Leibniz Transactions on Embedded Systems

Volume 3 | Issue 1 | June 2016

Vol. 3, Issue 1 ISSN 2199-2002 http://www.dagstuhl.de/lites

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
June 2016

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.

Digital Object Identifier
10.4230/LITES-v003-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.

Contact
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de
http://www.dagstuhl.de/lites

Contents

Regular Papers

Programming Language Constructs Supporting Fault Tolerance

Christina Houben[1] **and Sebastian Houben**[2]

1 **Rheinische Friedrich-Wilhelms-Universität**
Chemical Institutes, Bonn, Germany
c-k-houben@uni-bonn.de

2 **Ruhr-Universität Bochum**
Institute for Neural Computation, Bochum, Germany
http://orcid.org/0000-0002-2036-419X
sebastian.houben@ini.rub.de

Abstract

In order to render software viable for highly safety-critical applications, we describe how to incorporate fault tolerance mechanisms into the real-time programming language PEARL. Therefore, we present, classify, evaluate and illustrate known fault tolerance methods for software. We link them together with the requirements of the international standard IEC 61508-3 for functional safety. We contribute PEARL-2020 programming language constructs for fault tolerance methods that need to be implemented by operating systems, and code-snippets as well as libraries for those independent from runtime systems.

2012 ACM Subject Classification Control Structures and Microprogramming Control Structure Reliability, Testing, and Fault-Tolerance, Programming Languages, Language Constructs and Features, Computers in Other Systems, Real-time, Advanced Driver Assistance Systems, Space Flight

Keywords and phrases fault tolerance, functional safety, PEARL, embedded systems, software engineering

Digital Object Identifier 10.4230/LITES-v003-i001-a001

Received 2015-05-02 **Accepted** 2016-04-05 **Published** 2016-06-10

1 Introduction

Highly safety-critical applications need automation systems that are failsafe or at least fault tolerant. An automation system is called failsafe if it falls back into a stable state with a sufficient degree of functional safety. Functional safety is a system's property guaranteeing that the risk to harm human beings, environment or other assets is below a risk limit [3]. Automation systems are increasingly composed of software, thereby allowing to control more complex applications than pure hardware-based solutions, providing greater flexibility in adapting systems to changing requirements [22], consuming less space than mechanical constructions [9], and permitting to change system functionality by remote maintenance [17, 25]. Unfortunately, software cannot fall back into a safe state, triggered by laws of nature, like hardware [22]. Therefore, software systems have to use fault tolerance methods in order to provide functional safety as demanded in the international standard IEC 61508-3 [3]. Fault tolerance (FT) refers to a system fulfilling a specified function, even if a limited number of subsystems are erroneous [38]. FT methods rather prevent the consequences of a software error than the occurrence of the error itself [41]. FT methods, hence, are composed of error recognition and error treatment [38].

Our long-term objective is to adapt the real-time programming language PEARL-90 to functional safety as defined in the normative part of IEC 61508-3 [3]. In this paper we focus on the topic of fault tolerance by propagating syntax and semantics to its derivative PEARL-2020

and elucidating the rationale behind our adaptations. Introducing FT into PEARL is beneficial, since PEARL was the first language providing user-friendly concurrency concepts. These simple but powerful concurrency concepts render PEARL appropriate for highly safety-related software [22] which leads to a wide deployment throughout Europe [31]. PEARL is particularly tailored to process control due to its system part that addresses peripherals. It is standardised in DIN 66253-2 as PEARL-90 [4] and DIN 66253 Part 3 as Multiprocessor PEARL [1]. The latter is composed of two parts: the first part is a parametrisation language for compilers, binders and loaders and the second part describes various communication protocols for synchronous and asynchronous task cooperation and message passing. For a detailed introduction we refer to [19]. Currently, Müller and Schaible [32] map out a PEARL-90 compiler which will serve as a base for a future PEARL-2020 compiler. In our paper we contribute to the topic of fault tolerance in software as follows:

- We provide a complete overview of fault tolerance methods including their definition, classification and one or more coding examples, and
- present an evaluation of each FT method according to the insights from the examples with respect to stumbling blocks for programmers and suitability as language elements, code snippets or library implementations. In detail, we arrive at (c.f. Table 4):
  - additional PEARL syntax elements and semantics for assertions, load shedding, monotone tasks and the Byzantine method,
  - code snippets in PEARL for forward recovery, functional and implementation diversity,
  - PEARL library implementations for majority voting and other plausibility checks,
  - further changes to the PEARL-90 programming language, i.e. judiciously removing backward recovery, temporal redundancy and dynamic reconfiguration.

The outline of this paper is as follows. We start with related work in Section 2. Section 3 presents a comprehensive set of known fault tolerance methods for software with their classification and examples. A few examples came from literature, most of them are devised by ourselves in the application domains space flight and advanced driver assistance systems. Only one example per FT method will be realised in the final language standard, but we state them for comparison. This is the base for our framework addressing fault tolerance for software, in particular for PEARL. The framework comprises examples, code-snippets, library procedures and language primitives. In Section 4, we further explain the rationale behind this subdivision and link the FT methods with requirements from IEC 61508-3. We publish our implementations under the following web addresses:

- concrete examples and general fault tolerance handlers for PEARL-90:
  `http://sourceforge.net/projects/openpearl/files/Example`
- a fault tolerance library for PEARL-2020:
  `http://www.real-time.de/service/downloads.html`

2 Related Work

Most of the literature focuses on a single FT method. There are few papers that aim at combining more than one method: Shelton [36] derived patterns from three different FT methods considering task allocation and graceful degradation. DanYong and YongDuan [15] and Chen et al. [13] inspected fault tolerance from the mathematical point of view setting up differential equations in order to assess discrepancies from nominal values. Anwar [6] monitors a mainly software driven system and switches to a redundant mechanical system only in case of an error. A different area of application is treated by [30] who invented a fault tolerant TTP/C communication protocol. In contrast to these, our paper targets a larger amount of FT methods as can be seen in Table 4.

PEARL encompasses multiple derivatives, several of which already provide approaches for integrating FT. Multiprocessor PEARL offers keywords for dynamic reconfiguration. PEARL-90 allows for redundancy with the help of its easy to use concurrency concepts. This language derivative is branched into four subsets by [24], each addressing one of the four safety integrity levels (SILs) of IEC 61508. These subsets are Table-PEARL for SIL 4, Verifiable PEARL for SIL 3, Safe PEARL for SIL 2 and High Integrity-PEARL for SIL 1. Out of these, High Integrity-PEARL entails language elements to state alternative procedure bodies. Object-oriented derivatives of PEARL are PEARL* and Object-PEARL. PEARL* differs from PEARL-90 by keywords for objects and interfaces. Object-PEARL implements alternative methods and monotone tasks as FT concepts [14].

Fault tolerance in other languages is covered in Ada and languages like GRAFCET [2] for programmable logic controllers (PLCs). SPARK and Ravenscar inherit from Ada but restrict its variability for highly safety-critical applications. SPARK avoids error-prone language constructs and provides a certified tool chain for compiling [7], while Ravenscar parametrises the scheduling policy used in runtime systems in order to detect deadlocks and to guarantee timely task execution. PLC languages are standardised with libraries encompassing building blocks that support fault tolerance.

3 Fault Tolerance Methods

Fault tolerance methods are composed of error recognition and error treatment [38]. Error recognition is covered in the next section. A categorisation of error treatment methods is given in Section 3.2 and their elementary techniques in Section 3.3.

3.1 Error Recognition

In order to recognise an error, an automation system needs additional information on the range of values of an expected result. This additional information can either entail functional content, checked by voters, or is based on flow of control, checked by watchdogs. Examples for watchdogs are given in Table 1. Voters can be classified into absolute or relative voters. Absolute voters use hard-coded predicates as additional information, while relative voters compare the results of various implementations [10]. As a conclusion, absolute voters return whether a result is feasible or not. Since hard-coded predicates can be checked faster and more easily, they are predominantly used for dynamic redundancy. In contrast, a relative voter additionally returns a certain value of the output range or a combination of the output values. Examples of both types of voters are shown in Table 1.

Our first PEARL example is a majority voter. It can be used for `N` floating point numbers and returns the majority value if more than `N // 2` units possess a similar value or `ERROR_FLOAT` if not. Similar values are found by initial clustering. Afterwards, the cluster with most elements is processed.

```
MajorityVoting: PROCEDURE(Values() FLOAT IDENT) RETURNS(FLOAT);
  DCL Precision  FLOAT INIT(0.01),
      CntClasses FIXED INIT(0),
      (RoundValues(N), ClassValues(N)) FLOAT,
      CntClassMembers(N) FIXED, Found BIT,
      (CntMaxMembers, IdxMaxMembers) FIXED;
  ! Clustering of input values.
  FOR i FROM 1 TO N REPEAT
    RoundValues(i) := ROUND(Values(i) / Precision) * Precision;
    Found := '0'B;
    FOR k FROM 1 TO CntClasses REPEAT
      IF RoundValues(i) EQ ClassValues(k) THEN
        ! Existing class gets new member.
```

Table 1 Types of watchdogs and voters. This table shows error recognition methods that can be applied before any fallback state of a fault tolerance method is triggered.

watchdogs: - heartbeat to monitor the system's availability - compatibility of actual with formal arguments [8] - checking data integrity with respect to contents and structure [8] - supervision of infinite loops or other unintended branching of the control flow, realisable by diverse conditions in branches [8] - monitoring of runtime expenditure of tasks and noticing anomalies like frequent interruptions [8]
absolute voters: - pre- and post-conditions - assertions - invariants of loops or objects
relative voters: - majority voting, i.e. at least $\lceil \frac{n}{2} \rceil$ units out of n must provide the same result - consensus voting, i.e. largest number of members with equal results [42] - combinations like arithmetic mean, weighted sum, median or fuzzy logic [42] - statistical prediction, e.g. Kalman filtering [18] - test against the inverse of a function [8], e.g. matrix inversion by $A \cdot A^{-1} = I$ [33] - checksums and parity bits, e.g. checksums for matrix multiplication [33]

```
        CntClassMembers(k) := CntClassMembers(k) + 1;
        Found := '1'B;
        EXIT;
      FIN;
    END;
    IF Found EQ '0'B THEN
      ! A new class has to be created.
      CntClasses := CntClasses + 1;
      ClassValues(CntClasses) := RoundValues(i);
      CntClassMembers(CntClasses) := 1;
    FIN;
  END;
  ! Find class with max member count.
  CntMaxMembers := -1;
  IdxMaxMembers := -1;
  FOR i FROM 1 TO CntClasses REPEAT
    IF CntClassMembers(i) > CntMaxMembers THEN
      CntMaxMembers := CntClassMembers(i);
      IdxMaxMembers := i;
    FIN;
  END;
  IF CntMaxMembers > N // 2 THEN
    RETURN(Result);
  ELSE
    RETURN(ERROR_FLOAT);
  FIN;
END;
```

3.2 Error Treatment

3.2.1 Redundancy

Redundancy means deploying more resources than necessary [38]. Redundant units increase functional safety, because automation systems with n redundant and independent resources are robust against failing of $n-1$ resources. Furthermore, failsafe hardware does not need an error detection unit, since it fails due to environmental effects. Software, however, is in need of additional error detection. If a software implementation produces an erroneous result, this is revealed by the results from the other redundant implementations. For this reason, software is only robust against

Table 2 Matrix explaining hierarchy of error treatment with software fault tolerance.

	redundancy		graceful degradation	
	homogeneous	diversity	homogeneous	diversity
functional diversity		✓		✓
load shedding				✓
milestone method			✓	
implementation diversity		✓		✓
backward recovery		✓		✓
temporal redundancy	✓			
forward recovery		✓		✓
timed forward recovery		✓		✓
timed data diversity			✓	
dynamic reconfiguration	✓			
Byzantine method	✓			

less than $n-1$ errors. Hierarchical redundancy refers to nesting different fault tolerance methods. Redundancy is further divided into temporal, analytic, static and dynamic methods [36, 23]. We explain these types later on.

3.2.2 Diversity

Plain redundancy refers to a homogeneous composition of multiple units, while diversity is a form of redundancy with each unit being different from the others. Diversity is differentiated into the types implementation, functional, physical, manufacturing diversity and diversity of operating conditions [22]. Physical and manufacturing diversity refer to hardware-based automation systems. Since we consider software safety, we will not target hardware topics in the following. The other diversity types are integrated into the presented fault tolerance methods later on. For software, only diverse implementations increase functional safety. Homogeneous implementations would contain the same errors by definition [23]. Exceptions to this rule are race conditions and transient hardware errors influencing registers used by a program.

3.2.3 Graceful Degradation

An automation system degrades gracefully if it provides reduced albeit specified automation behaviour in case of too many errors [40]. Here, a system can degrade either in functionality or availability [40]. Graceful degradation comes along with all fault tolerance methods for software aside from static redundancy.

3.3 Elementary Techniques for Error Treatment

After the error recognition step one or more of the following treatment methods are applied for fault tolerance. These elementary treatment methods can be categorised into homogeneous redundancy, diversity and graceful degradation as described above, see Table 2.

3.3.1 Analytical or Functional Diversity

The diverse components in a system using analytical redundancy or functional diversity, respectively, possess different specifications. They provide distinct although related functions. The relationship between the components allows either to restore a corrupted value with the help of the other

functions, or permits relative plausibility checks. One example is distance, velocity and acceleration [36] linked by the formula $a = \dot{v} = \ddot{s}$. Another one describes the dynamics of gas [36] inside a combustion chamber given by the formula $pV = nRT$, where T is the temperature, p the pressure, V the gas volume, n the amount of substance and R the gas constant. Advantages and drawbacks of analytical redundancy can be studied by the following implementation:

```
TYPE TFunction REF PROC RETURNS(FLOAT);
TYPE TRelation REF PROC (Values() INV FLOAT IDENT) RETURNS(FLOAT);

TYPE TRelatedFunctions STRUCT (/
  Count            FIXED,
  Functions(1:Max) TFunction,
  Results  (1:Max) FLOAT,
    /* Results = measured values retrieved by Functions */
  Relations(1:Max) TRelation,
  Restored (1:Max) FLOAT
    /* Restored = calculated values retrieved by Relations */
/);

Initialisation: PROC (RelFuns TRelatedFunctions IDENT) GLOBAL;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Results(i) := RelFuns.Functions(i);
  END;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Restored(i) := RelFuns.Relations(i)(RelFuns.Results);
  END;
END;

Plausibility: PROC (RelFuns TRelatedFunctions INV IDENT) RETURNS(BIT) GLOBAL;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    IF ABS(RelFuns.Results(i) - RelFuns.Restored(i)) GT Epsilon THEN
      RETURN(False);
    FIN;
  END;
  RETURN(True);
END;

Restoration: PROC(RelFuns TRelatedFunctions IDENT, Index FIXED) GLOBAL;
  RelFuns.Results(Index) := RelFuns.Relations(Index)(RelFuns.Results);
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Restored(i) := RelFuns.Relations(i)(RelFuns.Results);
  END;
END;
```

- The precision of plausibility checks depends on the choice of the acceptance gap between the measured input values and the values calculated from the given formulas [33], see `Epsilon` in `Plausibility`.
- Plausibility checks between all participating diverse functions' results do not reveal which result is corrupt.
- This fault tolerance type is not suited as library implementation due to the following drawbacks:
- The complexity of this method is not situated in the library procedures, but in the implementation of the diverse functions.
- Additionally, the library procedures increase the program's complexity, since the linkage of the measured input values to arguments of the relationship functions has to be realised by a generic array `TRelatedFunctions.Results` in `Restoration` with no relation to the original variables' nomenclature.
- The restore-functions' signatures depend on the type of relationship, e.g. direct calculation in the case of gas dynamics vs. multiple inputs for derivatives in the case of movements.

In the next example we show how to use the library implementation from above in order to realise the gas dynamics formula. Line 3 shows the problem of mapping the application-related variables `(p, V, n, T)` to the formal parameters of the library `Values(1:Max)`, which is only viable by error-prone pointers (`REF` in PEARL):

```
DCL Max FIXED INIT(4);
DCL Values(1:Max) FLOAT;
DCL (p, V, n, T) REF FLOAT INIT(Value(1), Value(2), Value(3), Value(4));
DCL Funs() TFunction INIT(Piezo.Read, Flow.Read, GasScale.Read, Thermo.Read);
SPC (Pressure, Volume, Substance, Temperature) TRelation;
DCL Relations() TRelation INIT(Pressure, Volume, Substance, Temperature);
DCL GasChamberFuns TRelatedFunctions INIT(Max, Functions, Values, Relations);

Pressure: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN(n * R * T / V);
END;

Volume: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN(n * R * T / p);
END;

Substance: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN((R * T) / (p * V));
END;

Temperature: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN((n * R) / (p * V));
END;

GasChamberControl: TASK MAIN;
  REPEAT
    AFTER 5 MSEC RESUME;
    Initialisation;
    IF NOT Plausibility(GasChamberFuns) THEN
      ...
      Restoration(GasChamberFunctions, ErrIdx);
      ...
    FIN;
  END;
END;
```

3.3.2 N-Version Programming

With N-version programming, a software part addressing a certain problem provides at least two solution methods with the same in- and output interface. These solutions can be diverse implementations or further enrichment of a result. A solution may be fortified with information from a task that may also be skipped, or with information from a task that provides more precise results the longer it is executed. N-version programming is divided into three sub-methods, namely sieve method, milestone method and implementation diversity [29].

3.3.3 Load Shedding or Sieve Method

If a hazardous or unanticipated situation occurs, an embedded computing system might react by scheduling more tasks than in normal mode. The higher the number of unanticipated requests is, the higher the number of tasks to be executed will be. In such situations, not all of these tasks can meet their deadlines. A fault tolerance method handling such cases is load shedding or the sieve method. There are two variants, i.e. skipping all tasks of minor importance and extending the periods [34]. Properties are:

- In contrast to hardware redundancy, load shedding does not underutilise processors when no overloads occur, at the expense of losing minor functionality [34].
- The difference to monotone tasks is, that sieve methods shall be either completed or skipped entirely, because there is no benefit to partly execute them [29].
- A programmer shall be able to group sieve methods, since for some applications, it is useless or even flawed to execute certain tasks if others were skipped.
- Load shedding is interwoven with the runtime system, since the applied scheduling strategy must allow to detect transient overloads before important tasks miss their deadlines. Earliest

deadline first (EDF) is applicable as scheduling strategy. Moreover, worst case execution times (WCETs) must be determined by a compiler, as explained in the following paragraph.
- Load shedding needs a mechanism for stating which tasks to skip primarily. On one hand, this can be done by priorities as in the next example, on the other hand, by modes as in the example thereafter.

If load shedding is to be applied, worst case execution times for each affected task should be provided by the compiler in the declaration part. A compiler can determine WCETs statically by summing up known maximum runtimes for each assembler instruction, multiplying them for loops, and using the worst case branch on conditional program jumps [22]. This procedure nearly always yields too pessimistic WCETs [37]. Therefore, compilers can build upon one of the following three alternatives:
- restricting language constructs (e.g. prohibit unbounded loops) [22],
- simplifying processor architecture (e.g. omit pipelining and caches, only fixed point arithmetic) [8],
- assessing WCETs empirically with the help of very pessimistic cache trashers (guaranteeing maximum number of cache misses) [20].

