into account by a



■ **Figure 6** Combinational cones for SET modeling.

transition function that expresses a circuit state change during a clock cycle with an SET and returns a set of possibly corrupted states. In the next Section, we propose a safe approximation of the precise SET modeling in domains D_1 , D_2 , and D_3 .

6.2 Safe SET over-approximation

If a memory cell is connected by a combinational path to a component (wire or gate) where an SET occurs, this cell may be corrupted. We should find all sets of cells that can be corrupted at the same clock cycle to find the worst case. Each of these sets has a common combinational sub-circuit, in other words, a common combinational cone. The apex of such a cone is either the output of a memory cell or a primary input. A cone apex fully identifies a cone and the memory cells belonging to this cone.

In Figure 6, the cone with apex at c_1 includes both cells c_2 and c_3 . The cone with apex at p_1 also includes $\{c_1, c_2\}$. The cones with apexes at c_3 and c_2 contain $\{c_1\}$ and $\{c_2\}$ respectively.

As a result, the worst case scenario of any SET that happens inside a cone j is the union of all possible simultaneous corruptions of the memory cells $ms(j)$ in this cone. The power set $P(ms(j))$ is the set of all possible memory cell corruption configurations.

As soon as all corruption configurations are found, a new error injection procedure can be defined and used in both Algorithms 1 and 2 which remain the same. In particular, instead of mutually exclusive bit-flips injection to a state space S , expressed for SEU as $(\bigcup_{m_i \in M} S[m_i \leftarrow \mathbf{err}(m_i)])$, the corruption of the RSS by an SET is computed as the disjunction of possible simultaneous memory cells corruptions of the sets included in the cones after memory cells M or primary inputs I :

$$\bigcup_{j \in (M \cup I)} \left(\bigcup_{p \in P(ms(j))} S \left[\bigcap_{m_i \in p} m_i \leftarrow \mathbf{err}(m_i) \right] \right)$$

where $ms(j)$ is the subset of memory cells located in the cone with an apex at a memory cell or a primary input j .

Such corruption procedure is a safe over-approximation in the precise (D_t) and approximate (D_2 , D_3) domains. The complexity bottleneck of the approach is the power-set computation with a large number of memory cells in a single cone. However, in the case of the approximate logic domains D_2 and D_3 , we can consider only the worst case scenario: the simultaneous corruption of

■ **Table 4** Voter Minimization, SEU model, Boolean domains $D_1 \mid D_2 \mid D_3$.

	Circuit	FFs	Syn.	Semantic			Sem.Inp.			Sem.Out.		
Data Flow Int.				D_1	D_2	D_3	D_1	D_2	D_3	D_1	D_2	D_3
	Pipelined FP mult. [21]	121	0	0	0	0	0	0	0	0	0	0
	Pipelined log unit [21]	41	0	0	0	0	0	0	0	0	0	0
	Shift/Add mult. [33]	28	28	19	19	19	19	19	19	8	8	8
	ITC'99 [16](subset)											
Control Flow Intensive	b01 Flows comparator	5	3	3	3	3	3	3	3	3	3	3
	b02 BCD recognizer	4	3	2	3	3	2	3	3	2	3	3
	b03 Resource arbiter	30	29	17	29	17	17	29	17	17	29	17
	b06 Interrupt handler	9	3	3	3	3	3	3	3	3	3	3
	b08 Inclusion detector	21	21	21	21	21	0	21	0	0	21	0
	b09 Serial converter	28	21	20	20	–	20	20	–	20	20	–

A ‘–’ denotes an out of time termination of the analysis (>20 mins).

The “Syn.” column shows the results of the syntactic analysis.

all memory cells in a cone (without calculation of its powerset), computed as:

$$\bigcup_{j \in (M \cup I)} S \left[\bigcap_{m_i \in ms(j)} m_i \leftarrow \mathbf{err}(m_i) \right]$$

It may happen that the result of such SET insertion includes corrupted states that are not reachable because it does not take into consideration the internal error-masking capabilities of the combinational circuit. Nevertheless, we will see in the experiments that, for the analysis presented in this paper, such over-approximation is an appropriate choice.

7 Experimental results

The presented voter minimization technique has been implemented in Ocaml using the BDD library CUDD [43] and the OCaml interface MLCuddIDL [30]. Transition systems and set of states are expressed as BDD formulae [15].