The following PEARL specifications illustrate language constructs for load shedding in the context of space vehicles. `TIMING` and `LOADSHEDDING` are module parts like `SYSTEM` and `PROBLEM`. The timing part is a synopsis of the timing analysis of all tasks, where `WCET` states the worst case execution time determined by a compiler and `RESPONSE` states the available response duration determined by an engineer. The first example shows how to state priorities. In case of transient overloads, a runtime system has to remove all tasks beginning with those carrying the smallest priority value. It gradually removes all tasks with the next priority value until all remaining tasks can meet their deadlines.

```
TIMING;
  TelescopeAdjustment:
    WCET(5.1 MSEC) RESPONSE(10 MSEC);
  ...
LOADSHEDDING;
  PRIO(1): ! skip first when in emergency mode
    TelescopeAdjustment, AntennaAdjustment;
  PRIO(2): ! skip second when in emergency mode
    EngineFineControl, SolarCellsFineControl;
```

The aforementioned solution is based on two modes, namely normal and emergency mode [26], whereas the second example uses multiple modes and tasks assigned to them. The `TIMING` part equals the example from above, while the `LOADSHEDDING` part consists of several modes with several tasks, each of which is executed when the runtime system finds itself in the respective mode. `SCIENCEMODE` is the default and desired operating state. However, if no timely execution can be guaranteed, the runtime system switches into one of the other modes. In the worst case, `SAFEMODE` has to be executed. The mode is chosen as follows: Each time the schedule changes, i.e. a task is activated, suspended or terminated, the runtime slack of all active tasks is computed. If the slack is negative, the system has to switch into a lower mode. This is iterated until the slack is non-negative or `SAFEMODE` is reached. On positive slack the runtime system can change into a higher mode. In order to avoid toggling between two modes in both our examples, the following options are viable:
- the slack on the active mode must exceed a given threshold,
- after a given period of time, the system automatically switches into the next higher mode,
- the emergency mode is revoked by a human operator.

```
TIMING;
  MainEngineRoughControl:
    WCET(2.7 MSEC) RESPONSE(8 MSEC);
  ...
LOADSHEDDING;
  SCIENCEMODE:
    EngineFineControl, SolarCellsFineControl,
    TelescopeAdjustment, AntennaAdjustment;
  INTERMEDIATE:
    ...
  SAFEMODE:
    RollControl, EngineRoughControl, ...;
```

3.3.4 Monotone Tasks or Milestone Method

Monotone tasks produce results with higher quality the longer they are running. If such tasks are terminated before final completion, they return the most recent valid result. Each intermediate result represents a milestone. A monotone task is decomposable into a mandatory part and a number of optional parts [29]. In the following, we evaluate three implementation types for the milestone method:

```
DCL Flag BIT;
ZeroByNewton:
  PROCEDURE((F, D) TFunction) RETURNS(FLOAT);
  DCL Xi       FLOAT      INIT(0.0);
  DCL MaxLoops INV FIXED  INIT(20);
  DCL Loops    FIXED      INIT(0);
  WHILE (NOT Flag) AND (Loops <= MaxLoops)
  REPEAT
    Xi := Xi - F(Xi)/D(Xi);
    Loops := Loops + 1;
  END;
  RETURN(Xi);
END;
```

```
DCL Result FLOAT;
ZeroByNewton: PROCEDURE((F, D) TFunction);
  DCL Xi FLOAT INIT(0.0);
  FOR i FROM 1 TO 20 REPEAT
    Xi := Xi - F(Xi)/D(Xi);
    Result := Xi;
    UPDATE;
  END;
END;
```

```
ZeroByNewton:
  PROCEDURE((F, D) TFunction) RETURNS(FLOAT);
  DCL Xi FLOAT INIT(0.0);
  MILESTONE(Xi);
  ON EARLYEND: BEGIN RETURN(Xi); END;
  FOR i FROM 1 TO 20 REPEAT
    Xi := Xi - F(Xi)/D(Xi);
  END;
  RETURN(Xi);
END;
```

- The first implementation uses a flag for each task that can be terminated beforehand. To terminate a task, its flag has to be set. This implementation suffers from interspersing the pure task functionality with milestone checks and the unbounded duration between two milestones.
- Another implementation type by [14] is to introduce a keyword `UPDATE`. At each such update point the scheduler can decide whether to terminate or to continue the task. The drawbacks are the same as in the implementation before. Moreover, the result variable must be non-local.
- Our third proposition is to forbid termination heteronomy. Instead, a `TERMINATE` instruction for a monotone task shall produce a signal `EARLYEND`, which must be caught by monotone tasks. A signal handler inside such tasks, then, releases all resources and returns the most recent result. With this method, monotone tasks can be terminated at arbitrary points in time. The WCET of the signal handler is clear to state and allows to bound the termination's duration in an intelligible way. This implementation type needs atomic sections, which are not interruptible by signals in order to avoid race conditions for assignments to the result variable. Therefore, all commands executed on variables stated in a `MILESTONE` list are compiled as atomic.

3.3.5 Implementation Diversity

Implementation diversity uses at least two alternatives that fulfil the same function, but with different designs or implementations [22]. With respect to software, one can vary architecture, algorithms, data representations on the one hand, and operating systems, runtime systems, compilers, programming languages, integrated development environments and test methods on the other [23]. Implementation diversity can be used dynamically or statically. Dynamic execution means that a scheduler decides at runtime which alternative to execute next. In static execution, it is known beforehand that all alternatives are executed and, afterwards, a voter receives all results to determine the correct one. With the static variant, the alternatives can be executed sequentially or in parallel [23]. The following three implementations show these FT types by conventional PEARL constructs, while the fourth example demonstrates new language constructs proposed by [22].

With respect to functional safety, one can vary the alternatives in simplicity or in runtime complexity. These categories are not exclusive. Simplicity increases the quality of an alternative, since less complex solutions contain less programming errors [36]. Using the runtime consideration, one alternative shall produce an exact result with long processing time, the other an imprecise result with short processing time. Before executing an alternative, the scheduler calculates the amount of time available and chooses the alternative with the highest result quality under the constraint of timeliness. Table 3 shows which execution types are reasonable to combine. Dynamic implementation diversity is related to recovery blocks, as we will explain further on.

Without using specialised PEARL syntax, the following three examples demonstrate how to realise the aforementioned three implementation diversity types. By contrast, the fourth example uses specialised syntax from [22]. Therein, `DIVERSE` introduces a block of alternatives, where each block starts with `ALTERNATIVE`. The keyword `ASSURE` signals the beginning of the plausibility check.

```
SequentialStaticRedundancy: PROC(Input STRUCT, Output STRUCT IDENT) RETURNS(BIT);
  DCL Results(1:N) STRUCT;
  /* execute all alternatives */
  Results(1) := Alternative1(Input);
  ...
  Results(N) := AlternativeN(Input);
  /* relative plausibility check */
  RETURN(VoterDecision(Results, Output));
END;
```

Table 3 Implementation diversity types: In dynamic implementation diversity, a scheduler decides at runtime which alternative to execute next, while static means that all alternatives are executed and the results are passed on to a voter. These alternatives can be executed sequentially or in parallel. During design of the alternative functions one can aim at optimising simplicity, runtime or other goals.

	dynamic sequential	static sequential	static parallel
simplicity	✗	✓	✓
runtime	✓	✗	✗
other	✓	✓	✓

```
DCL Input STRUCT;
DCL Results(1:N) STRUCT;
DCL Barrier(1:N) SEMAPHORE PRESET(1);

ParallelStaticRedundancy: PROCEDURE(X STRUCT, Y STRUCT IDENT) RETURNS(BIT);
  Input := X;
  /* execute all alternatives */
  ACTIVATE Alternative1;
  ...
  ACTIVATE AlternativeN;
  /* join with all alternatives */
  REQUEST Barrier(1);
  ...
  REQUEST Barrier(3);
  /* relative plausibility check */
  RETURN(VoterDecision(Results, Y));
END;
```

```
DynamicRedundancy: PROCEDURE(X STRUCT, Y STRUCT IDENT) RETURNS(BIT);
  /* execute first alternative */
  Y := Alternative1(X);
  /* absolute plausibility check */
  IF PlausiCheck(X,Y) THEN RETURN('1'B); FIN;
  /* evtl. execute and check others */
  ...
  /* evtl. execute last alternative */
  Y := AlternativeN(X);
  /* absolute plausibility check */
  IF PlausiCheck(X,Y) THEN RETURN('1'B); FIN;
  /* return error if all tasks failed */
  RETURN('0'B);
END;
```

```
DIVERSE
  ALTERNATIVE
    Segments1 := RegionGrowing(Bitmap);
  ALTERNATIVE
    Segments2 := EnergyMinimisation(Bitmap);
  ASSURE
    IF Diff(Segments1, Segments2) < 0.2 THEN
      RETURN(Intersect(Segments1,Segments2));
    ELSE
      INDUCE Error;
FIN;
```

3.3.6 Recovery Blocks

Recovery blocks are a fault tolerance method, where the "blocks" represent diverse implementations executed dynamically in sequential order and "recovery" refers to variables that need to be kept in memory for roll-back in order to restore a stable program status if a block failed. The following subsections describe both variants, namely backward and forward recovery.

3.3.7 Backward Recovery

Backward recovery is employed as follows [35, 36]: At first, backward recovery uses a checkpoint, if one of the following alternative blocks changes input- or program-state-variables during execution. A checkpoint is a snapshot of at least all the variables that could be changed by one of the diverse alternatives in the recovery block. Second, a primary implementation representing the conventional algorithm is applied. Third, an absolute plausibility test is evaluated after each alternative has been processed. This test is called acceptance test. It can contain an alternative's own post-condition or the post-condition of all alternatives. If the acceptance test fails, the system is rolled back to the last checkpoint, i.e. all variables are restored, and the next alternative is executed. When passing the acceptance test, all other alternatives of the block are ignored. If all alternatives fail, a computation error has to be reported followed by a fall-back to a surrounding fault tolerance level. Further reasons to roll back and retry are internal computation errors like division by zero and time-outs. Backward recovery embodies the following properties:

– Building a checkpoint consumes runtime and memory even if no error occurs [39]. For a supercomputer, creating a checkpoint file on a disc is reported to consume up to 25 min [11].

– Rolling back to a checkpoint takes runtime, but only in case of errors [39].

– Restoring a checkpoint takes time as well [11].

± If preventive checkpoints are not possible due to runtime overhead, periodic checkpointing can be applied [11].

± The post-conditions must be simple and ideally proven correct in order to prevent introducing further design errors [22].

± For checkpointing, it would be beneficial to have a primitive at hand, that closely packs all checkpoint variables without padding in order to transfer a single memory block with one instruction. The packing becomes an optimisation problem if different checkpoints are necessary.

```
Segmentation: PROCEDURE RETURNS(TSegments);
  DCL (PreSegments, Segments) TSegments;
  PreSegments := PreSegmentation;
  ! checkpoint
  Segments := PreSegments;
  ! primary implementation
  RegionGrowing(Bitmap, Segments);
  ! acceptance test
  IF Quality(Segments) < 0.5 THEN
    ! roll-back
    Segments := PreSegments;
    ! alternative
    EnergyMinimisation(Bitmap, Segments);
    ! acceptance test
    IF Quality(Segments) < 0.5 THEN
      ! fall-back
      INDUCE Error;
    END;
  FIN;
  RETURN(Segments);
END;
```

If only one alternative is executed multiple times, backward recovery degrades to temporal redundancy. It is not reasonable to execute the same code twice due to the systematic nature of software errors. Hence, an error would be produced twice [22]. This is only useful in case of corrupt data, e.g. flipped bits, or race conditions.

3.3.8 Forward Recovery

Forward recovery is a fault tolerance method working in the same way as backward recovery with the sole difference that each alternative has its own pre-condition. The first alternative whose pre-condition is fulfilled is executed [21].

- The pre-conditions of forward recovery allow to skip alternatives if it is known beforehand that certain data would cause an alternative to fail.
- This fact saves runtime and decreases the hazard of executing program errors that could render the program unstable.
- Moreover, pre-conditions facilitate reading and understanding of the program text and contribute information for program verification.
- If an alternative triggers a peripheral process that irrevocably changes the program state, checkpointing is not possible, but forward recovery is [21].
- The diverse alternatives must employ different input interfaces in such a way that each subsequent pre-condition is not stronger than its predecessors' pre-conditions, i.e. weaker or not related. Otherwise, the program would contain unreachable code.
- The order of alternatives and their pre-conditions shall be checked during compilation.

```
EdgeDetection: PROCEDURE RETURNS(BIT);
  DECLARE Success BIT;
  CALL CopyImages;
  IF Weather.Bright AND Weather.Dry THEN
    Success := SobelOperator(VisualImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  IF Weather.Dark AND Weather.Dry THEN
    Success := SobelOperator(InfraredImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  IF Weather.DARK AND (Weather.Fog OR Weather.Rain) THEN
    Success := SobelOperator(RadarImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  RETURN(False);
END;
```

```
TYPE TVoid REF STRUCT (/ /);
TYPE TProcedure REF PROC;
TYPE TFunction REF PROC(TVoid) RETURNS(BIT);
TYPE TBlock REF PROC(TVoid) RETURNS(TVoid);

TYPE TAlternative STRUCT (/
  Precondition   TFunction,
  Implementation TBlock,
  Postcondition  TFunction
/);

ApplyForwardRecovery: PROCEDURE(
  Alternatives() INV TAlternative IDENT,
  Input TVoid) RETURNS(TVoid) GLOBAL;
  DCL (Checkpoint, Output) TVoid;
  DCL Next BIT INIT(True);
  Checkpoint := Input;
  FOR i FROM 1 TO UPB(Alternatives) REPEAT
    IF Alternatives(i).Precond(Input) THEN
      Output := Alternatives(i).Impl(Input);
      IF Alternatives(i).Postcond(Output)
        THEN RETURN(Output);
        ELSE Input := Checkpoint;
      FIN;
    FIN;
  END;
  RETURN(NIL);
END;
```

The preconditions can be restricted on response times, in order to meet deadlines in case of transient overloads. Here, a scheduler decides which alternative to execute depending on the response time the system must achieve and the WCET of the alternatives. The first example shows the syntax of [22], while the second one shows the syntax of [14].

```
EvasiveManeuvre: TASK RUNTIME SELECTABLE;
  BODY
    ALTERNATIVE WITH RUNTIME 2500 MSEC;
      CALL ParticleFilter;
      ! the best parameter estimation
      ! out of many simulations
    ALTERNATIVE WITH RUNTIME 25 MSEC;
      CALL SimulationOnlyWithAvr;
      ! improves based on average value
      ! with only one simulation
  FIN;
END;
```

```
BrakeControl CLASS [
  EvasiveManeuvre: PROCEDURE RETURNS(FLOAT) VIRTUAL;
    CALL ParticleFilter;
  END;
  EvasiveManeuvre:: ALTPROCEDURE;
    CALL SimulationOnlyWithAvr;
  END;
];
```

3.3.9 Time-constrained Data Diversity

Data diversity means that the same algorithm is executed with input data that is represented in different ways. Data diversity is split into three further types [5]. One type uses dynamic redundancy, where data is reformulated if the used implementation raises an error. Another type uses static redundancy, where data is reformulated n times and each version is processed by the same algorithm. Afterwards, a voter decides which output to return. Both methods circumvent errors due to unstable algorithms, but we recommend to use forward recovery instead and to use algorithms that are stable for a certain input space. In contrast, the third data diversity type can be employed to meet deadlines by resizing data to a smaller extent. For the implementation of a procedure using time-constrained data diversity, the procedure needs the remaining processing time and a worst case execution time with respect to a certain data size. Apart from that, an implementation is very clear to read.

```
ImageProcessing: PROCEDURE(Image TBitmap, RemainingTime DURATION);
  DCL WCETperPixel INV DURATION INIT(7 MSEC);
  Compress(Bitmap, Image.Width * Image.Height * WCETperPixel / RemainingTime);
  ...
END;
```

3.3.10 Dynamic Reconfiguration

After a watchdog has detected an error, it sets an error flag and, thereby, provokes the runtime system to shift certain task sets from one processor to another processor or node. Multiprocessor-PEARL realises dynamic reconfiguration by a `CONFIGURATION` part [1]. Its organisation is similar to High Integrity-PEARL, that uses `STATE`s with Boolean expressions and `LOAD...TO` as well as `REMOVE...FROM` keywords to shift task sets [22]. Dynamic reconfiguration creates a host of drawbacks:

– Dynamic reconfiguration suffers from hard-wiring of peripheral components to certain processors such that a shifted task cannot access certain resources [22].
– Some processors have to be reserved as spare units [11].
– Moreover, reconfiguration is a dynamic language construct. This implies that shifting a task set to another processor can cause unanticipated timing behaviour complicating the program schedule, provoking transient overload, or causing failures due to consuming an unexpected high amount of memory.

In order to avoid a system shut-down during reconfiguration, preventive migration together with error prediction can be applied [11]. Here, software execution continues, but processing speed slightly degrades during preventive migration. Preventive migration has to be combined with other fault tolerance methods if an error was not predicted [11].

3.3.11 Rejuvenation and Byzantine Method

Rejuvenation is restarting a system from a checkpoint [42]. The restart can be of the form rebooting, garbage collection, swapping space etc. [12]. It can be used for the whole system, a node or a task [12]. The Byzantine method uses rejuvenation for n systems that are restarted periodically, each at a different point in time [27]. After a spare unit restarts, it uses the state stored by the other units.

\+ The Byzantine method can bridge the time of switching to another processor [25].
\+ Rejuvenation can be used to discharge a processor from radiation [25].
\+ It avoids numerical error accumulation [12].
± The downtime for rejuvenation is planned and, therefore, less hazardous than an unplanned downtime due to an error [12].
± It is applicable when system resources are exhausted or data is corrupt [12]. In this case, rejuvenation is a crude method, because the error source is not detected.
– Program execution is interrupted during reboot [11].

4 Conclusions for PEARL-2020

After insights in multiple fault tolerance methods for software and their implementations, we select appropriate methods for highly safety-critical applications. Table 4 gives a synopsis of all methods presented, their objective, their linkage to IEC 61508-3 regulations and recommendations with respect to the four safety integrity levels (SILs). The higher the SIL is, the higher the integrity of an automation system has to be. The SIL is determined by questions considering extent and limitation of damage for human beings, environment and assets, probability of failure and duration of stay in a danger area. We marked which methods are applicable and recommend which implementation type to use in PEARL-2020, the new standard that shall substitute PEARL-90 and Multiprocessor-PEARL. Therefore, we can choose from four implementation types, namely hand-coded, code-snippets, library procedures and language primitives. For election, we consider the following constraints formulated with respect to language primitives. This leads us to the aim of devising language primitives only for FT methods that must be monitored by a runtime system.

\+ If a language provides more primitives than necessary, a programmer can choose the best-suited one, which improves readability of the source text.
– With too many primitives, programs become unintelligible if the primitives are mixed [28].

Table 4 Synopsis of the presented fault tolerance methods.

fault tolerance method	objective	IEC 61508	SIL 1	SIL 2	SIL 3	SIL 4	PEARL 2020	implementation
abs. plausibility checks	correctness	A.2.3a	+	+	+	++	✓	primitives
rel. plausibility checks	correctness	A.2.3b	±	+	+	±	✓	library
functional diversity	correctness	A.2.3e	±	±	+	++	✓	code-snippets
load shedding	timeliness	–					✓	primitives (states)
milestone method	timeliness	–					✓	primitives (update)
implementation diversity	correctness	A.2.3d	±	±	±	+	✓	code-snippets
backward recovery	correctness	A.2.3f	+	+	±	–	✗	–
temporal redundancy	correctness	A.2.4a	+	+	±	±	✗	–
forward recovery	correctness	–					✓	code-snippets
timed forward recovery	timeliness	–					✓	code-snippets
timed data diversity	timeliness	–					✓	hand-coded
dynamic reconfiguration	correctness	A.2.6	±	– –	– –	– –	✗	–
Byzantine method	correctness	–					✓	primitives

– Libraries are more appropriate than programming language constructs tailored to a certain area of application since the latter can become deprecated and cannot be exchanged easily if application areas develop further [16].
– With a minimum of language primitives, a language is easier to learn, better to understand and verify [22].
\+ With FT primitives, a compiler can choose whether to translate into sequential or parallel execution [22].
± Primitives as well as libraries allow to change program behaviour with only little adaptations to the source text, either by parametrisation of the compiler or by exchanging class and module names.