The introduced logic domains (D_1 , D_2 , and D_3) are encoded with multiple bits (two for D_1 and D_2 ; four for D_3) and the associated operators (*e.g.*, Tables 2 and 3) are expressed as logic formulae over those bits. For instance, the values of D_1 can be encoded with two bits (a, b) as:

$$\begin{array}{ll} 1 & \text{as } (1, 1) \\ 0 & \text{as } (1, 0) \\ \bar{0} & \text{as } (0, 0) \\ \bar{1} & \text{as } (0, 1) \end{array}$$

In this encoding, the first bit a is the correctness bit, and the second one b is the value bit. The *NOT* operator of D_1 can be represented by the function:

$$\neg_1(a, b) = (a, \neg b)$$

We used the Quine-McCluskey algorithm to simplify the boolean functions corresponding to the *AND* and *OR* operators of D_1 . The *AND* operator is encoded as:

$$\wedge_1((a_1, b_1), (a_2, b_2)) = (a_3, b_3)$$

$$\begin{aligned} \text{where } a_3 &= ((a_1 \wedge a_2) \vee (a_1 \wedge \neg b_1) \vee (a_2 \wedge \neg b_2) \vee (\neg a_2 \wedge (\neg b_1 \wedge b_2)) \vee (\neg a_1 \wedge (\neg b_2 \wedge b_1))) \\ b_3 &= b_1 \wedge b_2 \end{aligned}$$

And the *OR* operator is encoded as:

$$\vee_1((a_1, b_1), (a_2, b_2)) = (a_3, b_3)$$

$$\begin{aligned} \text{where } a_3 &= ((a_1 \wedge a_2) \vee (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (\neg a_1 \wedge (\neg b_1 \wedge b_2)) \vee (\neg a_2 \wedge (\neg b_2 \wedge b_1))) \\ b_3 &= b_1 \vee b_2 \end{aligned}$$

An encoding is needed for all the presented abstract domains. At least three different values for each bit (0, 1, and an incorrect value) must be encoded. Such an encoding cannot be replaced with circuit simulation or emulation where only logical one and zero are available.

BDDs proved to be quite efficient to express the data structures and the processing required by our technique. We made use of Rudell's sifting reordering [39] while building and applying the transition function. It allowed the semantic analysis of circuits up to 100 memory cells on a standard PC (Intel Core i5-2430M/2Gb-DDR3). For comparison, without reordering, the negative impact of big BDD structures on the algorithm performance was observed already for circuits with 20-30 memory cells. We did not put much efforts in the optimization but we believe that there remain much opportunities for improvement.

We used both fault-models $SEU(1, K)$ and $SET(1, K)$ with $K = 50$, which allows K cycles/-transitions to be computed effectively (Δ^K). The obtained results are *a fortiori* valid for any $K \geq 50$. However, for non-restrictive trivial input/output specification and small circuits, it is not worth to choose higher K values since all reachable states might be visited within a small number of execution steps K , and no further optimization will be achieved even if we continue the execution. When all reachable states are visited the execution can be stopped even if K steps have not been fully performed. Thanks to the encoding of input/output specification into the circuit structure (Section 5), the reachable states also contain the information about the values of input signals and the relevance of primary outputs (for the error-propagation analysis). The number of steps K needed to explore the whole state space varies depending on the specification and circuit complexity. For small circuit (*e.g.*, *b02*, *b01*) with simple input/output specification (*e.g.*, only the reset at the very beginning), we visit all reachable states in $K < 10$ steps. On the other hand, for larger circuits (shift/add multipliers or the circuit *b08*) with explicit complex input/output interface specifications (FSMs with 10 and more states), a higher value of K is rewarding and allows us to catch error masking behaviors that happen regularly (*e.g.*, circuit restarts or returns to the initial state in cyclic FSMs within every 30-40 cycles).

Our analysis has been applied to common arithmetic units taken from the *OpenCores* project [21] and from the *ITC'99* benchmark suite [16]. For each circuit, we defined non-restrictive input-output specification for the sake of generality. For the majority of the circuits, the input pattern specifies only synchronous reset at its initialization phase and no further reset (*b01*, *b02*, *b03*, *b04*, *b06*, *b09*). Such non-restrictive patterns may reduce achievable optimizations, which could be significantly increased if more details about the behavior of the surrounding circuit were provided. However, for the shift/add multiplier [33] the input-output specification is dictated by its functionality. The produced output is relevant only two cycle after the **start** signal has been raised (one cycle to fetch new data plus at least one cycle to process it). Since we should not assume when the output is read out, we suppose that the data output may be read at any time two cycles after the last

start and until the next **start**. As a result, our semantic analysis with this output specification shows that only the 8 product bits should be protected by voters.

Circuit *b08* represents a group of self-stabilizing circuits that return to their initial state (and wait for the next **start**) within a bounded number of cycles (for *b08*, this period is 8 cycles). Additionally, by functionality, the circuit is supposed to be restarted periodically. The corresponding input and output specification allowed us to suppress all voters. We would like to highlight that any circuit with internal counters has a similar behavior of self-stabilization (the shift/add multiplier is another example).