4.1 Individual Language Design Decisions

We suggest to introduce absolute plausibility checks by the primitives `PRECOND` for pre-conditions, `POSTCOND` for post-conditions and `ASSERT` for invariants within a procedure body. The explicit distinction of those three keywords addresses semi-automatic program verification for the pre- and post-conditions. Whether or not to check those assertions is to be parametrised in the compiler. Relative plausibility checks will be provided by a library implementation since the necessary checks, e.g. for majority voting, tend to be more complex and regulation A.2.8 from IEC 61508-3 demands to use verified software components.

We advocate for providing functional diversity and implementation diversity by use of code-snippets, because mapping the nomenclature of a program's area of application to arguments of an FT handler is misleading and IEC 61508-3 regulation B.1.5 advises against the use of pointers, refer to the example from Section 3.3.1. Benefits of code-snippets are that they guide programmers to best-practice and allow to easily switch from sequential to parallel execution or vice versa.

Regarding load shedding, two design decisions have to be made: First, whether to provide two states (normal and emergency) and assign a priority to every task group or to provide multiple user-defined states with associated tasks, second, how to indicate state transitions to the runtime system. We also point out that we neglect regulation A.2.3g from IEC 61508-3 which recommends a stateless program design. We prefer to define multiple states in the program system part since they

allow to discard but also re-enable tasks if a transient overload becomes more critical. Considering the second design choice there are two possible solutions as well: define the current state only by using the current processor load or let the transitions be initiated by the programmer. Both options bear the risk of toggling between two or more states. We prefer the second option since it allows to query diverse conditions. The conditions have to be stated within the `LOADSHEDDING` part of a PEARL module due to readability.

We base the milestone method on the language primitive `UPDATE` as proposed by [14], but with a single change: instead of using a global return variable we use PEARL's conventional function header syntax with the new attribute `MONOTONE` that entails the name of the return variable. With the example from Section 3.3.4 in mind, this means `ZeroNewton: PROC((F, D) TFunction) RETURNS(FLOAT) MONOTONE(Xi);` The hand-coded variant with flags is inappropriate, since abort conditions and setting the flag can be misimplemented by a programmer. The termination heteronomy variant is insufficient to handle semaphore operations in case of early function exits.

We decided to drop language constructs for backward recovery and temporal redundancy, as they are not recommended by IEC 61508-3 for higher SILs. Furthermore, backward recovery should be substituted by forward recovery and temporal redundancy only aims at race conditions that should be eliminated beforehand. Dynamic reconfiguration is left out as well, because IEC 61508-3 strongly advises against it due to its dynamic nature as stated in regulation B.1.2. In order to prevent the disadvantages that come with dynamic reconfiguration we support the use of the Byzantine method.

Forward recovery should be supported with code-snippets that entail sequential `IF`s for pre-conditions and nested `IF`s for post-conditions. Language primitives are ruled out, because they unnecessarily enlarge the set of keywords. Library procedures are excluded, since they imply the need for parametric polymorphism or inheritance. The code-snippet for time-constrained forward recovery extends the forward recovery code-snippet by one further input argument for the response duration. Time-constrained data diversity, on the other hand, should be hand-coded, because data can be simply and readably resized before the actual function implementation.

The rejuvenation of the Byzantine method has two aspects: what to rejuvenate and how. Since subsystems are addressed by other FT methods, we restrict it to the whole embedded system. Therefore, the time period and offset for rejuvenation of certain processors can be stated within the program's `ARCHITECTURE` part. We allow only restarting, since it enables radiation discharging without complicating the language with other options. PEARL is a real-time programming language, hence, memory operations, e.g. garbage collection, are forbidden. In principle, compiler hints for swapping would be allowed, but we prefer not to intervene into these processes. Finally, there is a need for a hand-coded initialisation procedure based on program states of the other Byzantine processors.

4.2 Criteria of Effectiveness

We conclude our paper with several measures on how to obtain quantitative results to demonstrate the effectiveness of the proposed approach, see Table 5. The coverage of IEC regulations can be derived from column *IEC 61508* of Table 4 for fault tolerance methods. For general language constructs, PEARL-2020 is as well-suited as other safety-related languages like MISRA-C, but no other language provides such powerful fault-tolerance language constructs. Thus, PEARL-2020 has a higher coverage than state-of-the-art languages. The IEC regulations enforce functional safety. The programmers, hence, are relieved from checking them explicitly and may concentrate on program validation and verification. Likewise, official certification authorities require less effort. Another criterion of effectiveness would be to compare accident statistics from programs developed with PEARL-2020 and other safety-related languages. We want to address this issue in future work.

Table 5 Criteria of effectiveness.

	main goal: functional safety
↗	coverage of IEC regulations (see column *IEC 61508* of Table 4)
↗	seamless implementation of various functional safety concepts, programmers do not need to address them explicitly
↘	programming effort and more effortless verification
↘	examination effort for certification authorities [22]
?	accident statistics for machines and plants that are programmed with PEARL-2020 compared to other languages like MISRA-C (future work)
↗	maintainability and debuggability
	side constraints: resources
↗	length of source code
→	memory for application data
→	runtime

We would like to juxtapose general program metrics of PEARL-2020 to other languages in a qualitative fashion: The length of PEARL-2020 source code is expected to become longer as many diverse checks are demanded. Memory requirements are equivalent to that of other safety-related languages because, even today, all devices implement diverse memory in agreement with IEC regulations. Runtime should remain at the same level.

Acknowledgement. We thank Daniela Horn for thoroughly proofreading the manuscript and the anonymous reviewers for their invaluable comments.

References

1 DIN 66253 Part 3. *PEARL for Distributed Systems*. Beuth, 1989.

2 IEC 60848. *GRAFCET Specification Language for Sequential Function Charts*. IEC, 2013.

3 IEC 61508-3. *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. IEC, 2010.

4 DIN 66253-2. *PEARL-90*. Beuth, 1998.

5 Paul Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Computers*, 37(4):418–425, 1988. doi:10.1109/12.2185.

6 Sohel Anwar, editor. *Fault Tolerant Drive By Wire Systems: Impact on Vehicle Safety and Reliability*. Bentham, 2011. doi:10.2174/9781608053070112010l.

7 John Barnes. *High Integrity Ada – The SPARK Approach*. Addison-Wesley, 1997.

8 Juliane Benra and Wolfgang A. Halang, editors. *Software-Entwicklung für Echtzeitsysteme*. Springer, 2009. URL: http://www.springer.com/de/book/9783642015953.

9 William Bolton. *Mechatronics: Electronic Control Systems in Mechanical and Electrical Engineering*, volume 3. Prentice Hall, 2004.

10 Josef Börcsök. *Funktionale Sicherheit*. VDE, 4th edition, 2014. URL: https://www.vde-verlag.de/buecher/483590/funktionale-sicherheit.html.

11 Franck Cappello, Henri Casanova, and Yves Robert. Checkpointing vs. migration for post-petascale supercomputers. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 168–177. IEEE Computer Society, 2010. doi:10.1109/ICPP.2010.26.

12 Vittorio Castelli, Richard E. Harper, Philip Heidelberger, Steven W. Hunter, Kishor S. Trivedi, Kalyanaraman Vaidyanathan, and William P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001. doi:10.1147/rd.452.0311.

13 Henan Chen, Yongduan Song, and Danyong Li. Fault-tolerant tracking control of fw-steering autonomous vehicles. In *2011 Chinese Control and Decision Conference (CCDC)*, pages 92–97, May 2011. doi:10.1109/CCDC.2011.5968152.

14 Matjaž Colnarič and Domen Verber. Dealing with tasking overload in object oriented real-time applications design. In *6th Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2001), 8-10 January 2001, Rome, Italy*, pages 214–222. IEEE Computer Society, 2001. doi:10.1109/WORDS.2001.945133.

15 Li DanYong and Song YongDuan. Adaptive fault-tolerant tracking control of 4ws4wd road vehicles: A fully model-independent solution. In *Chinese Control Conference (CCC)*, volume 31, pages 485–

492. IEEE, July 2012. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6389978.

16 Leberecht Frevert. Lösung von Echtzeitproblemen mit PEARL90-Objekten, 1998. URL: http://www.real-time.de/service/misc/GrundlagenOOP.pdf.

17 Kevin Fu. Trustworthy medical device software. *Public Health Effectiveness of the FDA 510(k) Clearance Process – Measuring Postmarket Performance and Other Selected Topics*, 2011. URL: http://www.nap.edu/read/13020/chapter/10.

18 Arthur Gelb, editor. *Applied Optimal Estimation*. MIT Press, 1974. URL: https://mitpress.mit.edu/books/applied-optimal-estimation.

19 GI-Working Group 4.4.2 "Real-Time Programming, PEARL". *PEARL 90 Language Report*, September 1998. Version 2.2. URL: http://www.real-time.de/service/misc/PEARL90-LanguageReport-V2.2-GI-1998-eng.pdf.

20 Julian Godesa and Robert Hilbrich. Framework für die empirische Bestimmung der Ausführungszeit auf Mehrkernprozessoren. In Wolfgang A. Halang, editor, *Funktionale Sicherheit, Echtzeit 2013, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V.(GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG), Boppard, 21. und 22. November 2013*, pages 77–86. Springer, 2013. doi:10.1007/978-3-642-41309-4_9.

21 Wolfgang A. Halang and Matjaž Colnarič. Dealing with exceptions in safety-related embedded systems. In *15th IFAC World Congress*, pages 983–988. Elsevier, 2002. doi:10.3182/20020721-6-ES-1901.00985.

22 Wolfgang A. Halang and Rudolf M. Konakovsky. *Sicherheitsgerichtete Echtzeitsysteme*. Springer, 2013. URL: http://www.springer.com/de/book/9783642372971.

23 Wolfgang A. Halang and Rudolf J. Lauber. *Echtzeitsysteme I*. FernUniversität Hagen, 2009.

24 Wolfgang A. Halang and Janusz Zalewski. Programming languages for use in safety-related applications. *Annual Reviews in Control*, 27(1):39–45, 2003. doi:10.1016/S1367-5788(03)00005-1.

25 F. Hubert. *Handbuch der Raumfahrttechnik*, volume 4, chapter Datenmanagement. Hanser, 2011.

26 Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.*, 12(9):890–904, 1986. doi:10.1109/TSE.1986.6313045.

27 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

28 Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.

29 Jane W. S. Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. Use of imprecise computation to enhance dependability of real-time systems. In Gary M. Koob and Clifford G. Lau, editors, *Foundations of Dependable Computing: Paradigms for Dependable Applications*, pages 157–182. Springer US, Boston, MA, 1994. doi:10.1007/978-0-585-27316-7_6.

30 Reinhard Maier, Günther Bauer, Georg Stöger, and Stefan Poledna. Time-triggered architecture: A consistent computing platform. *IEEE Micro*, 22(4):36–45, 2002. doi:10.1109/MM.2002.1028474.

31 Peter Marwedel. *Embedded Systems Design*. Springer, 2006. URL: http://www.springer.com/us/book/9789400702561.

32 Rainer Müller and Marcel Schaible. Die Programmierumgebung OpenPEARL90. In Wolfgang A. Halang and Herwig Unger, editors, *Industrie 4.0 und Echtzeit – Echtzeit 2014, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V.(GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG), Boppard, 20. und 21. November 2014*, Informatik Aktuell, pages 31–40. Springer, 2014. doi:10.1007/978-3-662-45109-0_4.

33 Paula Prata and João Gabriel Silva. Algorithm based fault tolerance versus result-checking for matrix computations. In *Digest of Papers: FTCS-29, 29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999*, pages 4–11. IEEE Computer Society, 1999. doi:10.1109/FTCS.1999.781028.

34 Parameswaran Ramanathan. Fault-tolerance in real-time control applications using *(m, k)*-firm guarantee. In *Digest of Papers: FTCS-27, 27th Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 132–141. IEEE Computer Society, 1997. doi:10.1109/FTCS.1997.614086.

35 B. Randell. System structure for software fault tolerance. *ACM SIGPLAN Notices – International Conference on Reliable Software*, 10(6):437–449, April 1975. doi:10.1145/390016.808467.

36 Charles Preston Shelton. *Scalable Graceful Degradation for Distributed Embedded Systems*. PhD thesis, Carnegie Mellon University, jun 2003. URL: https://users.ece.cmu.edu/~koopman/thesis/shelton.pdf.

37 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Akesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Systems*, 50(5-6):680–735, 2014. doi:10.1007/s11241-014-9204-x.

38 Jürgen J. Stoll. *Fehlertoleranz in verteilten Realzeitsystemen: Anwendungsorientierte Techniken*, volume 236 of *Informatik-Fachberichte*. Springer, 1990.

39 Dwight Sunada, David Glasco, and Michael J. Flynn. Multiprocessor architecture using an audit trail for fault tolerance. In *Digest of Papers: FTCS-29, 29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999*, pages 40–47. IEEE Computer Society, 1999. doi:10.1109/FTCS.1999.781032.

40 Matthias Tichy and Holger Giese. Extending Fault Tolerance Patterns by Visual Degradation Rules. In *2005 Workshop on Visual Modeling*

for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA, pages 67–74, September 2005. URL: http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2005/TG05.pdf.

41 Tjerk W. van der Schaaf and L. Kanse. *Human Error and System Design and Management*, chapter Errors and Error Recovery, pages 27–38. Number 253 in Lecture Notes in Control and Information Sciences. Springer, 2000. URL: http://www.springer.com/us/book/9781852332341.

42 Hongyu Sun Zaipeng Xie and Kewal Saluja. A survey of software fault tolerance techniques, 2006. URL: http://www.pld.ttu.ee/IAF0030/Paper_4.pdf.

Real-Time Scheduling on Uni- and Multiprocessors Based on Priority Promotions*

Risat Mahmud Pathan

Chalmers University of Technology
412 96, Göteborg, Sweden
http://orcid.org/0000-0002-9902-7558
risat@chalmers.se

Abstract

This paper addresses the problem of real-time scheduling of a set of sporadic tasks on uni- and multiprocessor platform based on priority promotion. A new preemptive scheduling algorithm, called Fixed-Priority with Priority Promotion (FPP), is proposed. In FPP scheduling, tasks are executed similar to traditional fixed-priority (FP) scheduling but the priority of some tasks are promoted at fixed time interval (called, promotion point) relative to the release time of each job. A policy called Increase Priority at Deadline Difference (IPDD) to compute the promotion points and promoted priorities for each task is proposed. FPP scheduling prioritizes jobs according to Earliest-Deadline-First (EDF) priority when all tasks' priorities follow IPDD policy.

It is known that managing (i.e., inserting and removing) jobs in the ready queue of traditional EDF scheduler is more complex than that of FP scheduler. To avoid such problem in FPP scheduling, a simple data structure and efficient operations to manage jobs in the ready queue are proposed. In addition, techniques for implementing priority promotions with and without the use of a hardware timer are proposed.

Finally, an effective scheme to reduce the average number of priority promotions is proposed: if a task set is not schedulable using traditional FP scheduling, then promotion points are assigned only to those tasks that need them to meet the deadlines; otherwise, tasks are assigned traditional fixed priorities without any priority promotion. Empirical investigation shows the effectiveness of the proposed scheme in reducing overhead on uniprocessor and in accepting larger number of task sets in comparison to that of using state-of-the-art global schedulability tests for multiprocessors.

2012 ACM Subject Classification Real-time systems, Process management, Scheduling, Embedded and cyber-physical systems

Keywords and phrases Real-Time Systems, Priority Promotion, Schedulability Analysis, Schedulability Condition

Digital Object Identifier 10.4230/LITES-v003-i001-a002

Received 2015-08-20 **Accepted** 2016-04-05 **Published** 2016-06-10

1 Introduction

The thirst to utilize increasingly more processing capacity of underlying hardware platform while meeting the deadlines of hard real-time sporadic tasks has resulted in the design of numerous scheduling algorithms. The preemptive dynamic-priority-based EDF scheduling is an optimal algorithm for uniprocessor: if there is an algorithm that can schedule a task set such that all the deadlines are met, then the task set is also schedulable using EDF scheduling [17]. In contrast, fixed-priority scheduling does not provide such a guarantee, even under the (optimal for uniprocessor) Deadline-Monotonic (DM) priority assignment [22].

* This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA and by the ARTEMIS Joint Undertaking under grant agreement no. 621429 for EMC2 project. This article is based on our earlier work in [24].

For uniprocessor, although EDF can better utilize the processing capacity, many practical systems implement FP scheduling due to its efficient run-time support and low overhead in managing the ready queue. For multiprocessors, there is no evidence whether global fixed-priority (G-FP) scheduling dominates or is dominated by the global earliest-deadline first (G-EDF) scheduling: some task set may only be deemed schedulable using the state-of-the-art G-EDF test while others only using G-FP test [15]. The question this paper addresses is: *Can we combine the schedulability and implementation benefits of both FP and EDF?*

In many real-time systems, e.g., avionics, spacecraft and automotive, it is important to efficiently use the processing resources due to size, weight and power constraints. Reducing overhead of task scheduling in such systems can cut cost for mass production of, for example, cars, trucks or aircraft. A theoretically "good" scheduling algorithm may not be used in practice if the overhead of implementation (e.g., managing tasks in the ready queue) is large. This paper proposes an unifying approach to integrate the schedulability and implementation benefits of both FP and EDF scheduling.

A new preemptive scheduling algorithm, called Fixed Priority with Priority Promotion (FPP), is proposed in this paper. Under FPP scheduling, each task has a fixed priority that may undergo priority promotion at fixed time intervals (called, *promotion points*) relative to the release time of each job. For example, consider task τ_i that has (initial) fixed priority p with two promotion points δ_1 and δ_2 at which the priority of the task is promoted to priority levels p_1 and p_2 such that $\delta_1 < \delta_2$ and $p_2 < p_1 < p$ (lower priority value implies higher fixed priority). If a job of task τ_i is released at time r_i, the priority of this job is p at time r_i and promoted to priority levels p_1 and p_2 at time $(r_i + \delta_1)$ and $(r_i + \delta_2)$, respectively. After a task's priority is promoted, its priority remains at this promoted priority until either (i) the task completes execution, or (ii) another promotion point is reached at which the task's priority is again promoted. As will be evident later, two or more jobs may have the same fixed priorities due to priority promotion. The FPP scheduler has a special tie-breaking policy in such case: a newly released job cannot preempt a currently-executing job if both of these jobs have the same priority. Other than priority promotion, FPP scheduling is same as traditional FP scheduling on uniprocessor and multiprocessors[1] platform while applicable to implicit-, constrained- or arbitrary-deadline sporadic tasks.

The FPP scheduler consists of a *dispatcher* and a *ready-queue manager*. The dispatcher at each time instant dispatches the highest-priority ready job if a processor is idle. If all the processors are busy, then a newly released job with higher priority can preempt a currently-executing relatively lower priority job. Active jobs that cannot be executed wait in the ready queue. The ready-queue manager inserts and removes jobs to and from the ready queue. The ready-queue manager also takes care of priority promotion of the jobs that are currently awaiting execution in the ready queue.

The effectiveness of FPP scheduling in meeting the deadlines of the tasks depends on the promotion points and promoted priorities of each task. A simple policy called Increase Priority at Deadline Difference (IPDD) to compute (offline) the promotion points and promoted priorities for each task is proposed. When all the tasks are assigned priorities based on IPDD policy, it will be shown that the FPP scheduling essentially prioritizes jobs of the tasks according to EDF priority. Recall that a job with shorter absolute deadline has smaller priority in EDF scheduling. We say that job J_a has higher EDF priority than job J_b if the absolute deadline of J_a is shorter than that of job J_b. Executing jobs of the tasks in EDF order but using priority-promotion-based FPP scheduler is one of the major contributions in this paper.