Table 4 summarizes the results of the analysis on those circuits in D_1 , D_2 , and D_3 , with the fault-model $SEU(1, K)$. The column FFs shows the total number of memory cells in the original circuit, while the other columns show the number of remaining voters in the TMR circuit after the syntactic and semantic steps (without, with input, with input and output interfaces). In each case, we give the results obtained with the three logic domains.

The syntactic step eliminates all voters in circuits with a pipelined architecture such as adders, multipliers, or logarithmic units. With rolling pipelined architectures, a control part and a looped dataflow circuit may require voter protection (*e.g.*, none of the 28 voters of the shift/add multiplier are removed with only the syntactic analysis).

In general, control intensive circuits require a protection of their FSMs. Almost all memory cells of the serial flow comparator (*b01*) or the serial-to-serial converter (*b09*) have to be protected. Nevertheless, our analysis is capable of suppressing a significant amount of voters in many control intensive circuits. A circuit is usually composed of data and control flow parts and we can expect that most voters in the data flow part can be suppressed.

The logic domain D_2 is, most of the time, precise enough. However, correcting a bit-flip in D_2 (*e.g.*, $0 \rightarrow \bar{U} \rightarrow U$) loses information. In some circuits, like *b03* and *b08*, substantial logical error masking is performed by an FSM and the analysis fails to detect it.

The precision of the domain D_3 allows us to achieve better optimizations than the domain D_2 in circuits *b03* and *b08* (see Table 4). With D_3 , the corrupted FSM will recover to a precise state, while with D_2 its cells will recover to the correct unknown value U . This precise state plays a crucial role to show that the rest of the circuit, that depends on this FSM, will be cleaned up too.

The results for $SET(1, K)$ are shown in Table 5. The number of suppressed voters did not change with D_2 . However, even the proposed approximations in Section 6.2 does not help to resolve the complexity problem for some circuits when analyzed with D_1 and D_3 . The bottleneck results from the large number of corruption combinations if a single combinatorial cone includes many memory cells. For example, in the circuit *b03*, there is an FSM of 2 cells where each cell is connected through a combinatorial circuit to 26 memory cells (mainly controlling their enable signals). As a result, to approximate the impact of an SET in this FSM, we have to calculate all possible corruption combinations of 26 cells, which is 2^{26} configurations. The circuits that could not be analysed are marked by * in Table 5.

The scalability of logic domains D_1 , D_2 , and D_3 has also been compared. Figure 7 presents the growth of the RSS S_i after i iterations (see Section 3) for the *b03* and *b06* circuits. The fixed point is reached with less iterations in D_2 , and the number of states grows exponentially for D_1 versus linearly for D_2 . The same behavior is observed in all considered circuits.

The logic domain D_3 reaches the fixed-point as fast as D_1 while keeping the same precision. This fact is demonstrated in Table 6 where we measured the number of cycles to calculate the RSS for each domain (the column “# iterations”). The column “seconds” gives the execution time spent to calculate the RSS, and the last column “# BDD nodes”, gives the complexity of the RSS BDD representation in terms of allocated BDD nodes. On the one hand, the number of

■ **Table 5** Voter Minimization, SET model, Boolean domains $D_1 \mid D_2 \mid D_3$.

	Circuit	FFs	Synt.	Semantic			Sem.Inp.			Sem.Out.		
Data Flow Int.				D_1	D_2	D_3	D_1	D_2	D_3	D_1	D_2	D_3
	Pipelined FP mult. [21]	121	0	0	0	0	0	0	0	0	0	0
	Pipelined log unit [21]	41	0	0	0	0	0	0	0	0	0	0
	Shift/Add mult. [33]	28	28	–	19	–	–	19	–	–	8	–
	ITC'99 [16](subset)											
Control Flow Intensive	<i>b01</i> Flows comparator	5	3	3	3	3	3	3	3	3	3	3
	<i>b02</i> BCD recognizer	4	3	2	3	3	2	3	3	2	3	3
	<i>b03</i> Resource arbiter	30	29	–	29	–	–	29	–	–	29	–
	<i>b06</i> Interrupt handler	9	3	3	3	3	3	3	3	3	3	3
	<i>b08</i> Inclusion detector	21	21	–	21	21	–	21	0	–	21	0
	<i>b09</i> Serial converter	28	21	–	20	–	–	20	–	–	20	–

A '–' denotes an out of time termination of the analysis (>20 mins)

'Syn.' column shows the results of syntactic analysis

BDD nodes allocated to represent the RSS in larger circuits (*b03*, *b08*, *b09*) is much smaller with D_3 than with D_1 . On the other hand, the BDD structures in D_3 require more variables and are more time consuming to manipulate. The domain D_3 overapproximates the RSS (see Section 3.3), which leads to less allocated nodes in the larger circuits. While it allows us to keep the necessary precision for optimizations comparable to the ones allowed by D_1 , our existing implementation of D_3 would require further optimizations to be considered as an interesting compromise.