[1] In this paper, the term "FPP scheduling" in general applies to scheduling on uniprocessor and multiprocessors. For multiprocessors, FPP scheduling means global FP scheduling with priority promotion.

Since jobs can be prioritized in EDF order, the management of jobs in the ready queue of FPP scheduler would suffer from the same overhead problems (as discussed by Buttazzo [12]) if it is implemented similar to that of traditional EDF scheduler. On the other hand, the ready queue management and run-time support for traditional FP scheduling is much simpler, which is the main reason for its popularity in many commercial real-time kernels. This paper proposes a simple data structure and constant-time, i.e., $O(1)$ operations for implementing the ready queue. The ready queue management using the proposed scheme has similar benefits as that of traditional FP scheduler, which is another major contribution of this paper.

The only source of additional overhead for managing the jobs in the ready queue of FPP scheduler in comparison to that of FP scheduler is the cost of priority promotion. To reduce such overhead due to priority promotion, a joint priority assignment and schedulability test, called FPP_Test, is proposed for FPP scheduling. The FPP_Test assigns traditional fixed priorities (with no promotion point) to some tasks while assigns priorities to other tasks (with promotion points) using IPDD policy. Such priority assignment is effective for task set that is neither schedulable using pure FP scheduling nor using pure EDF scheduling. This result is very important for scheduling on multiprocessors since neither FP nor EDF scheduling is optimal for multiprocessor scheduling which is in contrast to scheduling on uniprocessor for which EDF is the optimal algorithm. The FPP_Test thus combines the schedulability benefits of both fixed and dynamic (i.e., IPDD) priorities in addition to having the similar implementation benefits of traditional FP scheduler.

To measure the effectiveness of FPP scheduling in terms of reducing overhead for managing jobs in the ready queue in comparison to that of EDF scheduling, the execution of randomly generated task sets is simulated using both FPP and EDF scheduling. The ready queues are simulated using the proposed data structure (presented in Subsection 4.2.1) for FPP scheduler and using a priority queue implemented as a binary min-heap (as is used in [10]) for EDF scheduler. The simulation result shows that ready queue management of FPP scheduler suffers significantly less overhead in comparison to that of EDF scheduler.

The FPP_Test is applicable to both uniprocessor and multiprocessor platform. On uniprocessor platform, any task set schedulable using the optimal preemptive EDF scheduling is also schedulable using FPP scheduling. Thus, FPP is also optimal for uniprocessor. On multiprocessor platform, it will be shown that the FPP_Test dominates both the state-of-the-art G-FP and G-EDF tests. Simulation result shows the effectiveness of FPP scheduling in determining higher percentage of schedulable task sets and in reducing the number of preemptions and migrations.

Finally, techniques to implement priority promotion with and without using hardware timer are proposed. We tackle the challenge of using one hardware timer to implement multiple priority promotions that are due at some later time. In addition, we also address the problem of implementing priority promotions without using a timer (i.e., based on pure software approach). In such software-based approach, a technique called *delayed promotion* is used: some jobs' priority promotions are delayed until it is necessary to ensure specific property of the underlying schedule.

Related Work. The FPP algorithm is similar to the well-known *dual-priority* scheduling which was first proposed by Burns and Wellings [11] in 1993, and analyzed by Davis and Wellings [13, 16] considering shared resources, release jitter and for scheduling soft real-time tasks. In dual-priority scheduling, each task undergoes priority promotion only once. In contrast, a task in FPP scheduling may have more than one promotion point. The reason for having more than one promotion point is to have the power to prioritize jobs in EDF order to meet deadlines.

Gonzalez Harbour et al. [18] considered scheduling FP scheduling of periodic tasks where each task is divided into a collection of precedence-constrained subtasks such that each subtask has its own priority. It is shown using an example by Burns and Wellings [11] that a task set

may be schedulable in dual-priority scheduling and may not be schedulable using the approach proposed by Gonzalez Harbour et al. [18]. After around one-and-half decade, Burns [9] presented an open problem at the RTSOPS seminar in ECRTS 2010: Is the utilization bound of dual-priority scheduling of implicit-deadline tasks on uniprocessor 100%? While Burns [9] solved this problem for task set having $n = 2$ tasks, the answer to this question for $n > 2$ is still unknown for dual-priority scheduling. This paper will show that the utilization bound of FPP scheduling of implicit-deadline tasks on uniprocessor is 100% for any n.

Organization. The rest of the paper is organized as follows. Section 2 presents the task model. The IPDD policy and important lemmas of this policy are presented in Section 3. The dispatcher and ready queue manager of FPP scheduler are presented in Section 4. Technique to reduce the total number of promotion points, particularly, the FPP_Test is proposed in Section 5. The FPP_Test is applied to both uni- and multiprocessors considering constrained-deadline tasks and experimental results are presented in Section 6 and Section 7, respectively. Techniques to implement priority promotions for FPP scheduling are proposed in Section 8. Finally, Section 9 concludes this paper.

2 Task Model

This paper considers scheduling a collection of n sporadic tasks in set $\Gamma = \{\tau_1, \dots \tau_n\}$. Each task τ_i is characterized by a triple (C_i, D_i, T_i), where C_i represents the worst-case execution time (WCET), D_i is the relative deadline, and T_i is the minimum inter-arrival time of the jobs or instances of task τ_i. Successive arrivals of the instances (called *jobs*) of task τ_i are separated by at least T_i time units. Each job of task τ_i after its release requires at most C_i units of execution time before its relative deadline. The *release time* and *absolute deadline* of job J_a of task τ_i are respectively denoted by r_a and d_a such that $d_a = r_a + D_i$.

A job is called *active* if it is released but has not completed its execution. An active job may be *in execution* or *awaiting* execution in the ready queue at any time instant. The FPP scheduling is applicable to implicit-, constrained- and arbitrary-deadline tasks. This paper assumes that lower priority value implies higher priority levels; i.e., 1 and n are the highest and lowest priority levels, respectively.

3 Priority Promotion Policy: IPDD

The effectiveness of FPP scheduling depends on the promotion points and promoted priorities for each task. In this section, the IPDD priority-promotion policy that can prioritize jobs of the tasks in EDF order while executing using priority-promotion-based FPP scheduling is presented.

The IPDD priority-promotion policy requires n distinct fixed-priority levels to determine the promotion points and promoted priorities of n tasks. Tasks are indexed in deadline-monotonic order, i.e., if $j < i$ for any two tasks τ_j and τ_i, then $D_j \leq D_i$. Therefore, there are $(i-1)$ tasks (i.e., $\tau_1, \tau_2, \dots \tau_{i-1}$) that have their relative deadlines no larger than that of task τ_i. The IPDD policy computes the promotion points and promoted priorities for each task $\tau_i \in \Gamma$ as follows:

- Task τ_i has i different priority levels: starting from priority level i to the highest priority level 1. Task τ_i's initial priority i is promoted $(i-1)$ times. At each of the $(i-1)$ promotion points, the priority is promoted by one priority level. Figure 1 depicts IPDD priority-promotion policy for task τ_i.
- Each job of task τ_i when released has (initial) priority level i, which is promoted to priority level $(i-1)$ at the first promotion point; then promoted to priority level $(i-2)$ at the second

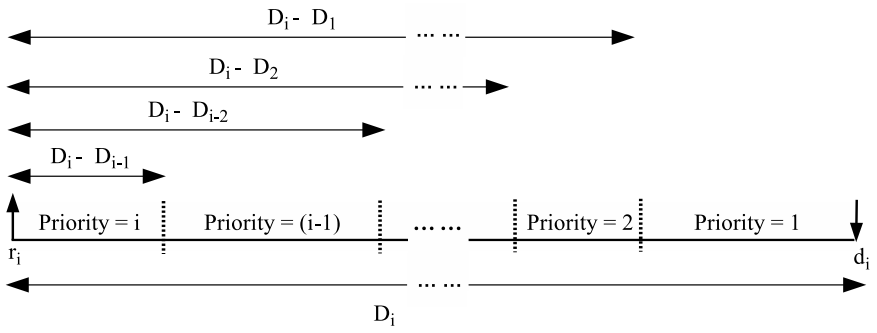


Figure 1 IPDD priority-promotion policy for task τ_i. Consider that an arbitrary job of task τ_i is released at time r_i and has deadline at $d_i = r_i + D_i$. The dotted vertical lines are the promotion points. The κ^{th} promotion points is $(D_i - D_{i-\kappa})$ time units later than time r_i for $\kappa = 1, 2 \ldots (i-1)$. Between two consecutive promotion points t_a and t_b, the priority of the job remains at the priority level set at the earlier promotion point t_a.

promotion point; and continuing in this manner, finally, promoted to the (highest) priority level 1 at the last, i.e., $(i-1)^{th}$ promotion point.

- The promotion points apply to each job of task τ_i. Each promotion point of a job is a fixed time interval from the release time of the job. The κ^{th} promotion point is equal to the (relative) deadline difference of tasks τ_i and $\tau_{i-\kappa}$, which is equal to $(D_i - D_{i-\kappa})$. The priority of each job of τ_i is promoted to priority level $(i-\kappa)$ at the κ^{th} promotion point which is $(D_i - D_{i-\kappa})$ time units later than its release time, for $\kappa = 1, 2, \ldots (i-1)$. Since $D_i \geq D_{i-1} \ldots \geq D_1$, we have $(D_i - D_{i-1}) \leq (D_i - D_{i-2}) \ldots \leq (D_i - D_1)$, which implies that priority of task τ_i is non-decreasing.
- If any two tasks τ_i and τ_j, where $i < j$, have the same relative deadline, then task τ_j's initial priority is i (not j) since $D_j - D_i = 0$ and the promotion points of τ_j are computed the same way that are computed for τ_i.

Each task τ_i has i priority levels $i, (i-1), \ldots 1$ with total $(i-1)$ promotion points. And, each task τ_j has j priority levels $j, (j-1), \ldots i, (i-1), \ldots 1$ with total $(j-1)$ promotion points where $i < j$. In other words, according to IPDD policy, i priority levels are common (shared) for any two tasks τ_i and τ_j whenever $i < j$. Due to sharing of priority levels, the number of *distinct* fixed-priority levels required to assign priorities to all the n tasks is at most n. As will be evident later, if two or more newly released jobs have the same initial priority, the FPP scheduler breaks the tie arbitrarily. If a newly released job has the same priority as that of a currently-executing job, the new job does not preempt the executing job in FPP scheduling. Example 1 demonstrates the IPDD policy using an example of three tasks.

▶ **Example 1.** Consider (C_i, D_i, T_i) for three tasks $\tau_1 \equiv (1, 2, 4), \tau_2 \equiv (4, 7, 8)$ and $\tau_3 \equiv (3, 10, 16)$. In IPDD policy, each job of task τ_1 starts with priority level 1 which is never promoted since there is no other task with smaller relative deadline than D_1. If a job of task τ_1 is released at time r_1, then the priority of this job remains at priority level 1 during $[r_1, r_1 + D_1)$.

Since there is one other task (i.e., τ_1) with smaller relative deadline than D_2, each job of τ_2 starts with priority level 2, which is promoted exactly once at time $r_2 + (D_2 - D_1) = r_2 + (7-2) = r_2 + 5$, where r_2 is the release time of an arbitrary job of τ_2. The priority of the job remains at priority level 2 and 1 respectively during $[r_2, r_2 + 5)$ and $[r_2 + 5, r_2 + D_2) = [r_2 + 5, r_2 + 7)$.

Since there are two other tasks (i.e., τ_1 and τ_2) with smaller relative deadlines than D_3, each job of τ_3 starts with priority level 3, which is promoted twice – first at time $r_3 + (D_3 - D_2) =$

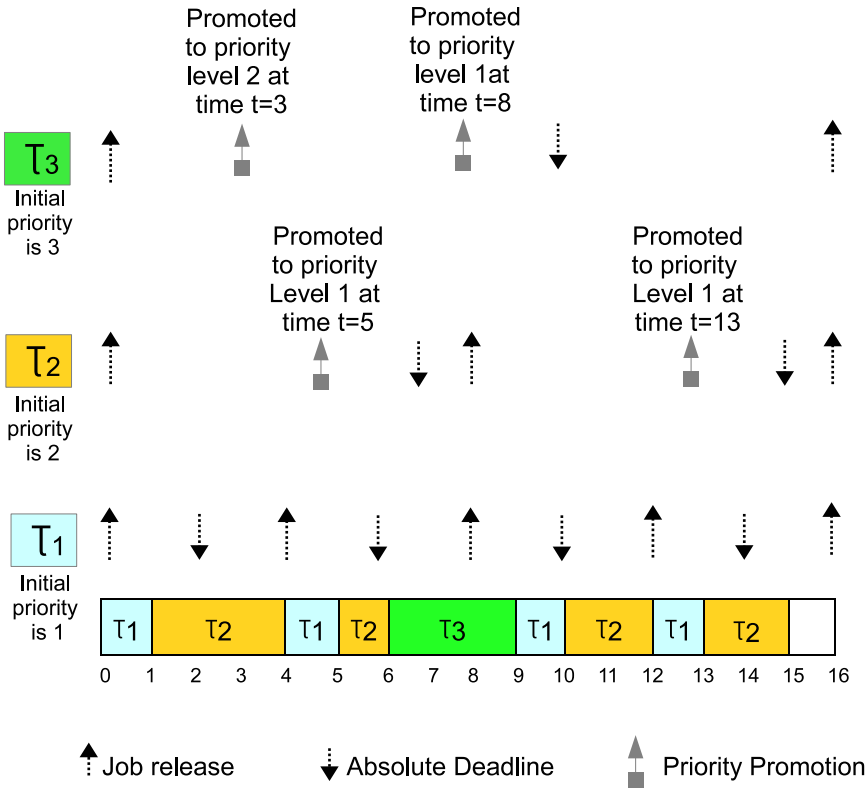


Figure 2 FPP schedule of three tasks in Example 1 where each task is assigned priorities using IPDD policy. Note that task τ_3's priority is promoted to priority level 1 at time $t = 8$ and is not preempted by the third job of task τ_1 that is released at time $t = 8$ although both jobs have the same priority. This is because a newly released job cannot preempt a currently executing job if both have the same priority in FPP scheduling.

$r_3 + (10 - 7) = r_3 + 3$ and second at time $r_3 + (D_3 - D_1) = r_3 + (10 - 2) = r_3 + 8$, where r_3 is the release time of an arbitrary job of τ_3. The priority of the job is at priority level 3, 2 and 1 respectively during $[r_3, r_3 + 3)$, $[r_3 + 3, r_3 + 8)$ and $[r_3 + 8, r_3 + D_3) = [r_3 + 8, r_3 + 10)$. The FPP schedule of this task set (assuming strictly periodic release for all tasks staring from time 0) is given in Figure 2 (the scheduler is formally presented in Subsection 4.1).

The remainder of this section presents important Lemmas regarding the properties of IPDD policy and will be used to show that FPP scheduling generates EDF schedule.

▶ **Lemma 2.** *If the priority of job J_a is promoted to priority level ℓ at time t_a, then $(d_a - t_a) = D_\ell$.*

Proof. Assume that job J_a is a job of task τ_i. Therefore, $d_a = r_a + D_i$. According to IPDD policy, $t_a = r_a + (D_i - D_\ell)$. Consequently, $(d_a - t_a) = D_\ell$. ◀

▶ **Lemma 3.** *If job J_a has higher priority than another job J_b at time t according to IPDD policy, then*

1. *the deadline d_a is smaller than the deadline d_b, and*
2. *J_a's priority never becomes smaller than that of J_b.*

Proof. Consider that J_a and J_b have priorities ν and ℓ at time t where $\nu < \ell$. We will show that (1) $d_a < d_b$, and (2) J_a's priority is never becomes smaller than that of J_b.

Since J_b's priority is ℓ at time t, its priority will ultimately be promoted to (higher) priority level ν according to IPDD policy. Let J_b's priority will be promoted to priority ν at time t_b where $t < t_b$. On the other hand, J_a's priority is already at priority ν at time t. Let J_a's priority be set to priority ν at time t_a where $t_a \leq t$. Therefore, $t_a < t_b$. From Lemma 2, it follows that $(d_a - t_a) = D_\nu$ and $(d_b - t_b) = D_\nu$. Since $t_a < t_b$, it follows that $d_a < d_b$ (part (1) is proved).

It follows from IPDD policy that the priorities of J_a and J_b are set to priority level κ, for $\kappa = \nu, (\nu - 1), \ldots 1$, respectively at time $(t_a + D_\nu - D_\kappa)$ and $(t_b + D_\nu - D_\kappa)$. Since $t_a < t_b$, we have $(t_a + D_\nu - D_\kappa) < (t_b + D_\nu - D_\kappa)$ for $\kappa = \nu, \ldots 1$. Therefore, priority of J_a is promoted to higher priority level earlier than that of J_b. Any job having priority κ remains at priority level κ for duration of $(D_\kappa - D_{\kappa-1})$ time units in IPDD policy. Therefore, J_a's priority is never smaller than that of J_b (part (2) is proved). ◀

▶ **Lemma 4.** *Consider that job J_a has priority ℓ at time t according to IPDD promotion policy. If a new job J_b of task τ_ℓ is released at time t, then the $d_a \leq d_b$.*

Proof. According to IPDD policy, job J_b of task τ_ℓ has priority ℓ at time t since it is released at time t. Since J_b is released at time t, we have $(d_b - t) = D_\ell$.

Job J_a's priority is already at priority ℓ at time t. Let J_a's priority be set to priority level ℓ at time t_a where $t_a \leq t$. From Lemma 2, we have $(d_a - t_a) = D_\ell$. Since $t_a \leq t$, we have $(d_a - t) \leq D_\ell$. From $(d_b - t) = D_\ell$ and $(d_a - t) \leq D_\ell$, its follows that $d_a \leq d_b$. ◀

▶ **Lemma 5.** *Consider set $\mathcal{J}$ of active jobs. If all the jobs in $\mathcal{J}$ have same priority ℓ at some time instant, then the job with the earliest deadline is promoted to priority level ν no later than that of any other job in $\mathcal{J}$, where $\nu < \ell$.*

Proof. Consider any two jobs J_a and J_b in $\mathcal{J}$. Without loss of generality assume that $d_a \leq d_b$. Let J_a and J_b are promoted to higher-priority level ν at time t_a and t_b, respectively. We will show that $t_a \leq t_b$, which implies that J_a with deadline no later than that of J_b is promoted to priority level ν no later than that of job J_b. It follows from Lemma 2 that $(d_a - t_a) = D_\nu$ and $(d_b - t_b) = D_\nu$. Since $d_a \leq d_b$, we have $t_a \leq t_b$. ◀

It will be shown based on Lemmas 2–5 that the FPP scheduling generates the EDF schedule of the tasks when priorities to all the tasks are given using IPDD policy. The dispatcher and the ready queue manager of FPP scheduler are presented in Section 4. Then Section 5 presents techniques to reduce the number of promotion points by not assigning priorities to all the tasks using IPDD policy (i.e., some tasks have no promotion point).

4 Dispatcher and Ready Queue Manager

The dispatcher of the FPP scheduler determines which active job to execute while the ready-queue manager is responsible for managing the ready jobs in the ready queue.

4.1 The Dispatcher

The dispatcher of FPP scheduler considering global multiprocessor scheduling on m identical processors is presented below. When $m = 1$, this dispatcher applies to uniprocessor. In addition, some important events related to the operations performed by the ready-queue manager are also highlighted below. The FPP dispatcher at each time t works as follows:

- At most m highest-priority jobs at time t are dispatched for execution. If t is the promotion point for a currently-executing job, then its priority is promoted[2] at time t.
- If all the m processors are busy and a new job J_{new} with priority *higher* than that of the currently-executing lowest-priority job J_{exe_low} is released at time t, then J_{new} starts execution by preempting J_{exe_low}. The preempted job J_{exe_low} is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the "`rel_prmt`" event.
- If all the m processors are busy and a new job J_{new} with priority *not* higher than that of the currently-executing lowest-priority job J_{exe_low} is released at time t, then J_{new} does not preempt J_{exe_low}. And, J_{new} is inserted in the ready queue. This (insertion) event managed by the ready-queue manager is called the "`rel_no_prmt`" event. Note that if J_{new} has the same priority as that of J_{exe_low} (ties in priority ordering), then J_{new} does not preempt J_{exe_low}.
- If some processor becomes idle while the ready queue is not empty, then the job having the highest priority from the ready queue is removed and dispatched for execution on the idle processor. The ready-queue manager performs this removal and this event is called the "`idle_remv`" event.

In summary, the `FPP` dispatcher works similar to traditional global `FP` scheduler with one additional feature: jobs may undergo priority promotion. If the total number of active jobs is not more than m, then all active jobs are in execution and the ready queue is empty. If the total number of active jobs is more than m, then all the processors are busy and some active jobs are in the ready queue. Example 1 presents the `FPP` schedule for three tasks. Theorem 6 proves that `FPP` scheduling executes jobs in `EDF` order if all tasks have priorities based on `IPDD`.

▶ **Theorem 6.** *If tasks are given priorities based on the `IPDD` policy, then the jobs of the tasks are executed in `EDF` order by the `FPP` scheduler at each time instant t.*

Proof. If the number of active jobs is no more than m, then each active job is executing at time t on separate processor. The claim of this theorem holds trivially. Now consider the case when the number of active jobs is exactly m at time t. If a new job J_{new} arrives at time t (i.e., number of active job becomes larger than m) such that the priority of J_{new} is not higher than that of the currently-executing lowest-priority job J_{exe_low}, then J_{new} is inserted in the ready queue. If J_{new}'s priority is smaller than that of J_{exe_low}, then from Lemma 3 it follows that the absolute deadline of job J_{new} is larger than that of job J_{exe_low}. If J_{new}'s priority is equal to J_{exe_low}, then it follows from Lemma 4 that the deadline of job J_{new} is not smaller than that of job J_{exe_low}. Similarly, since J_{exe_low} is the currently-executing lowest-priority job, its deadline is not smaller than any other currently-executing job. Therefore, job J_{new} with `EDF` priority not higher than any of the currently-executing job is inserted in the ready queue.

On the other hand, if J_{new} has higher priority than that of job J_{exe_low}, then J_{new} preempts the execution of J_{exe_low} and J_{exe_low} is inserted in the ready queue. According to Lemma 3, job J_{new} has earlier deadline than that of job J_{exe_low}. Therefore, job J_{exe_low} having a relatively lower `EDF` priority is inserted in the ready queue.

[2] For example, to perform the promotion, a special task can be designed whose only job is to promote the priority of the application tasks, as pointed by Burns [11] for dual-priority scheduling. In addition, all the priority promotions of a currently-executing job may be postponed until a new job is released. This is because the execution of a currently-executing job may be interfered only if a new job is released. When a new job is released at time t, the priority of the currently-executing job is determined considering the last priority promotion at or immediately before t. This can avoid unnecessary overhead due to priority promotion of the executing tasks. Section 8 will present different hardware and software-based techniques to implement priority promotion.

According to Lemma 3, if job J_a is prioritized by the dispatcher over another job J_b, then job J_b will never have higher priority (even if its priority might be promoted) than job J_a at another (future) time instant. Consequently, if job J_b is inserted in the ready queue because it cannot be prioritized by the dispatcher over another job J_a, then job J_b from the ready queue (even if its priority might be promoted) cannot preempt the execution of job J_a at some other (later) time. Therefore, no job that is inserted in the ready queue can preempt the jobs that are in execution.

Whenever some processor becomes idle at time t while the ready queue is not empty, the highest-priority job from the ready queue is removed and dispatched for execution. However, due to priority promotion, there may be multiple jobs waiting at the highest priority level in the ready queue. It will be shown in Subsection 4.2 that the ready-queue manager (when handling the `idle_remv` event) removes the job with shortest deadline (i.e., highest-priority `EDF` job) from the ready queue. Therefore, jobs are executed in `EDF` priority order in `FPP` scheduling in all cases. ◀

Now we concentrate on the ready queue manager, in particular, the events it has to manage. In addition to the `rel_prmt`, `rel_no_prmt` and `idle_remv` events, the ready queue manager needs to handle another event. If some job's priority is to be promoted while that job is awaiting execution in the ready queue, the ready-queue manager needs to manage this promotion. This event managed by the ready-queue manager is called "`pri_prom`" event. Therefore, the ready-queue manager needs to handle four different events: `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom`. If multiple events occur at the same time, they are managed by the ready-queue manager in any order.

A Note on Ready-Queue Manager for `FP` Scheduler. The ready-queue manager of traditional `FP` scheduling also needs to manage the `rel_prmt`, `rel_no_prmt` and `idle_remv` events. The ready-queue for `FP` scheduler can be implemented as an array of length n where task control blocks (TCBs) of the ready tasks are stored. The κ^{th} position of the array stores the TCB of the ready task that has current priority κ for $\kappa = 1, \ldots n$.

If `rel_prmt` or `rel_no_prmt` event occurs, then a job (particularly, J_{exe_low} or J_{new}) is inserted in the ready queue. If a job of task τ_κ is to be inserted in the ready-queue, then the priority of τ_κ is used to index the ready-queue array position at which the TCB of τ_κ is stored. Therefore, insertion is done in constant time.

If an `idle_remv` event occurs, then the highest-priority job from the ready queue is removed and dispatched for execution. Finding the highest-priority job from the ready-queue can be performed in constant time as follows. A bitmap array $B[n \ldots 1]$ of the ready-queue array is maintained. Initially, all the elements in bitmap B are zero to specify that there is no job awaiting execution at any priority level in the ready-queue array. When a job with priority κ is inserted in to the ready-queue array, the κ^{th} bit of the bitmap is set (i.e., $B[\kappa] = 1$) to specify that there is a job awaiting execution in the ready queue at priority level κ. Determining the highest-priority job from the ready queue is to find the position of the least set bit in the bitmap $B[n \ldots 1]$, which can be performed in constant time using, for example, deBruijn sequence [21], if not supported as a machine-level instruction [27]. Once the position is known, the TCB of the job is removed and corresponding job is dispatched and we set $B[\kappa] = 0$. An interesting discussion how the highest-priority task from the ready queue can be removed efficiently for `FP` scheduling can be found in [27].

In `FP` scheduling at most one element (i.e., job) is stored at each priority level in the ready queue. If the number of supported priority levels is smaller than the number of fixed-priority tasks, then the ready queue may be split into n different priority levels as is discussed by Buttazzo in [12]. In such case, the a FIFO queue is maintained at each priority level.

A Note on Ready-Queue Manager for EDF Scheduler. Implementing EDF requires to keep track of all absolute deadlines and perform a dynamic mapping between absolute deadlines and priorities. If the number of possible absolute deadlines for all the active tasks is larger than the the total number of distinct priority levels, managing the ready tasks is complex in EDF scheduling. If the number of active tasks is smaller than the number of different priority levels, updating the ready queue has higher overhead in comparison to that of FP scheduling, as is discussed by Buttazzo [12].

In the worst case, all the ready jobs may need to be remapped to new priority levels, which increases the overhead of ready queue management. The complexity and overhead in managing ready queue of EDF scheduler make it less popular in commercial kernel although EDF always performs better in terms of schedulability on uniprocessor when overheads are not considered.

Theorem 6 shows that the FPP scheduling executes jobs similar to EDF when tasks are given priorities using IPDD policy. However, managing jobs in the ready queue of FPP scheduler, if implemented similar to that of known for EDF, will have the same overhead problems that EDF suffers. In this paper, a new ready-queue management scheme for FPP scheduler is proposed. In particular, a data structure for the ready queue and constant-time operations to manage each of the `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` events is proposed.

In FPP scheduling, if the priority of a currently-executing job is promoted, then such promotion does not remap any job in the ready queue. However, if the priority of a job *residing* in the ready queue is promoted, then such promotion (as will be evident shortly) remaps that job to a new position in the ready queue data structure and thus incurs overhead.

Inspired by the discussion of Buttazzo [12], the *overhead model* this paper considers is the sum of total number of times each of the jobs in the ready queue is remapped to some other position. It will be empirically shown, based on this overhead model, that FPP scheduler has significantly lower overhead than that of EDF. Such low overhead of FPP scheduler shall make it popular in practice.

4.2 The Ready Queue Manager

This subsection presents the data structure of the ready queue and operations to handle the events `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom`.

4.2.1 Data-Structure for the Ready Queue

Due to priority promotion and sharing of priority levels in FPP scheduling, multiple active jobs may have the same priority at the same time instant. This is because two jobs of two different tasks τ_i and τ_j shares i priority levels where $i < j$. Therefore, the ready queue may need to store more than one job at the same priority level. An array of total n linked lists are used to implement the ready queue of the FPP scheduler. The κ^{th} linked list at any time instant stores all the TCBs of the ready jobs that have priority level κ at that time instant.

The κ^{th} linked list has two pointers: `head[`κ`]` and `tail[`κ`]` that respectively point the first and last TCB in the κ^{th} linked list. This ready queue data structure along with the bitmap is depicted in Figure 3. The purpose of the bitmap $B[n \dots 1]$ is to perform efficient searching to find the highest priority job from the ready queue.

4.2.2 Operations by the Ready Queue Manager

The ready-queue manager updates the ready queue whenever `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` event occurs. The jobs stored in the ready queue at any time instant will satisfy the following two properties:

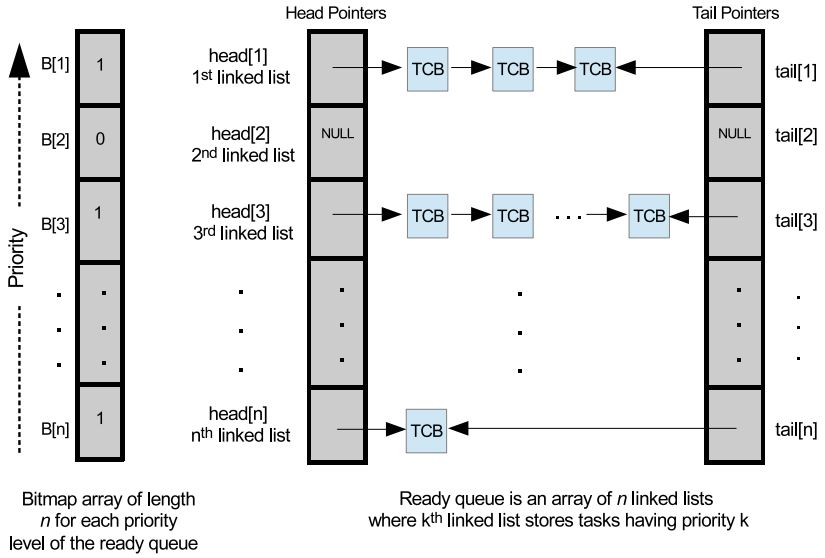


Figure 3 Proposed data structure of the ready queue for `FPP` scheduling.

- **P1**: All jobs stored in the ν^{th} linked list of the ready queue have higher `EDF` priorities than any other job stored in the ℓ^{th} linked list where $\nu < \ell$.
- **P2**: All jobs in the ℓ^{th} linked list are stored in order of non-increasing `EDF` priority, i.e., the `head[`ℓ`]` and `tail[`ℓ`]` respectively points the highest and lowest priority `EDF` job in the ℓ^{th} linked list for $\ell = 1, \ldots n$.

Assume that these two properties hold at time t_0 (such t_0 exists at least for the case when the system starts, i.e., when there is no job in the ready queue). Consider that some event (`rel_prmt`, `rel_no_prmt`, `idle_remv` or `pri_prom`) occurs at time t such that there is no other event after t_0 and before t. We will show that how properties P1 and P2 continue to hold after ready queue is updated to handle the event that occurs at time t. Maintaining properties P1 and P2 are very important to ensure that operations on the ready queue can be done in constant time. Given the structure of the ready-queue in Subsection 4.2.1, operations on the ready queue for managing events `rel_prmt`, `rel_no_prmt`, `idle_remv` and `pri_prom` are presented below.

Event `rel_prmt`. This event occurs if a newly released job J_{new} starts executing by preempting currently-executing lowest-priority job J_{exe_low}. The TCB of job J_{exe_low} is inserted in the ready queue. If priority of job J_{exe_low} is ℓ when preempted by J_{new}, then the TCB of job J_{exe_low} is *inserted at the front* of the ℓ^{th} linked list of the ready queue. The insertion at the front is done in constant time using the `head[`ℓ`]` pointer. We set $B[\ell] = 1$ to specify that there is a TCB awaiting execution at priority level ℓ.

Since job J_{exe_low} has priority ℓ at time t and was in execution, the $1^{st}, 2^{nd}, \ldots (\ell-1)^{th}$ linked lists of the ready queue at time t are empty. It follows from the proof of Theorem 6 that any job in the ready queue at time t neither has higher priority nor has earlier deadline than the currently-executing lowest-priority job J_{exe_low}. Therefore, inserting job J_{exe_low} at the front of the ℓ^{th} linked list at time t guarantees that P1 and P2 continues to hold.

Event `rel_no_prmt`. This event occurs if a newly released job J_{new} cannot preempt currently-executing lowest-priority job J_{exe_low} and J_{new} is inserted in the ready queue. If priority of J_{new}

is ℓ at time t, then the TCB of J_{new} is *inserted at the end* of the ℓ^{th} linked list. The insertion at the end is done in constant time using the `tail[`ℓ`]` pointer. We set $B[\ell] = 1$ to specify that there is a TCB awaiting execution at priority level ℓ.

Since P1 holds at time t_0 and job J_{new} has priority ℓ at time t, it follows that all the jobs in the κ^{th} linked list at time t have higher and lower `EDF` priorities than that of job J_{new} where $\kappa < \ell$ and $\kappa > \ell$, respectively. According to Lemma 4, all the jobs in the ℓ^{th} linked list at time t have their absolute deadlines no later than that of job J_{new} since job J_{new} is released at time t. Therefore, inserting J_{new} at the end of the ℓ^{th} linked list at time t guarantees that P1 and P2 continues to hold.

Event `idle_remv`. This event occurs when some processor becomes idle while the ready-queue is not empty. In such case, the highest-priority job from the ready queue is removed and dispatched for execution. The highest-priority job is in the *lowest-indexed non-empty* linked list. The lowest-indexed non-empty linked list (i.e., non-empty linked-list at the highest priority level) is found in constant time using bitmap B based on the same technique used to find the highest-priority ready task in traditional `FP` scheduling, for example, using deBruijn sequence [21].

Assume that the ℓ^{th} linked list of the ready queue is the lowest-indexed non-empty linked list. Note that there may be multiple jobs awaiting execution in the ℓ^{th} linked list. The job from the *front* of the ℓ^{th} linked list is *removed* and dispatched for execution. The removal from the front is done in constant time using `head[`ℓ`]` pointer. If `head[`ℓ`]` becomes NULL after this removal (i.e., the ℓ^{th} linked list becomes empty), then we set the $B[\ell] = 0$ to specify that there is no TCB awaiting execution at priority level ℓ.

Since property P2 holds at time t_0, the job from the *front* of the lowest-indexed non-empty linked list has the highest `EDF` priority at time t. And, after the removal of this job the remaining jobs in the ready queue also satisfy P1 and P2 since removal a job cannot violate P1 or P2.

Event `pri_prom`. This event occurs at time t when the priority of some job in the ready queue is to be promoted. If the priority of a job from the ℓ^{th} linked list is to be promoted to priority level ν, then this job is removed from the ℓ^{th} linked list and inserted to the ν^{th} linked list.

Since P2 holds at time t_0, the job at the front of the ℓ^{th} linked list has deadline no later than any other jobs in ℓ^{th} linked list. According to Lemma 5, given a set of jobs having the same priority ℓ at time t_0, the priority of the job with earliest deadline will be promoted to priority level ν no later than any other job in that set. Consequently, the priority of the job at the front of the ℓ^{th} linked list is to be promoted to priority level ν. Let job J_a is the job at the front of the ℓ^{th} linked list.

The TCB of job J_a is *removed from the front* of the ℓ^{th} linked list and *inserted at the end* of the ν linked list. The removal and insertion can be done in constant time using the `head[`ℓ`]` and `tail[`ν`]` pointers, respectively. Finally, if the ℓ^{th} linked list becomes empty after this removal, then we set the $B[\ell] = 0$. We set $B[\nu] = 1$ to specify that the ν^{th} linked list is now not empty.

Since the promoted priority of job J_a is ν at time t, all jobs in the κ^{th} linked list, where $\kappa > \nu$, have absolute deadline larger than that of job J_a at time t according to Lemma 3. Since P1 holds at time t_0 and job J_a is in the ℓ^{th} linked list at time t_0, it follows that all the jobs in the ν^{th} linked list have smaller absolute deadlines than that of job J_a. Therefore, inserting J_a at the end of the ν^{th} linked list ensures that P1 and P2 continues to hold.

In summary, when all the tasks are given priorities based on `IPDD` policy, the `FPP` scheduler executes jobs in `EDF` priority order. Therefore, existing `EDF` schedulability tests for uniprocessor and multiprocessors (i.e., `G-EDF` test) can be used to determine whether `FPP` scheduling can guarantee the schedulability of the tasks that are given priorities using `IPDD` policy.

If a currently-executing job's priority is promoted, then such promotion does not need to reorder the jobs in the ready queue (i.e., no ready queue management overhead is incurred). In contrast, if the priority of a job residing in the ready queue is promoted, then such promotion repositions the job to a higher-priority position in the ready queue data structure and thus incurs overhead. While the ready queue management of FPP scheduler has similar implementation benefits of FP scheduler (i.e., each event can be handled in constant time), the only source of additional overhead in comparison to FP scheduler is the cost of priority promotion. However, the number of promotion points can be reduced by assigning some tasks of a task set traditional fixed priorities with no promotion point. To this end, a technique to reduce the total number of promotion points and a new schedulability test called FPP_Test for FPP scheduling are proposed in next section.

5 FPP_Test to Reduce Number of Promotions

In this section, a schedulability test called FPP_Test to determine whether a task set is schedulable in FPP scheduling is proposed. The FPP_Test also determines the priorities of the tasks where a subset of the tasks is assigned traditional fixed priorities (without any priority promotion) while other tasks are assigned priorities (with priority promotion) based on the IPDD policy.

To determine which tasks can be assigned fixed priorities with no promotion point, an important feature of the state-of-the-art FP schedulability test is exploited. For uniprocessor and multiprocessor (global) FP scheduling, the corresponding state-of-the-art schedulability tests are of *iterative* nature: the schedulability of each task τ_i is tested separately. For example, the well-known response time analysis (RTA) for FP scheduling on uniprocessor [2] and multiprocessors [26] is of iterative nature: the response time R_i of each task τ_i is computed. The crucial observation is that when determining the schedulability of τ_i using an iterative FP schedulability test for uniprocessor [2] or for multiprocessors [25], the worst-case interference computation due to the higher-priority tasks in set $hp(i)$ does not assume that the jobs of the tasks in $hp(i)$ are also scheduled using FP scheduling; rather, it only assumes that jobs of the tasks in $hp(i)$ have higher priorities and cause maximum interference on τ_i. Consequently, Corollary 7 holds.