The bar graph of Figure 8 shows the ratio of the size of the RSS in D_1 to the corresponding size in D_2 . The RSSs in D_1 are several orders larger than the corresponding ones in D_2 . The most computation demanding step of the whole analysis is checking error propagation (see Section 5). A prohibiting growth of BDD structures representing the set of states E_i was observed with D_1 for circuits of around 30 memory cells. The logic domain D_2 allows the analysis (with input and output interfaces) of much larger circuits, up to 100 cells.

In order to evaluate the benefits of our analysis, TMR has been applied to the benchmarks with the minimized set of voters. The inserted voters are triplicated following the practice in the existing industrial tools to avoid a single-point of failure and to protect against SETs. The final circuits have been synthesized with *Synplify Pro* with no optimization applied (Resource Sharing, FSM Optimization, etc.). As a case study, we have chosen Flash-based ProASIC3 FPGA as a synthesis target. Its configuration memory is immune to soft-errors [34] and data memory is protected with voters. Table 7 compares the size and maximum frequency of the circuit with full TMR (*i.e.*, voters after each FF) versus TMR with the optimized number of voters. The gains are presented in terms of the required FPGA hardware Core Cells (*hw* column) and maximum synthesizable frequency (*MHz* column). The gain in the maximum frequency depends on the location of the removed voters (in the circuit critical path or not). The reduction in area directly depends on the number of suppressed voters (up to 55%).

Instead of using symbolic simulation with BDDs, we could have chosen a satisfiability-based approach and have encoded the same algorithm as a bounded model checking problem. We could benefit from the efficiency of SAT solvers and possibly improve the scalability of our algorithms.

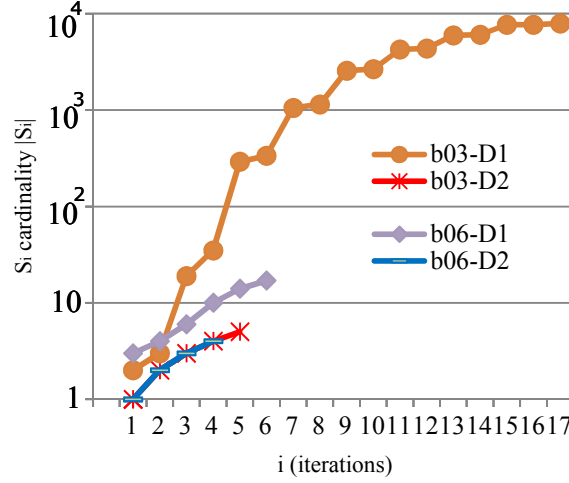


Figure 7 Logic Domain Comparison: Reachable State Space Size.

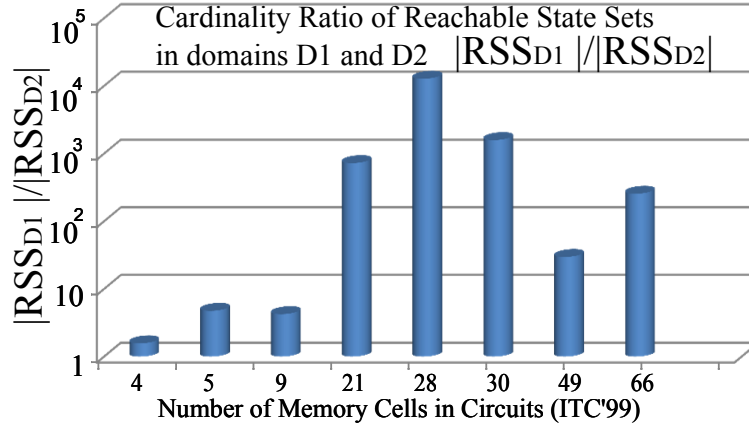


Figure 8 Logic Domain Comparison: Size Ratio of RSS.

However, since we use multi-value logic domains and the error-correction/-injection operators **vot** and **err**, a SAT-based approach would require a non-trivial circuit orchestration. This option offers nonetheless a possible avenue for future research.

8 Related work

Existing industrial tools for applying TMR into FPGA protect against both kinds of soft error, SEUs and SETs. They include the Xilinx XTMR tool [6,46], BYU/Los Alamos National Laboratory B-TMR [37], Synopsys's Synplify Premier [44], and Mentor Graphics Precision Hi-Rel [19]. In these tools, TMR is applied to circuit parts chosen by the user and, thus, the resulting circuits might not be fault-tolerant unless voters are inserted after *each* memory cell and primary circuit outputs. [19] proposes a protection technique against SEUs that requires only memory cells triplication with a majority voter insertion. But this approach relies on the assumption that only memory cells are influenced by radiation particles and that no signal perturbations in a combinatorial circuit occur. Thus, unlike our technique, the technique of [19] protects only against SEUs and not against SETs.