▶ **Corollary 7.** *If task τ_i is deemed to be schedulable at priority level ℓ using an iterative test where $hp(i)$ is assumed to be the set of higher priority tasks, then the schedulability of τ_i is preserved when it is assigned traditional fixed-priority level ℓ regardless whether the jobs of the tasks in $hp(i)$ are scheduled using dynamic or fixed priority.*

It follows from Corollary 1 that if some task τ_i is deemed schedulable using an iterative test at fixed-priority level ℓ, then task τ_i does not need to have any promotion point and the tasks in $hp(i)$ may be assigned fixed or IPDD (i.e., essentially dynamic) priorities in FPP scheduling. The FPP_Test is designed based on this observation.

The FPP_Test (presented in Figure 4) requires two schedulability tests to determine the schedulability of a task set in FPP scheduling. These two tests, denoted by T_{fp} and T_{edf} in Figure 4, are not "real" schedulability tests. Depending on the task model (e.g., implicit-, constrained- or arbitrary-deadline) and processor platform (uniprocessor or multiprocessors), we will plug in the state-of-the-art iterative FP test and EDF test respectively in place of T_{fp} and T_{edf}. In Sections 6–7, the actual tests used in place of T_{fp} and T_{edf} are presented respectively for uniprocessor and multiprocessor platform.

The FPP_Test in Figure 4 takes as input a task set Γ and returns "true" if the task set is deemed to be FPP schedulable; otherwise, it returns "false". The FPP_Test also determines the tasks that are given fixed priorities and the tasks that are given priorities based on IPDD policy.

Algorithm: `FPP_Test` (Task Set Γ)

```
1.  For priority level k = n to k = 1
2.    For each priority-unassigned task τ_i ∈ Γ
3.      If τ_i is schedulable at priority level k using
4.        test T_fp with all other priority-unassigned
5.        tasks assumed to have higher priorities
6.      Then
7.          assign τ_i to priority k
8.          break (continue outer loop)
9.      End If
10.   End For
11.   If all the priority-unassigned tasks pass T_edf, Then
12.     Compute promotion points only for the
13        priority-unassigned tasks using IPDD policy
        //Comment: IPDD policy uses (higher) priority levels
        // k, (k − 1) ... 1 for these priority-unassigned tasks
14.     Return True
15.   Else
16.     Return False
17.   End If
18. End For
19. Return True
```

Figure 4 Improved priority promotion policy for `FPP` scheduling.

Initially, all the tasks in set Γ are "priority-unassigned" in Figure 4, i.e., no task has any priority. Based on Audsley's OPA algorithm [1], the `FPP_Test` starts assigning traditional fixed priorities to the tasks starting from the lowest priority level. For each priority level k in line 1, some priority-unassigned task is searched using the inner loop in line 2-10 to assign it the fixed-priority level k. Whether or not a (priority-unassigned) task, say task τ_i, can be assigned priority level k is determined in line 3–5 by applying the iterative `FP` test T_{fp} and assuming higher priorities for all other (priority-unassigned) tasks. If such a task τ_i is found in line 3–5, then task τ_i is assigned the traditional fixed-priority level k in line 7 and the priority assignment for next (higher) priority level starts by jumping from line 8 to line 1. If the outer loop in line 1–18 terminates after assigning fixed priorities to all the tasks in Γ in line 7, then the algorithm returns "true" in line 19. And, `FPP` schedules all tasks similar to `FP` scheduling without any priority promotion.

If no task can be assigned the current priority level k (i.e., the test in line 3–5 is false for all the priority-unassigned tasks), then the inner loop in line 2–10 terminates. In such case, there exist some priority-unassigned tasks. The schedulability of all these priority-unassigned tasks are tested in line 11 by applying test T_{edf}. If these priority-unassigned tasks pass T_{edf}, then the promotion points *only* for these priority-unassigned tasks are computed in line 12-13 based on the `IPDD` policy and the algorithm returns "true" in line 14. Otherwise, the algorithm returns "false" in line 16. Note that the tasks assigned priorities using the `IPDD` policy in line 12-13 have higher priorities than any task that is given traditional fixed priority in line 7.

The `FPP_Test` guarantees schedulability of Γ using `FPP` scheduling if it returns "true". Assume that when the algorithm returns true, there are q tasks that are assigned traditional fixed priorities in line 7 and the remaining $(n - q)$ tasks are given priorities based on `IPDD` policy in line 12-13 for some q, $0 \leq q \leq n$. Each of the q tasks that is given traditional fixed priority in line 7 is schedulable in `FPP` scheduling based on Corollary 1. The schedulability of the $(n - q)$ tasks that are given `IPDD` priorities is not affected by the q tasks because these q tasks are given lower

(traditional) fixed priorities. Since the $(n-q)$ tasks, having priorities based on IPDD policy, are essentially scheduled in EDF order by the FPP scheduler (proved in Section 4), satisfying the T_{edf} test in line 11 guarantees that these $(n-q)$ tasks are also schedulable in FPP scheduling. If a task set is schedulable using traditional FP scheduling, then no promotion point is assigned to any task using the FPP_Test and all tasks are executed similar to traditional FP scheduling using FPP scheduler.

The ready queue management scheme of Subsection 4.2.2 still applies when priorities are assigned using FPP_Test. This is because the ready jobs of the q tasks having traditional fixed priorities are (i) stored in the linked lists corresponding to the q lower (i.e., $(n-q+1), \ldots n$) priority levels, (ii) never promoted to a higher priority level since they have no promotion point, and (iii) dispatched for execution only after all the jobs that are given the $(n-q)$ higher ($i.e., (n-q), \ldots 1$) priority levels are dispatched for execution. Properties P1 and P2 (defined in Section 4.2.2) do not necessarily need to hold for the tasks that are assigned traditional fixed priorities but always hold for the tasks assigned priorities using IPDD policy.

6 FPP_Test for Uniprocessor

In this section, the FPP_Test in Figure 4 is applied for determining schedulability of constrained-deadline tasks on uniprocessor. The response-time test for uniprocessor FP scheduling proposed by Audsley et al. [2] is considered in place of T_{fp} in Figure 4 to determine whether a (priority-unassigned) task τ_i can be assigned fixed priority level k. Note that this response-time test (which is an exact test) combined with Audsley's OPA algorithm in Figure 4 guarantees optimal fixed-priority assignment. And, the quick processor demand analysis (QPA), which is an exact EDF test, proposed by Zhang and Burns [30], is considered in place of T_{edf} in line 11 to determine whether all the priority-unassigned tasks are schedulable using EDF. The QPA test is an efficient implementation of the processor demand analysis proposed by Baruah et al. [4].

If a task set is EDF schedulable, then the QPA test in line 11 will also be satisfied when not all the tasks are assigned fixed priorities in line 7. Consequently, any task set that is schedulable using optimal EDF scheduling on uniprocessor also satisfies the FPP_Test. Since preemptive EDF is optimal [17], the FPP scheduling where priorities are assigned using the FPP_Test is also an optimal scheduling algorithm for constrained-deadline tasks on uniprocessor. For implicit-deadline task sets, the utilization bound of FPP algorithm is thus 100% because the utilization bound of EDF for such task system is 100% [23].

While the performance of FPP scheduling in terms optimality is same as EDF scheduling, it is not straightforward to see whether the overhead for managing jobs in the ready queue of FPP scheduler is lower or higher than that of EDF scheduler. Simulation using randomly generated task sets is conducted to measure such overhead.

Task Set Generation for Uni- and Multiprocessors. Each of the experiments is characterized by a pair (m, n) where m is the number of processors and n is the cardinality of a task set. For experiments on uniprocessor, we use $m = 1$. The UUnifast-Discard algorithm [14] is used to generate n utilization values of a task set. This algorithm takes as input the number of tasks n and total utilization U of the n tasks. And, it generates n utilizations $\{u_1, u_2, \ldots u_n\}$ of the n tasks such that the total utilization of these n tasks is U. Once a set of n utilizations $\{u_1, u_2, \ldots u_n\}$ of a task set is generated, the other parameters of each task τ_i in the task set are generated as follows:

- The minimum inter-arrival time T_i of each task τ_i is generated from the uniform random distribution within the range $[10ms, 1000ms]$.

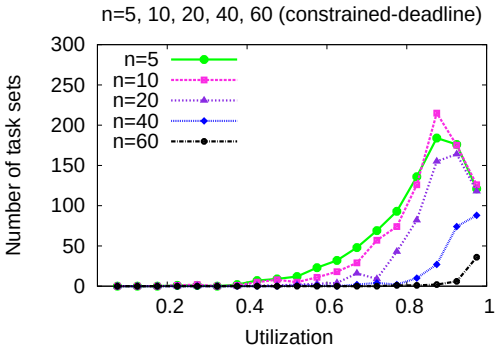


Figure 5 Number of task sets (out of 1000 task sets) that are schedulable using FPP/EDF scheduling but not schedulable using FP scheduling.

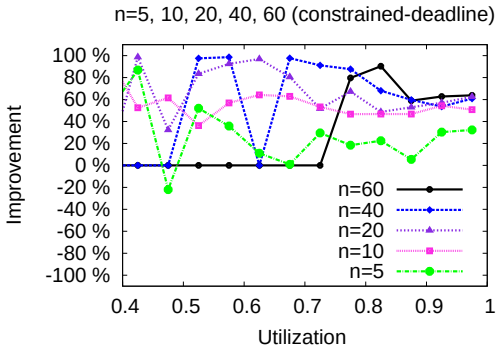


Figure 6 Improvement of FPP over EDF scheduling.

- The WCET of task τ_i is set to $C_i = u_i \cdot T_i$.
- The relative deadline D_i of task τ_i is generated from the uniform random distribution within the range $[C_i, T_i]$ for constrained deadline tasks; otherwise D_i is set to T_i for implicit-deadline tasks.

Task sets are randomly generated at 40 different utilization levels $\{0.025m, 0.05m, \ldots 0.975m, m\}$ for each experiment (m, n). A total of 1000 task sets at each of the 40 utilization levels are generated. Each of the 1000 task sets generated at a particular utilization level, say U, has cardinality n and total utilization equal to U.

Sources of Overhead. Insertion/removal of jobs to/from the ready queue of FPP scheduler (as discussed in Subsection 4.2.2) can be done in constant time. However, jobs that are in the FPP ready queue may need to change their position (i.e., upgraded to higher-priority linked list) due to pri_prom events. On the other hand, if the ready queue of EDF scheduler is implemented as binary min-heap [10] or binomial min-heap [8], then each insertion/removal of a job to/from the ready queue of EDF scheduler may need to reorder the remaining jobs in the ready queue in order to satisfy the *min-heap* property. Our objective is to compare such overhead in terms of total number of times different jobs in the ready queue change their position to handle pri_prom event (in FPP scheduling) and to maintain min-heap property (in EDF scheduling). The EDF ready queue is simulated using a binary min-heap where the ready job with the shortest absolute deadline is stored in the root. And, the FPP ready queue is simulated using the proposed data structure in Figure 3.

Experiments (Uniprocessor). The randomly-generated task sets that are (exclusively) schedulable using FPP/EDF (i.e., satisfy FPP_Test) and *not* schedulable using traditional FP scheduling are considered to compare overheads between FPP and EDF. Figure 5 presents the number of such task sets for each utilization level for different n.

For each such task set, the execution is simulated using both FPP and EDF scheduling. The ready jobs that need to await execution are stored in the corresponding ready queue and reordered when necessary. Since it is not computationally feasible to consider all possible release offsets and inter-arrival separations of sporadic tasks exhaustively in simulation, all release offsets are set to zero and all tasks are released periodically. The simulation is run for L time units where $L = min\{lcm(T_1, T_2, \ldots T_n), 10^8\}$ to avoid simulation for very large hyperperiod.

Overhead Metric. For each utilization level, the sum of total number of times each of the jobs of a task set change their position in the ready queue is computed and then the average over all task sets is determined for both FPP and EDF scheduling. $\texttt{EDF}_{av}$ and $\texttt{FPP}_{av}$ denote the average number of times the jobs of a task set change their positions in EDF and FPP ready queue, respectively. The improvement of managing jobs in the ready queue of FPP scheduler in comparison to EDF scheduler at each utilization level is:

$$\texttt{Improvement} = \frac{\texttt{EDF}_{av} - \texttt{FPP}_{av}}{max\{\texttt{EDF}_{av}, \texttt{FPP}_{av}\}} \times 100\% \,.$$

The value of Improvement ranges in $[-100\%, +100\%]$. For example, $\texttt{Improvement} = -50\%$ implies that the ready queue of EDF scheduler on average can reduce 50% overhead of managing jobs in the ready queue of FPP scheduler. And, $\texttt{Improvement} = +60\%$ implies that ready queue of FPP scheduler on average can reduce 60% overhead of managing jobs in the ready queue of EDF scheduler.

Empirical Results (Uniprocessor). The results of a series of simulations for different $n \in \{5, 10, 20, 40, 60\}$ are presented in Figure 6 where the x-axis is the utilization level U and the y-axis represents Improvement. The Improvement is non-negative in almost all the utilization levels[3]. The improvement of FPP scheduler over EDF scheduler is significant in most cases, i.e., FPP incurs noticeably less overhead (in terms of number of times jobs in the ready queue are remapped to new positions) than that of EDF.

The "positive" improvement of FPP is due to two main reasons. First, the proposed data structure for FPP ready queue enables a job to be inserted/removed to/from the ready queue in constant time without causing other existing jobs in the ready queue to change their position. In contrast, each insertion/deletion to/from the ready queue of EDF scheduler may cause multiple (i.e., $O(logn)$) jobs to change their positions to maintain the min-heap property. Second, the FPP_Test test is effective in reducing the number of promotion points. This is verified by observing (the outcome of FPP_Test on random task sets) that it is almost always the case where some tasks for the majority of the task sets are given traditional fixed priorities with no promotion point.

The reduction in promotion point at each higher utilization level for task set with larger cardinality is much higher than that of task set with smaller cardinality. For example, the average reduction in promotion points at $U = 0.8$ is around 80% and 40% for task sets with cardinality $n = 40$ and $n = 10$, respectively. This is because, when the number of tasks in a task set for a given utilization level is fewer, the utilization of individual task is relatively larger and the execution time of individual task tends to be larger. As execution time a individual task increases, jobs in the ready queue stay longer and may undergo a relatively larger number of priority promotions. And, more priority promotions cause higher number of times the jobs in the ready queue change their positions.

Observing the significant reduction in overhead, it is expected that if the ready queue of EDF scheduler is implemented using some other data structure, the benefit of proposed ready-queue management scheme for FPP scheduler will still be realized. To verify this, experiment using other data structure is needed and is left as a future work.

[3] The value of Improvement at relatively lower utilization levels (e.g., when $U < 0.4$) is caused by very few task sets (Figure 5 presents the number of such task sets) and such outliers can be ignored.

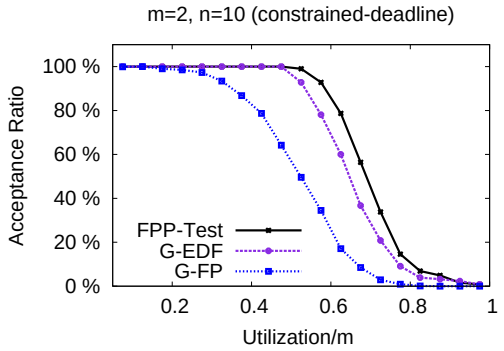


Figure 7 Acceptance ratio of `FPP_Test`, `G-FP` [26] and `G-EDF` [6].

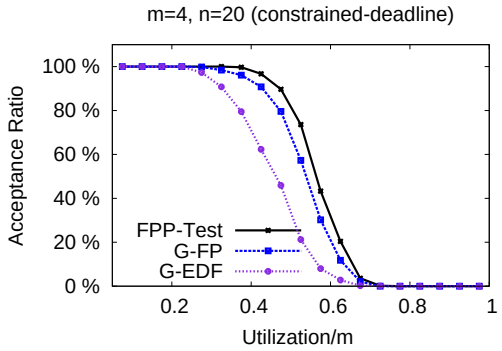


Figure 8 Acceptance ratio of `FPP_Test`, `G-FP` [26] and `G-EDF` [6].

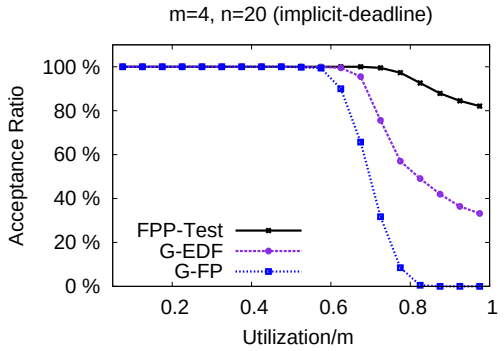


Figure 9 Acceptance ratio of `FPP_Test`, `G-FP` [26] and `G-EDF` [6].

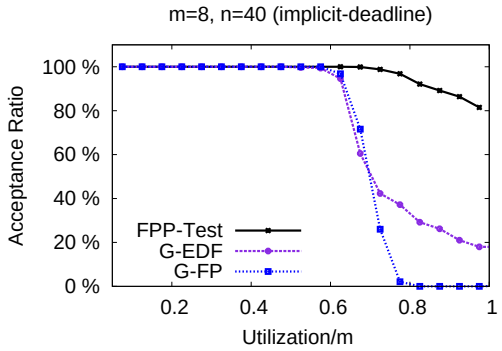


Figure 10 Acceptance ratio of `FPP_Test`, `G-FP` [26] and `G-EDF` [6].

7 `FPP_Test` for Multiprocessors

In this section, the `FPP_Test` in Figure 4 is applied to determine schedulability of constrained-deadline tasks scheduled on multiprocessors. For `G-FP` scheduling, Pathan and Jonsson [26] recently proposed an iterative `G-FP` test that is shown to perform better than any other iterative test proposed earlier. This `G-FP` test [26] is used in place of T_{fp} in Figure 4 to determine if a priority-unassigned task τ_i can be assigned fixed priority level k. For `G-EDF` scheduling, Bertogna and Baruah [6] proposed a step-by-step approach to apply different `G-EDF` schedulability tests proposed by other researchers. This `G-EDF` test in [6] is used in place of T_{edf} in line 11 of Figure 4 to determine if all the priority-unassigned tasks are schedulable on m processors using `G-EDF` scheduling.

There is no evidence regarding whether the `G-FP` test in [26] dominates or is dominated by the `G-EDF` test in [6]. It is not difficult to see that if a task set is deemed to be schedulable using `G-FP` test [26] or `G-EDF` test [6], then that task set also passes the `FPP_Test` for multiprocessors. In other words, the `FPP_Test` dominates the state-of-the-art `G-FP` and `G-EDF` tests. To measure the improvement of `FPP_Test` over `G-FP` test and `G-EDF` test, experiments using randomly generated task sets are conducted.

Empirical Results. For each experiment (m, n), random task sets are generated using the approach presented earlier.

The schedulability of each of the 1000 task sets generated at each utilization level is determined based on `FPP_Test`, `G-FP` test [26] and `G-EDF` test [6]. The acceptance ratio for each test at each utilization level is computed. The acceptance ratio of a schedulability test is the percentage of task sets deemed schedulable at a given utilization level.

A series of experiments for different (m, n), where $m \in \{2, 4, 8\}$ and $n \in \{3m, 5m, 10m\}$, are conducted. The result of two experiments with parameters $(m = 2, n = 10)$ and $(m = 4, n = 20)$ are presented in Figure 7 and Figure 8. The x-axis represents the system utilization U/m for utilization level U and the y-axis represents the acceptance ratio.

The performance of `G-EDF` test is better than `G-FP` test when $m = 2$ and $n = 10$ in Figure 7. This behavior is reversed in Figure 8. This shows neither `G-FP` nor `G-EDF` test empirically performs better than the other. The `FPP_Test` does not only theoretically dominate but also empirically performs better than both `G-FP` and `G-EDF` tests. The performance of `FPP_Test` test using implicit-deadline tasks is significantly better (see Figure 9 and Figure 10).

The difference in acceptance ratios among the tests are more pronounced at higher utilization level since task sets with large total utilization are difficult to schedule. The `FPP_Test` has the ability to accept higher percentage of task sets in comparison to that of `G-FP` and `G-EDF` tests by exploiting the benefits of both fixed and dynamic priority.

7.1 Preemptions and Migrations

To investigate whether `FPP` scheduling incurs higher or lower number of preemptions and migrations in comparison to `G-EDF`, simulations are conducted. Execution of randomly-generated task sets that are not `G-FP` schedulable but schedulable using *both* `FPP` and `G-EDF` are simulated for `FPP` and `G-EDF` scheduling. For each utilization level $U \in \{0.025m, 0.05m, \ldots m\}$, the average number of preemptions and migrations that a task set suffers is computed for both `FPP` and `G-EDF` scheduling.

$\texttt{GEDF}_{avpr}$ and $\texttt{FPP}_{avpr}$ denote the average number of preemptions that a task set suffers in `G-EDF` and `FPP` scheduling, respectively. Similarly, $\texttt{GEDF}_{avmg}$ and $\texttt{FPP}_{avmg}$ denote the average number of migrations that a task set suffers in `G-EDF` and `FPP` scheduling, respectively. The improvement by `FPP` in reducing preemptions and migrations in comparison to `G-EDF` at each utilization level is:

$$\texttt{Improvement(prmt)} = \frac{\texttt{GEDF}_{avpr} - \texttt{FPP}_{avpr}}{max\{\texttt{GEDF}_{avpr}, \texttt{FPP}_{avpr}\}} \times 100\%$$

$$\texttt{Improvement(migr)} = \frac{\texttt{GEDF}_{avmg} - \texttt{FPP}_{avmg}}{max\{\texttt{GEDF}_{avmg}, \texttt{FPP}_{avmg}\}} \times 100\%$$

The value of improvement ranges in $[-100\%, +100\%]$. For example, $\texttt{Improvement(prmt)} = +50\%$ implies that `FPP` on average can reduce 50% preemptions that occur in `G-EDF` scheduling. The results of simulations for different $n \in \{10, 20\}$ and $m \in \{2, 4\}$ are presented in Figure 11 and Figure 12, where the x-axis is the system utilization U/m and y-axis represents `Improvement(prmt)` and `Improvement(migr)`, respectively.

The value of `Improvement` for both preemptions and migrations is non-negative in almost all the utilization levels. `FPP` scheduling can significantly reduce the number of preemptions and migrations that are incurred in `G-EDF` scheduling. Since it is more difficult to schedule task sets at higher utilization levels, the improvement decreases as utilization level increases in Figure 11 and Figure 12.

7.2 Other Implementation Issues

Brandenburg and Anderson identified six different major sources of overhead in implementing `G-EDF` algorithm using a Linux extension, called LITMUS$^{\mathrm{RT}}$, that allows different multiprocessors

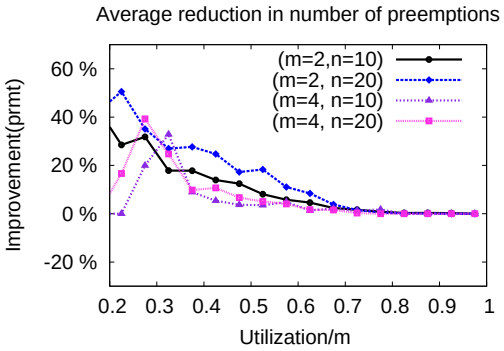


Figure 11 `Improvement(prmt)` of `FPP` over `G-EDF` scheduling.

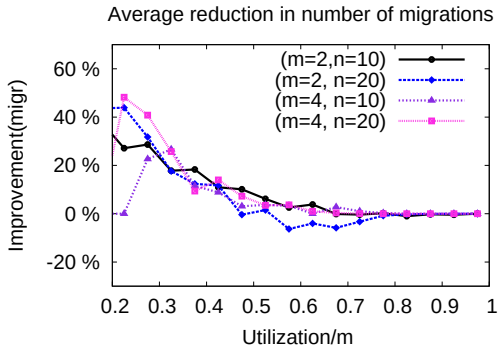


Figure 12 `Improvement(migr)` of `FPP` over `G-EDF` scheduling.

algorithm to be implemented as plugin components [7]. By conducting similar experiments for `FPP` scheduling, the execution time of each task can be inflated to account such overhead in the corresponding schedulability analysis. Although this paper does not implement `FPP` scheduling algorithm in an RTOS, two important implementation issues of `FPP` scheduler warrant further discussion: sharing the ready queue and managing timers for multiple `pri_prom` events.

If scheduling decisions are handled on multiple processors, for example, arrival of different jobs are handled concurrently on different processors (similar to [7]), then the ready queue of `FPP` scheduler is a shared resource. This ready queue needs to be protected against concurrent updates using synchronization primitives. As a result, the operations on the ready queue of `FPP` scheduler can be done in constant time (as discussed in Section 4.2) plus any additional delay incurred by such synchronization primitives. Note that race condition in handling multiple events in `FPP` scheduling will not occur. This is because when multiple `rel_prmt`, `rel_no_prmt`, `idle_remv` and/or `pri_prom` events occur very close in time, then these events can be handled in any order and properties **P1** and **P2** (defined in Section 4.2.2) continue to hold regardless of the order these (nearly concurrent) events are processed.

Another issue to implement `FPP` scheduling is the mechanism used to implement priority promotion. One way to implement such priority promotion is by using hardware timers to handle `pri_prom` events: when a programmed timer expires, the handler can promote the priority and repositions the ready job to the appropriate higher-priority linked list of the ready queue. If the number of hardware timers is not sufficient to implement all the `pri_prom` events, then a queue of timers needs to be managed. In next section, different techniques to implement priority promotions are proposed. Given the effectiveness of `FPP` scheduling in reducing (i) the number of re-mappings of jobs in the ready queue, and (ii) the number of preemptions and migrations, I expect that `FPP` scheduling when implemented on real platform would show benefits over `EDF` scheduling.

Applicability of `FPP_Test` to Arbitrary-Deadline Tasks. The `FPP_Test` in Figure 4 can also be applied arbitrary-deadline tasks as follows. For uniprocessor platform, the iterative `FP` test proposed by Lehoczky [20] can be used as the T_{fp} test and the QPA test [30], which also applies to arbitrary-deadline tasks, can be used as the T_{edf} test. For multiprocessor platform, the iterative OPA-incompatible global `FP` test proposed for arbitrary-deadline tasks by Guan et al. [28] can be made OPA-compatible using approach used by Davis and Burns for the `DA-LC` test in [14]. This new test then can be used as the T_{fp} test in Figure 4. And, the `G-EDF` test proposed by Baruah and Baker [3] can be used as the T_{edf} test for arbitrary-deadline tasks.

8 Techniques to Implement Priority Promotion

This section presents different techniques to implement priority promotion, i.e., how event `pri_prom` is implemented and is handled. First, four different hardware timer-based approaches are presented to implement priority promotion (Subsection 8.1) along with a discussion about advantage and disadvantage of each alternative. Second, a software-based approach that does not rely on any support of hardware timer is presented (Subsection 8.1). A detailed implementation of priority promotion using each of the suggested approaches is left as a future work.

In `FPP` scheduling, the priorities of the currently-executing jobs need to be promoted so that preemption decision can be taken when a (new) job is released while all the cores are busy. In addition, priorities of the jobs stored in the ready queue need to be promoted to ensure that properties P1 and P2 (also restated below for better readability of this section) always hold.

- **P1**: All jobs stored in the ν^{th} linked list of the ready queue have higher `EDF` priorities than any other job stored in the ℓ^{th} linked list where $\nu < \ell$.
- **P2**: All jobs in the ℓ^{th} linked list are stored in order of non-increasing `EDF` priority, i.e., the `head[`ℓ`]` and `tail[`ℓ`]` respectively points the highest and lowest priority `EDF` job in the ℓ^{th} linked list for $\ell = 1, \ldots n$.

Based on the operations on the ready queue (please see Subsection 4.2.2) an important observation for job J_k of task τ_k is presented below:

▶ **Observation 8.** *The TCB of job J_k is never stored in the (lower-priority) ℓ^{th} link lists where $\ell = (k+1), (k+2), \ldots n$. This is because the priority of the job is k or higher (i.e., $(k-1), (k-2), \ldots 1$). When the TCB of job J_k is in the ℓ^{th} linked list of the ready queue at time t where $1 \leq \ell \leq k$, its absolute deadline is not larger than D_ℓ relative to t.*

Maintaining both P1 and P2 at each time instant is the key to efficiently (in constant-time) perform the insertion and removal operations to and from the ready queue, which is same as `FP` scheduler in terms of time complexity (please see Subsection 4.2.2). However, unlike `FP` scheduling, priority promotions cause repositioning of jobs in the ready queue. Overhead related to such repositioning depends on how priority promotion (i.e., event `pri_prom`) is implemented and handled. In this section, we present techniques to implement priority promotions (i) based on hardware-based approach using timers (Subsection 8.1), and (ii) software-based approach with no timer (Subsection 8.2). In the remainder of this section, we consider that all the tasks are assigned priorities based on `IPDD` policy and need priority promotion.

According to `IPDD` priority-promotion policy, the initial priority of a job of task τ_i is i. This initial priority i is promoted to priority levels $(i-1), (i-2), \ldots 1$ at offsets $(D_i - D_{i-1}), (D_i - D_{i-2}), \ldots (D_i - D_1)$ relative to the release time of the job. The difference in time between the $(\kappa-1)^{th}$ and κ^{th} promotion times is denoted as θ^i_κ and is given as follows for $\kappa = 1, 2, \ldots (i-1)$:

$$\theta^i_\kappa = D_{i-(\kappa-1)} - D_{i-\kappa} \tag{1}$$

If a job of task τ_i is released at time r_i, the first promotion point is at $(r_i + \theta^i_1)$ which is θ^i_1 time units later than its release time. The second priority-promotion point is θ^i_2 time units later than the 1^{st} priority promotion point. In general, the κ^{th} priority-promotion point is θ^i_κ time units later than the $(\kappa-1)^{th}$ priority-promotion point.

8.1 Hardware Timer-Based Priority Promotion

Most computer platforms have a clock that increments a counter and can be programmed to generate an interrupt when the counter reaches a certain expiration count called the expiration

time. The combination of a clock and an expiration time is called a timer. In this subsection, we present different alternatives to implement priority promotion based on such hardware timers.

Alternative 1. Consider a platform where the number of hardware timers is sufficient such that one timer can be used for each active job. For such a platform, a one-shot timer can be programmed at each promotion time of an active job such that the timer expires at next priority-promotion time. Based on this principle, when a job of task τ_i is released at time r_i, a hardware timer is programmed to expire after θ_1^i time units. Remember that the initial priority i has to be promoted to priority level $(i-1)$ at time $(r_i + \theta_1^i)$.

If a job completes its execution before the timer is expired, then the timer is disabled to avoid unnecessary interrupt at expiration. Otherwise, when the timer generates an interrupt at time $(r_i + \theta_1^i)$, the priority of the job is promoted to priority level $(i-1)$ and the timer is programmed again to expire after θ_2^i time units at which priority is promoted to $(i-2)$ and so on. Following this approach, the timer is programmed to expire after θ_κ^i time units whenever priority of the job is set to priority level $(i-(\kappa-1))$ for $\kappa = 1, 2, \ldots (i-1)$. After the priority of the job is set to (highest) priority level 1, the timer is no more programmed since there is no more priority promotion of this job according to `IPDD` policy.

While this approach is simple, it may not work for platform where the number of active jobs is larger than the number of available hardware timer. In such case, a queue of (future) timed events related to (future) priority promotions needs to be implemented using, for example, a single hardware timer. In the remainder of this subsection, different alternatives to manage a queue of (priority-promotion related) timed events using a single hardware timer are presented.

Alternative 2. A queue of (future) timed events can be maintained based on a *timing wheel* which is a sequential array of records [29]. Inspired from the discussion of Baruah et al. in [5], we assumed that time is represented using non-negative integers. The required number of records of the timing wheel depends on the maximum time difference between any two consecutive priority promotions. Based on Eq. (1), the maximum length between two consecutive priority promotions of a job of τ_i is $\max_{\kappa=1}^{i-1} \{\theta_\kappa^i\}$. Consequently, the maximum length of any two consecutive priority promotions for any task in set $\{\tau_1, \tau_2, \ldots \tau_n\}$ is $\max_{i=1}^{n} \{\max_{\kappa=1}^{i-1} \{\theta_\kappa^i\}\}$. For implementing a queue of priority-promotion related timed events, the required number of records of the timing wheel, denoted by L, is given as follows in Eq. (2):

$$L \;=\; 1 + \max_{i=1}^{n} \left\{ \max_{\kappa=1}^{i-1} \{\theta_\kappa^i\} \right\} \tag{2}$$

At each position of the timing wheel, a linked list of pointers to jobs' TCBs is stored. A variable to track *current time*, called `CT`, is initially set to 0. This variable is incremented $(mod\ L)$ at every system tick. In other words, `CT` is set to 1 after the first tick, set to 2 after the second tick, set to back to 1 after L ticks, and so on. Such a timing wheel is shown in Figure 13 and Figure 14.

When a job of task τ_i is released, its next promotion time is θ_1^i time units later relative to current time. A pointer to the TCB of this job is stored in the linked list of the timing wheel at position $(\texttt{CT} + \theta_1^i)\ mod\ L$. When `CT` points to a location of the timing wheel at which the linked list is non-empty, the priorities of the jobs pointed by the elements of this linked list are promoted. After the priority of a job of task τ_i at time `CT` is promoted to priority level $(i-(\kappa-1))$ for some κ such that $1 \leq \kappa \leq (i-1)$, a pointer to the TCB of this job is again stored in the linked list at position $(\texttt{CT} + \theta_\kappa^i)\ mod\ L$ of the timing wheel for next (i.e., κ^{th}) priority promotion of this job.

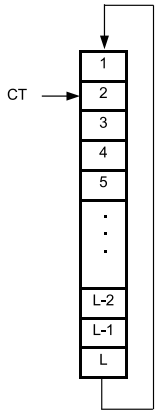


Figure 13 A timing wheel. CT points to the second record. At every system tick, CT is incremented by 1 (*mod* L).

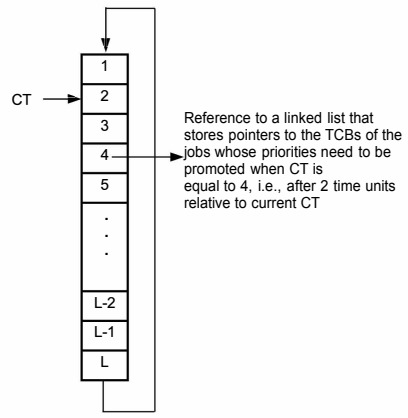


Figure 14 The linked list at position 4 stores pointers to the TCBs of those jobs who need priority promotion when CT will point location 4. The CT is now at location 2 will point location 4 after 2 ticks.

According to Eq. (2), the value of L is one larger than the maximum difference between any two consecutive promotion points. Therefore, $(\texttt{CT}+\theta^i_\kappa)$ *mod* L is never equal to CT. In other words, to distinguish between two (future) priority promotions separated by exactly $\max_{i=1}^{n}\{\max_{\kappa=1}^{i-1}\{\theta^i_\kappa\}\}$ time units, the number of records (as is shown in Eq. (2)) of the timing wheel is one larger than the maximum difference between any two consecutive priority promotions.

Alternative 2 to manage multiple priority promotions requires the variable CT to be incremented at every timer's tick which may have high overhead. Next we propose Alternative 3 based on timing wheel but with the exception that variable CT is not incremented at every system's tick.

Alternative 3. Similar to Alternative 2, multiple (future) timed events related to priority promotions are stored in the linked lists of a timing wheel but without requiring to increment variable CT at every system's tick. A bitmap, denoted by $S[1, 2 \ldots L]$, of length L corresponding to the timing wheel is maintained. If the linked list of the a^{th} position of the timing wheel is empty, then $S[a] = 0$; otherwise, $S[a] = 1$. When all the linked lists of the timing wheel are empty, we set CT to 0. The main idea is to program a one-shot timer that expires after a duration equal to the earliest time of occurrence of any (future) timed event stored in the timing wheel. The approach is described as follows:

- **(How an element is inserted in an empty timing wheel?)** Consider that there is no element in the timing wheel, i.e., $\texttt{CT} = 0$ and $\forall_\ell; S[\ell] = 0$ at time t. Now consider that a new (future) timed event appears at time t such that this event needs to occur after a time units relative to time t. In other words, there is a future timed event that occurs after a time units relative to CT. To implement priority promotion related to this event, a pointer to the TCB of the job is inserted in the a^{th} linked list of the timing wheel. A one-shot timer is set to expire after a time units. To remember the total duration for which this timer is programmed, we set Initial Value of the Timer as $\texttt{IVT} = a$. We also set $S[a] = 1$ to specify that the a^{th} linked list of the timing wheel is non-empty.
- **(How an element is inserted in a non-empty timing wheel?)** Now consider that a new (future) timed event appears at time t' such that this event needs to occur after c time units relative to time t' and the timing wheel is non-empty at time t'. Since the timing wheel is non-empty, the timer (that was last programmed) is running (not yet expired) at time t'.

Assume that at time t' the remaining time to expire the timer is b. The total time elapsed between the time instant when the timer was last programmed and time t' is $(\texttt{IVT} - b)$ since IVT stores the total duration for which the time was last programmed. At time t' the value of CT is updated by setting it equal to the old value of CT plus elapsed time from the time CT was last set, i.e, $\texttt{CT} = \texttt{CT} + (\texttt{IVT} - b)$. In addition, IVT is set to $\texttt{IVT} = b$ to reflect the fact that the timer expires after b time units relative to the (updated) current time CT. After updating CT and IVT, we consider where the new event that appears at time t' is to be inserted in the timing wheel.

If $c < b$, then the new event has an earlier time of occurrence than that of any event stored in any linked list of the timing wheel. Remember that the one-shot timer was programmed to expire at time when the earliest event in the timing wheel will occur. Since $c < b$, the one-shot timer is reprogrammed at time t' to expire after c time units and we set $\texttt{IVT} = c$. On the other hand, if $c > b$, then the new event does not have an earlier occurrence time and the one-shot timer is not reprogrammed. After CT and IVT are updated at time t', a pointer to the TCB of the job corresponding to the new event is inserted to the $(\texttt{CT} + c)^{th}$ linked list of the timing wheel. Note that the value $(\texttt{CT} + c)$ essentially means $(\texttt{CT} + c)$ *mod* L. If $(\texttt{CT} + c)^{th}$ linked list was empty before this insertion, we set $S[q] = 1$ where $q = (\texttt{CT} + c)$ *mod* L.

- **(How to deal with timer's expiration?)** The timer expires when no new event with earlier time of occurrence appears after IVT was last set. When the one-shot timer expires after IVT time units and generates an interrupt, the value of CT is updated by setting $\texttt{CT} = (\texttt{CT} + \texttt{IVT})$ *mod* L. And, the priorities of the jobs pointed by the TCB pointers stored in the $\texttt{CT}^{th}$ linked list of the timing wheel are promoted. If the next promotion time for such a job is a time units later, then the pointer to the TCB of that job is again inserted in the $(\texttt{CT} + a)^{th}$ linked list and we set $S[\texttt{CT} + a] = 1$. After processing each element of the $\texttt{CT}^{th}$ linked list in the timing wheel, all the TCB pointers are removed from the $\texttt{CT}^{th}$ linked list and we set $S[\texttt{CT}] = 0$.