■ **Table 6** Time and memory resources to calculate the RSS.

		δ , sec	# iterations	seconds	# BDD nodes
b01	D_1	0.037	9	0.01	156
	D_2	0.037	6	0.01	78
	D_3	0.060	6	0.01	151
b02	D_1	0.020	9	0.005	81
	D_2	0.020	9	0.04	66
	D_3	0.024	9	0.01	127
b03	D_1	0.42	17	2.53	1506
	D_2	0.44	7	0.28	311
	D_3	875.670	7	235.13	668
b06	D_1	0.044	8	0.024	473
	D_2	0.052	6	0.018	130
	D_3	0.056	6	0.02	256
b08	D_1	0.364	40	3.14	27813
	D_2	0.356	5	0.02	324
	D_3	41.49	5	48.08	1222
b09	D_1	31.332	32	27.57	2919
	D_2	0.852	20	1.04	446
	D_3	>1000	-	-	-

While our static analysis uses exclusively logical masking to tolerate transient errors, many other works rely on electrical and latching-window properties of hardware to estimate the chance that errors will not manifest in failures. This is the primary reason why a good part of research on voter insertion, Selective Triple-Modular Redundancy (STMR), and partial hardware redundancy mainly focus on probabilistic approaches [2, 28, 31, 40]. Contrary to our approach, they are not interested in formal guarantees that the final circuit tolerates a fault-model. [31] shows how selective voter insertion minimizes the negative timing impact of TMR. In [38], probabilities are used to apply TMR on selected portions of the circuit (STMR). In [40], STMR of combinational circuits specifies input interfaces using input signal probabilities. The main advantage of STMR over TMR is that the area of the STMR circuit is roughly two-thirds of the area of the TMR circuit. An original probabilistic-based idea is given in [29] that allows a certain level of degradation in output correctness in order to optimize TMR at a Data Flow Graph (DFG) abstraction level. While this technique is originally dedicated to heterogenous systems, it could be applied to Digital Signal Processing (DSP) hardware as well. Since the proposed methods are probabilistic, some errors may propagate to primary outputs. In our approach, the circuit is guaranteed to mask *all* possible errors of the considered fault model.

Other works use model-checking to guarantee user-defined fault-tolerance properties [3, 41]. [41] investigates which memory cells in SpaceWire node have to be protected so that even under an SEU occurrence the circuit keeps its functional properties, expressed as 39 assertions in linear temporal logic. If a cell is protected (fabricated with a special technology), an SEU cannot corrupt

■ **Table 7** Frequency and area gain of optimized *vs* full TMR.

	TMR circuit	voters	MHz	gain	hw	gain
Data Flow Intens.	Pipelined FP Multiplier 8x8	121	60.5		2338	
	Optimized	0	71.0	17.4%	1831	21.7%
	Pipelined logarithm unit	41	128.3		693	
	Optimized	0	184.1	43.5%	447	35.5%
	Shift/Add multiplier 8x8	28	106.0		537	
	Optimized	8	108.0	1.9%	408	24.0%
Control Flow Intensive	b01 Flows comparator	5	162.6		126	
	Optimized	3	162.6	0%	114	9.5%
	b02 BCD recognizer	4	181.9		69	
	Optimized	2	206.6	13.6%	60	13.1%
	b03 Resource arbiter	30	81.6		594	
	Optimized	17	109.0	33.6%	576	3.0%
	b06 Interrupt handler	9	144.8		168	
	Optimized	3	144.8	0%	134	20.2%
	b08 Inclusions detector	21	115.4		484	
	Optimized	0	142.4	23.4%	216	55.4%
	b09 Serial converter	28	89.4		584	
	Optimized	20	95.0	6.3%	565	3.3%

it. On the other hand, a protected cell consumes more power than a non-protected memory cell. As a result of verification-guided replacement of protected cells by their non-protected alternatives, a 4.45X reduction in power has been achieved. The work [3] formally proves that some system properties of ATM controller are kept if an SEU happens. The authors evaluate the probability to obtain the expected property under faults.

Another group of formal studies investigates sequential circuit robustness symbolically [4, 23] or by interpolation [13]. Since robustness is introduced probabilistically these work combine both formal and probabilistic worlds.

While the aforementioned formal studies do not address voter minimization, their approaches to fault-tolerance and robustness are related to our work.

It is worth noticing that the introduced reachability analysis with multi-value encoding can be also interpreted within the well-known tainting dataflow-based analysis [18] and path sensitisation theories [14]. The former assigns a security-related mark to each information bit and tracks its propagation, just like we tag some bits as erroneous. The later approaches check if there is a path so that a signal change along that path alters the output. In our case, the signal change corresponds to an error injection, *e.g.*, a bit-flip, and we check whether this change can propagate to corrupt the output.

9 Conclusion

We proposed a logic-level verification-guided approach to minimize the number of voters in TMR circuits that guarantees a user-defined fault-model to be masked. Our approach is based on reachable state set computations and input/output interface specifications. In order to avoid analyzing the triplicated circuit, we introduced three logic domains, which allowed us to perform the analysis on a single copy of the circuit. Our analysis shows that some voters are useless and can be safely removed from the TMR application. We have used as case studies several arithmetic circuits as well as the benchmark suite *ITC'99*. They show that our technique allows not only a significant reduction in the amount of hardware resources (up to 35% for data flow intensive circuits and up to 55% for control flow intensive ones), but also a significant increase in the clock rate, compared to the full TMR method that inserts a voter after each memory cell. We demonstrated that the choice of the logic domain influences the scalability of the analysis and its precision. We considered both SEU and SET fault-models and explained the modeling methodology. As the experimental results show, the same level of optimization can be reached for both fault-models, but the SET model implies a potentially large number of corruption combinations to be examined, which can cause an analysis bottleneck.

In this article, we have only considered hardware redundancy (TMR) but our approach also applies to time redundancy. Time-redundant schemes mask errors by voting on re-computed data. Such schemes reuse the combinational part of the circuit and have a lower hardware overhead at the price of a lower throughput. We have proposed new fault-tolerance techniques based on time-redundancy in [12] and [11]. We have demonstrated in [9] how the present voter minimization technique could be applied to them.

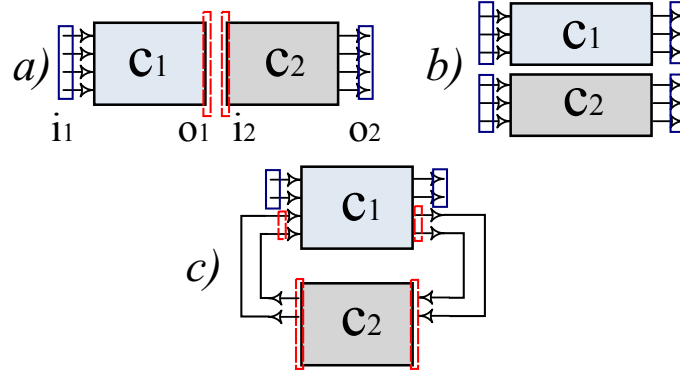
Further research directions include taking into account other optimization criteria such as frequency and allowing the analysis of large circuits by making our approach modular. We review these issues in turn.

Frequency maximization

Voters ordering, discussed in Section 3.2, could also take into account other optimization criteria than voter minimization. For instance, we may increase the maximum synthesizable frequency by removing first the voters on the critical path. However, removing a voter from the critical path may make another path critical. Thus, the choice of the next voter to remove depends not only on the existing ordering but also on the current critical path. However, the critical path strategy may not result in the minimal number of voters. In this sense, the two criteria “number of voters” and “synthesizable frequency” are orthogonal, and bi-criteria optimization must be studied.

Modularity

Applying our analysis in a modular manner can increase its scalability and, consequently, the applicability of the proposed technique to large circuits. The hierarchical compositional design of today's circuits makes it natural to decompose a circuit to the IPs of its block-by-block structure. Such structural partitioning requires the deep design understanding and has already been used in the model checking of Intel CPUs [1]. In our case, the presented analysis can be applied to circuit sub-components after the decomposition. After the minimization of internal voters in each sub-circuit, the components should be interconnected again to rebuild the whole design. However, the interconnection wires should include voters to guaranty the fault-tolerance property of the final optimized circuit. Such an approach is not optimal even if the local input/output specifications are precise, because some of the interconnection voters may be redundant. Only a global analysis



■ **Figure 9** *a)* sequential, *b)* parallel, and *c)* feedback circuit decomposition.

on the blocks containing such wire with a voter (as an input or output wire) can safely remove interconnection voters.

If a decomposition in sub-circuits is not known, the circuit netlist has to be automatically divided and the input-output specifications of its parts have to be found. These steps by themselves present complex tasks and require deep investigation. Here, we sketch some preliminary ideas about how these problems can be solved. First, a circuit netlist can be separated according to some syntactic criteria, *e.g.*, the circuit cuts should be performed at wires that are included in the biggest number of sequential loops. Such an approach eliminates as many sequential loops as possible by reducing the number of sequential loops in each sub-component. It limits the number of potential points where the voters have to be inserted.

After the circuit decomposition, our semantic analysis can be applied to each of its sub-parts. The main difficulty lies in the identification of input/output specification of each sub-circuit to perform the local semantic analyses. Figure 9 presents three cases of the circuit separation: *a)* sequential, *b)* parallel, and *c)* feedback decomposition.

While the input/output specification for c_1 and c_2 sub-circuits can be extracted from the global specification in the parallel decomposition (case *b*, Figure 9), the sequential and feedback decompositions (cases *a* and *c*) create unknown internal specifications (marked in red). They have to be found for each sub-part. Consider, for instance, the unknown input specification i_2 for the sequential decomposition (case *a*). The signals in i_2 are the outputs o_1 . Since the netlist c_1 and its input specification i_1 fully describe the behavior of c_1 , o_1 and i_2 can be described by the same NBA. In the worst case, such NBA could be as big as c_1 multiplied by the size of i_1 , which can be prohibitive for the following semantic analysis of c_2 sub-circuit. Consequently, the extracted NBA should be over-approximated to lower the complexity. Naturally, the over-approximation may influence the precision of the further semantic voter minimization in c_2 . The feedback decomposition is even more complex because of the mutual dependency between sub-components c_1 and c_2 .

These modularity issues are complex but important and valuable since many other static analyses of circuits could benefit from them.

References

- 1 M.D. Aagaard, R.B. Jones, and C.-J.H. Seger. Formal verification using parametric representations of boolean constraints. In *Design Automation Conference (DAC)*, pages 402–407, 1999.
- 2 B. Baykant Alagoz. Fault masking by probabilistic voting. *OncuBilim Algorithm And Systems Labs*, 9(1), 2009.
- 3 S. Baarir, C. Braunstein, et al. Complementary formal approaches for dependability analysis. In *IEEE Int.Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 331–339, 2009. doi:10.1109/DFT.2009.21.
- 4 S. Baarir et al. Feasibility analysis for MEU robustness quantification by symbolic model checking. In *Proceedings in Formal Methods of Software Design*, 2011.
- 5 A.L. Bogorad et al. On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance. *IEEE Trans. on Nuclear Science*, pages 2804–2806, 2011.
- 6 Brendan Bridgford, Carl Carmichael, and Chen Wei Tseng. Single-event upset mitigation selection guide. Application Note XAPP987 (v1.0), Xilinx, 2008.
- 7 P. Brinkley, P. Avnet, and C. Carmichael. SEU mitigation design techniques for the XQR4000XL. Xilinx, 2000.
- 8 S. P. Buchner and M. P. Baze. Single-event transients in fast electronic circuits. *IEEE NSREC Short Course*, pages 1–105, 2001.
- 9 Dmitry Burlyaev. Design, optimization, and formal verification of circuit fault-tolerance techniques. *PhD thesis Joseph Fourier University/INRIA*, November 2015.
- 10 Dmitry Burlyaev, Pascal Fradet, and Alain Girault. Verification-guided voter minimization in triple-modular redundant circuits. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24–28, 2014*, pages 1–6, 2014.
- 11 Dmitry Burlyaev, Pascal Fradet, and Alain Girault. Automatic time-redundancy transformation for fault-tolerant circuits. *International Symposium on Field-Programmable Gate Arrays*, pages 218–227, February 2015.
- 12 Dmitry Burlyaev, Pascal Fradet, and Alain Girault. Time-redundancy transformations for adaptive fault-tolerant circuits. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8, 2015.
- 13 Gianpiero Cabodi and Satnam Singh, editors. *Complete and Effective Robustness Checking by Means of Interpolation*. Formal Methods in Computer-Aided Design (FMCAD), 2012.
- 14 Albert C. L. Chiang, Irving S. Reed, and Anthony V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Trans. Computers*, 21(2):189–195, 1972. doi:10.1109/TC.1972.5008925.
- 15 E.M. Clarke, J.R. Burch, O. Grumberg, D.E. Long, and K.L. McMillan. Automatic verification of sequential circuit designs. *Phil. Trans. R. Soc. Lond. series A*, 339:105–120, 1992.
- 16 F. Corno, M.S. Reorda, and G. Squillero. RT-level ITC’99 benchmarks and first ATPG results. *Design Test of Computers*, pages 44–53, 2000. doi:10.1109/54.867894.
- 17 Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.
- 18 Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. doi:10.1145/359636.359712.
- 19 Roger D. Do. New tool for FPGA designers mitigates soft errors within synthesis. *DSP-FPGA.com Magazine*, December 2011.
- 20 P.E. Dodd, M.R. Shaneyfelt, J.R. Schwank, and G.L. Hash. Neutron-induced soft errors, latchup, and comparison of SER test methods for SRAM technologies. *International Electron Devices Meeting*, pages 333–336, 2002.
- 21 Michael Dunn. Open source hardware IPs: OpenCores project. Logarithm Unit; Launchbird Design Systems, Inc. Floating Point Multiplier. URL: <http://opencores.org>.
- 22 Guy Even, Joseph (Seffi) Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In *Int. Conf. on Int. Prog. and Combinatorial Opt.*, pages 14–28, 1995.
- 23 Görschwin Fey, André Sülflow, and Rolf Drechsler. Computing bounds for fault tolerance using formal techniques. In *Proceedings of the 46th Design Automation Conference, DAC*, pages 190–195, 2009.
- 24 Sandi Habinc. Functional triple modular redundancy FTMR. *European Space Agency Contract Report*, FPGA-003-01, 2002.
- 25 K.J. Hass and J.W. Ambles. Single event transients in deep submicron CMOS. In *42nd Midwest Symposium on Circuits and Systems*, pages 122–125 vol. 1, 1999.
- 26 John P. Hayes, Ilia Polian, and Bernd Becker. An analysis framework for transient-error tolerance. In *25th IEEE VLSI Test Symposium (VTS 2007)*, 6–10 May 2007, Berkeley, California, USA, pages 249–255, 2007.
- 27 T. Heijmen. Soft-error vulnerability of sub-100-nm flip-flops. *14th IEEE Int.On-Line Testing Symposium*, pages 247–252, 2008.
- 28 *International Test Conference, ITC’03, Breaking Test Interface Bottlenecks*, Charlotte (NC), USA, 2003. IEEE Computer Society. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8970>.
- 29 T. Imagawa, H. Tsutsui, H. Ochi, and T. Sato. A cost-effective selective TMR for heterogeneous coarse-grained reconfigurable architectures based on DFG-level vulnerability analysis. In *Design, Automation, and Test in Europe (DATE)*, pages 701–706, March 2013. doi:10.7873/DATE.2013.151.
- 30 B. Jeannet. MLCUDDIDL: An OCaml interface for the CUDD BDD library. <http://pop-art.inrialpes.fr/~bjeannet/mlxxidl-forge/mlcuddidl/index.html>. Accessed: 2014-09-01.

- 31 Jonathan M. Johnson and Michael J. Wirthlin. Voter insertion algorithms for FPGA designs using triple modular redundancy. In *FPGA*, pages 249–258, 2010.
- 32 R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 43:85–103, 1972.
- 33 Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.
- 34 Microsemi Corporation. *Neutron-Induced Single Event Upset SEU*, 55800021-0/8.11 edition, 2011.
- 35 H. T. Nguyen and Y. Yagil. A systematic approach to SER estimation and solutions. *Proc. Int. Reliability Physics Symp.*, pages 60–70, April 2003.
- 36 Ilia Polian, Bernd Becker, Masato Nakasato, Satoshi Ohtake, and Hideo Fujiwara. Low-cost hardening of image processing applications against soft errors. In *21th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2006)*, 4-6 October 2006, Arlington, Virginia, USA, pages 274–279, 2006.
- 37 B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving FPGA design robustness with partial TMR. *IEEE International Reliability Physics Symposium*, pages 226–232, 2006.
- 38 O. Ruano, P. Reviriego, and J.A. Maestro. Automatic insertion of selective TMR for SEU mitigation. *European Conference on Radiation and its Effects on Components and Systems*, pages 284–287, 2008.
- 39 Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of CAD-93*, pages 42–47, 1993.
- 40 P.K. Samudrala et al. Selective triple modular redundancy based single-event upset tolerant synthesis for FPGAs. *IEEE Transactions on Nuclear Science*, pages 284–287, October 2004.
- 41 S.A. Seshia, Wenchao Li, and S Mitra. Verification-guided soft error resilience. In *DATE '07*, pages 1–6, 2007. doi:10.1109/DATE.2007.364501.
- 42 P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002. doi:10.1109/DSN.2002.1028924.
- 43 F. Somenzi. CUDD: CU Decision Diagram Package, release 2.5.0. <http://vlsi.colorado.edu/~fabio/CUDD>. Accessed: 2014-09-01.
- 44 Angela Sutton. Creating highly reliable FPGA designs. *Military&Aerospace Technical Bullentin*, Issue 1:5–7, 2013.
- 45 J. von Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies*, Princeton Univ. Press, pages 43–98, 1956.
- 46 Xilinx TMR Tool product brief, 2006.
- 47 J.F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996.

Glossary

BDD Binary Decision Diagram.

DFG Data Flow Graph.

DSP Digital Signal Processing.

FF Flip-Flop.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

IC Integrated Circuit.

MVFS Minimum Vertex Feedback Set.

NBA Non-deterministic Büchi Automaton.

RSS Reachable State Set.

SER Soft-Error Rate.

SET Single-Event Transient.

SEU Single-Event Upset.

STMR Selective Triple-Modular Redundancy.

TET Transient Error Tolerance.

TMR Triple-Modular Redundancy.