  If the timing wheel is not empty after handling a timer's expiration, then we have to program the timer for the next promotion event that will occur the earliest. To program the timer for the next earliest promotion time, the position of the first set bit of bitmap S starting from position CT is determined. This can be done based on techniques similar to finding the highest priority tasks from the ready queue of FP scheduler. Let this position is $(\texttt{CT} + k)$ *mod* L, i.e., this position points to the k^{th} linked list relative to the index of $\texttt{CT}^{th}$ linked list. If k is not a valid index, then there is no element in the timing wheel and we set $\texttt{CT} = 0$. For a valid k, the jobs corresponding to the TCB pointers stored in the $(\texttt{CT} + k)^{th}$ linked list have the earliest promotion time. The timer is programmed to expire after k time units and we set $\texttt{IVT} = k$.

Note that in the approach described above, variable CT is not updated at every system's tick. It is updated either when a new event is to be stored in the timing wheel and/or when the timer expires. One of the limitations with this approach is that if L is too large, then a hierarchical bitmap needs to be maintained (as is suggested in [27]).

Alternative 4. This alternative to manage a queue of (future) timed events is based on the RELTEQ approach proposed in [19]. The main idea of RELTEQ is that events are stored in an ordered list based on the time of occurrences of the events relative to each other. The advantage of this approach is that no bitmap needs to be maintained. But the disadvantage is that linear search is needed to find the appropriate position for each new event. Please see details of RELTEQ in [19].

8.2 Software-Based Approach to Priority Promotion

In this subsection, we present software-based approach to show how priority promotion can be implemented without using a hardware timer. Under this scheme, the the priorities of the jobs that are in execution are never promoted. The priority of a job is promoted only if the job is in the ready queue. For such a job in the ready queue, priority promotions that are due at a time instant are *delayed* as long as properties P1 and P2 of the ready queue of `FPP` scheduler hold.

The jobs whose promotions are delayed are called *colluding jobs*. When any of the `rel_prmt` or `rel_no_prmt` event occurs (i.e., a new TCB has to be inserted in the ready queue due to the release of new job), we check if insertion of this new TCB in the ready queue according to the approach presented in Subsection 4.2.2 could violate property P1 or P2. Note that such violation may happen since priorities of the colluding jobs were not promoted when their promotions were due and the corresponding TCBs of colluding jobs were not repositioned in the right place in the ready queue. If we detect that such violation would occur, we fix the collusion by promoting some or all colluding jobs so that property P1 and P2 continue to hold after insertion of the new TCB.

A job J_i of task τ_i is promoted according to `IPDD` priority-promotion policy in order to determine whether a newly released job J_k of task τ_k needs to preempt the execution of J_i or not. In contrast, the priorities of the currently-executing jobs are never promoted in delayed preemption strategy. Whether a newly released job J_k preempts J_i or not can be determined by comparing their absolute deadlines since we want to execute jobs in `EDF` order. Therefore, we compare $(r_i + D_i)$ and $(r_k + D_k)$. If $(r_i + D_i) > (r_k + D_k)$, then J_k preempts J_i; otherwise, J_k does not preempt J_i. In the remainder of this section, we present how the new TCB is inserted in the ready queue such that property P1 and P2 of the ready queue continue to hold after this insertion.

Assume that property P1 and P2 hold at time t. Consider an earliest time instant t' at which a new job J_k of task τ_k is released and all the cores are busy such that $t < t'$. A new TCB (i.e., TCB of J_k or TCB of the preempted job J_i) is to be inserted in the ready queue at time t'. Since t' is the earliest time at which a new TCB is to be inserted in the ready queue, property P1 and P2 continue to hold during the entire interval $[t, t')$ even if all promotions of the jobs in the ready queue during this interval are delayed. In delayed promotion strategy, the promotions of the jobs are delayed until a new job is released. Due to such delayed promotions, colluding jobs in the ready queue are not promoted (i.e., repositioned) to higher-priority linked lists of the ready queue. Consequently, property P1 and P2 may not hold at time t' if the new TCB is inserted without fixing the collusion. The challenge is to propose mechanism to ensure that after inserting the new job, property P1 and P2 continue to hold also at time t'. There are two cases to consider:

- Case (i) – Job J_k preempts J_i.
- Case (ii) – Job J_k does not preempt J_i.

Case (i) – J_k preempts J_i: In such case, a newly released job J_k preempts the currently-executing lowest `EDF` priority job J_i at time t'. Since J_i was in execution just before J_k was released and because property P1 and P2 hold during $[t, t')$ during which no new job is released, the `EDF` priority of job J_i is larger than the `EDF` priority of any other job in the ready queue.

Job J_i is inserted at the front of the highest-priority non-empty linked list if the index of the highest-priority non-empty linked list of the ready queue is smaller than or equal to i; otherwise, it is inserted as the first element in the i^{th} linked list of the ready queue. Such insertion can be done in constant time and ensures that Observation 8 still holds. It is easy to see that property P1 and P2 continue to hold at time t' after inserting the new TCB of J_i in the ready queue even though all due promotions during $[t, t']$ are delayed. In such case, the delayed promotions in $[t, t']$ are delayed further.

Algorithm: `Fix_Collusion`**(Job** J_k**)**

// This algorithm is executed when a new job J_k cannot
// preempt any job and needs to be inserted in
// the ready queue (i.e., when `rel_no_prmt` event occurs)

1. $NextList = k + 1$
2. $s =$ Position of the first set bit of bitmap $B[NextList \dots n]$
3. **If** s is a valid position of bitmap B
4. **While** (the absolute deadline of the first element of the s^{th}
5. linked list is smaller than the absolute deadline of J_k)
6. $J =$ remove the first element from the s^{th} linked-list
7. Insert J at the end of the k^{th} linked list
8. **If** the s^{th} linked list is empty
9. $NextList = s + 1$
10. Go to Step 2
11. **End If**
12. **End While**
13. **End If**
14. Insert J_k at the end of the k^{th} linked list

Figure 15 Coalescing Priority Promotion to handle `rel_no_prmt` event.

Case (ii) – J_k does not preempt J_i: In such case, a newly released job J_k does not preempt the currently-executing lowest `EDF` priority job J_i at time t'. The TCB of job J_k is to be inserted in the ready queue. Based on the operations proposed in Subsection 4.2.2, this new TCB is inserted at the end of the k^{th} linked list since the priority of job J_k at time t' is k. However, property P1 and P2 may not hold at time t' because promotions of the colluding jobs during $[t, t')$ are delayed and not placed in the right position in the ready queue before inserting this new TCB. The strategy to fix the collusion before inserting job J_k is as follows.

The absolute deadline of job J_k is D_k time units later than time t' because job J_k is released at time t'. Since jobs in the ℓ^{th} linked list of the ready queue at time t' have their absolute deadlines no later than D_ℓ time units relative to t' for $\ell = 1, 2, \dots k$, the `EDF` priorities of the jobs in these linked lists are higher than that of job of J_k (follows from Observation 8). If job J_k is inserted at the end of k^{th} linked list, property P1 and P2 continue to hold at time t' for the $1^{st}, 2^{nd}, \dots, (k-1)^{th}, k^{th}$ linked lists even if priority of the jobs in these linked lists are not promoted during $[t, t']$.

On the other hand, since priority promotions of the jobs in $(k+1)^{th}, (k+2)^{th}, \dots, (n-1)^{th}, n^{th}$ linked lists are also delayed during $[t, t']$, the actual `EDF` priority of some of the colluding jobs in these linked lists may be higher than that of job J_k at time t'. Such colluding jobs need to be promoted (i.e., need to be repositioned) to fix the collusion so that property P1 and P2 continue to hold after job J_k is inserted at the end of the k^{th} linked list. Although there may be many colluding jobs, we only need to promote the priority (i.e., reposition in the ready queue) of those colluding jobs from the $(k+1)^{th}, (k+2)^{th}, \dots, (n-1)^{th}, n^{th}$ linked lists whose `EDF` priority is larger than the `EDF` priority of J_k at time t' to fix the collusion.

Notice that property P1 and P2 hold for the jobs in the $(k+1)^{th}, (k+2)^{th}, \dots, (n-1)^{th}, n^{th}$ linked lists before inserting the new TCB at time t'. Before inserting job J_k at the end of the k^{th} linked list, the higher `EDF` priority jobs from the $(k+1)^{th}, (k+2)^{th}, \dots, (n-1)^{th}, n^{th}$ linked lists are inserted (in decreasing `EDF` order) at the end of the k^{th} linked list. Finally, job J_k is inserted at the end of the k^{th} linked list. The following algorithm in Figure 15, called `Fix_Collusion`, implements this insertion.

Algorithm `Fix_Collusion` in Figure 15 selects those jobs from the $(k+1)^{th}, (k+2)^{th}, \ldots, (n-1)^{th}, n^{th}$ linked lists whose `EDF` priorities are higher than that of job J_k. These selected jobs are inserted in decreasing `EDF` order at the end of the k^{th} linked list, and finally, job J_k is inserted at the end of the k^{th} linked list. We will show that property P1 and P2 continues to hold after algorithm `Fix_Collusion` is executed at time t'.

Line 1 initializes a variable $NextList$ to $(k+1)$ in order to start the search from the $(k+1)^{th}$ linked list. However, the $(k+1)^{th}$ linked list may be empty. The index of the first non-empty linked list among the $(k+1)^{th}, (k+2)^{th}, \ldots, (n-1)^{th}, n^{th}$ linked lists is determined in line 2 based on the bitmap $B[NextList, \ldots n]$. The index of the first set bit of the bitmap $B[NextList, \ldots n]$ is stored in variable s in line 2.

If all of the $(n-k)$ lower priority linked lists are empty, then s has an invalid index. The condition in line 3 checks whether all of the $(n-k)$ lower priority linked lists are empty or not. If the condition in line 3 is false (i.e., there is no non-empty linked lists among the $(n-k)$ lower priority linked lists), then there is no TCB in the $(k+1)^{th}, (k+2)^{th}, \ldots, (n-1)^{th}, n^{th}$ linked lists and the TCB of job J_k is inserted at the end of the k^{th} linked list in line 14. Since P1 and P2 hold for k higher priority linked lists before inserting J_k (i.e., jobs in these linked lists have deadline no larger than D_k relative to time t') and since the deadline of J_k is D_k at time t', P1 and P2 hold after job J_k is inserted.

If condition in line 3 is true, then s is the index of the highest-priority non-empty linked lists among the $(k+1)^{th}, (k+2)^{th}, \ldots, (n-1)^{th}, n^{th}$ linked lists. In such case, all the TCBs of the jobs having higher `EDF` priority than J_k (based on the condition of the while loop in line 4–5) are removed from the s^{th} linked list one-by-one in line 6 and inserted in non-increasing `EDF` priority order at the end of the k^{th} linked list in line 7.

Since the first job is removed from the s^{th} linked list each time condition in line 4–5 is true and because property P2 is satisfied (jobs in each list are in non-increasing `EDF` priority order), inserting the removed job J at the end of the k^{th} linked list ensures that jobs in the k^{th} linked list are in non-increasing `EDF` order.

We exit from the while loop in two cases: (i) some job's `EDF` priority in the s^{th} linked list is lower than the `EDF` priority of J_k (i.e., condition in the while loop is false), or (ii) all the jobs from the s^{th} linked list are removed (i.e., condition in line 8 is true). In the first case, the algorithm exit from the loop and executes line 14. This is because there is no other jobs in the $(n-k)$ lower priority linked list having higher `EDF` priority. In the second case (when the condition in line 8 is true), the s^{th} list is empty and the `NextList` is set to $(s+1)$ to select other higher `EDF` priority jobs from the remaining $(n-s)$ linked lists. In such case, the algorithm jumps to line 2 from line 10 and continues as described above. It is easy to see that number of times the while loop executes is no more than the number of delayed promotions in $[t, t']$. When the algorithm stops, property P1 and P2 hold at time t'. Note that we need no timer to implement priority promotions based on software-based delayed promotion mechanism. Evaluating all these priority promotion schemes and the implementation of `FPP` scheduling on real platform is left as a future work.

9 Conclusion

The proposed `FPP` scheduling algorithm shows how jobs can be executed in `EDF` order based on priority promotion. For uniprocessor, the `FPP` scheduling is also optimal as `EDF` scheduling. For multiprocessors, it dominates the state-of-the-art `G-FP` and `G-EDF` tests for constrained-deadline tasks. A technique to reduce the number of promotion points is proposed so that overhead is low. The proposed data structure and operations for managing jobs in the ready queue of `FPP` scheduler have benefits similar to that of traditional `FP` scheduler. Techniques to implement

priority promotions based on hardware timers or purely in software are proposed. Simulation results show that the overhead for managing jobs in the FPP ready queue is reduced significantly in comparison to that of an EDF scheduler.

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, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=238595.

3 Sanjoy K. Baruah and Theodore P. Baker. Global EDF schedulability analysis of arbitrary sporadic task systems. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 3–12. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.27.

4 Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium – 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 182–190. IEEE Computer Society, 1990. doi:10.1109/REAL.1990.128746.

5 Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990. doi:10.1007/BF01995675.

6 Marko Bertogna and Sanjoy K. Baruah. Tests for global EDF schedulability analysis. *Journal of Systems Architecture – Embedded Systems Design*, 57(5):487–497, 2011. doi:10.1016/j.sysarc.2010.09.004.

7 Björn B. Brandenburg and James H. Anderson. On the implementation of global real-time schedulers. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 214–224. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.23.

8 Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November to 3 December 2008*, pages 157–169. IEEE Computer Society, 2008. doi:10.1109/RTSS.2008.23.

9 Alan Burns. Dual Priority Scheduling: Is the Processor Utilisation bound 100%? In *Proc. of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS), in conjunction with the ECRTS*, 2010. URL: https://www.cs.york.ac.uk/ftpdir/papers/rtspapers/R:Burns:2010b.pdf.

10 Alan Burns, Marina Gutierrez, Mario Aldea Rivas, and Michael González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Trans. Computers*, 64(5):1241–1253, 2015. doi:10.1109/TC.2014.2322619.

11 Alan Burns and Andrew J. Wellings. Dual priority assignment: A practical method for increasing processor utilisation. In *Fifth Euromicro Workshop on Real-Time Systems, RTS 1993, Oulu, Finland, June 22-24, 1993. Proceedings.*, pages 48–53. IEEE, 1993. doi:10.1109/EMWRT.1993.639052.

12 Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29(1):5–26, 2005. doi:10.1023/B:TIME.0000048932.30002.d9.

13 Robert I. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. *Technical Report YCS 230, Dept of Computer Science, University of York, UK*, 1994. URL: https://www.cs.york.ac.uk/ftpdir/reports/94/YCS/230/YCS-94-230.ps.Z.

14 Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority preemptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011. doi:10.1007/s11241-010-9106-5.

15 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

16 Robert I. Davis and Andy J. Wellings. Dual priority scheduling. In *16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings*, pages 100–109. IEEE Computer Society, 1995. doi:10.1109/REAL.1995.495200.

17 Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

18 Michael González Harbour, Mark H. Klein, and John P. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *Proceedings of the Real-Time Systems Symposium – 1991, San Antonio, Texas, USA, December 1991*, pages 116–128. IEEE Computer Society, 1991. doi:10.1109/REAL.1991.160365.

19 Mike Holenderski, Wim Cools, Reinder J. Bril, and Johan J. Lukkien. Multiplexing real-time timed events. In *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2009, September 22-25, 2008, Palma de Mallorca, Spain*, pages 1–4. IEEE, 2009. doi:10.1109/ETFA.2009.5347183.

20 John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the Real-Time Systems Symposium – 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 201–209. IEEE Computer Society, 1990. doi:10.1109/REAL.1990.128748.

21 Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de bruijn sequences to index a 1 in

a computer word. *MIT Technical Report*, 1998. URL: http://supertech.csail.mit.edu/papers/debruijn.pdf.

22 Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982. doi:10.1016/0166-5316(82)90024-4.

23 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.

24 Risat Mahmud Pathan. Unifying fixed- and dynamic-priority scheduling based on priority promotion and an improved ready queue management technique. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 209–220. IEEE Computer Society, 2015. doi:10.1109/RTAS.2015.7108444.

25 Risat Mahmud Pathan and Jan Jonsson. Improved schedulability tests for global fixed-priority scheduling. In Karl-Erik Årzén, editor, *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*, pages 136–147. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.21.

26 Risat Mahmud Pathan and Jan Jonsson. Interference-aware fixed-priority schedulability analysis on multiprocessors. *Real-Time Systems*, 50(4):411–455, 2014. doi:10.1007/s11241-013-9198-9.

27 Michael Short. Improved task management techniques for enforcing EDF scheduling on recurring tasks. In Marco Caccamo, editor, *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, pages 56–65. IEEE Computer Society, 2010. doi:10.1109/RTAS.2010.22.

28 Youcheng Sun, Giuseppe Lipari, Nan Guan, and Wang Yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, pages 1–9. IEEE Computer Society, 2014. doi:10.1109/RTCSA.2014.6910543.

29 George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 25–38. ACM, 1987. doi:10.1145/41457.37504.

30 Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Computers*, 58(9):1250–1258, 2009. doi:10.1109/TC.2009.58.

Modeling Power Consumption and Temperature in TLM Models

Matthieu Moy[1], Claude Helmstetter[2], Tayeb Bouhadiba[3], and Florence Maraninchi[4]

1 **Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France**
CNRS, VERIMAG, F-38000 Grenoble, France
http://orcid.org/0000-0002-6054-8882
matthieu.moy@imag.fr

2 **Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France**
CNRS, VERIMAG, F-38000 Grenoble, France
http://orcid.org/0000-0002-2323-6919
claude.helmstetter@imag.fr

3 **Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France**
CNRS, VERIMAG, F-38000 Grenoble, France
http://orcid.org/0000-0003-1694-6709
tayeb.bouhadiba@imag.fr

4 **Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France**
CNRS, VERIMAG, F-38000 Grenoble, France
http://orcid.org/0000-0003-0783-9178
florence.maraninchi@imag.fr

Abstract

Many techniques and tools exist to estimate the power consumption and the temperature map of a chip. These tools help the hardware designers develop power efficient chips in the presence of temperature constraints. For this task, the application can be ignored or at least abstracted by some high level scenarios; at this stage, the actual embedded software is generally not available yet.

However, after the hardware is defined, the embedded software can still have a significant influence on the power consumption; i.e., two implementations of the same application can consume more or less power. Moreover, the actual software power manager ensuring the temperature constraints, usually by acting dynamically on the voltage and frequency, must itself be validated. Validating such power management policy requires a model of both actuators and sensors, hence a closed-loop simulation of the functional model with a non-functional one.

In this paper, we present and compare several tools to simulate the power and thermal behavior of a chip together with its functionality. We explore several levels of abstraction and study the impact on the precision of the analysis.

2012 ACM Subject Classification Hardware - Power and energy - Power estimation and optimization - Chip-level power issues

Keywords and phrases Power consumption, Temperature control, Virtual prototype, SystemC, Transactional modeling

Digital Object Identifier 10.4230/LITES-v003-i001-a003

Received 2015-07-10 **Accepted** 2016-04-29 **Published** 2016-06-29

1 Introduction

Reducing power consumption of Systems-on-a-Chip is an important challenge. Clearly, portable devices should save energy to maximize battery life, but other issues like heat dissipation [21], voltage drop [11] or faster aging [14] due to overheating are also of growing importance.

With modern CMOS processes, static power consumption, due to leakage current, is increasing. Power-saving techniques like clock-gating, that act only on dynamic consumption are no longer sufficient. More heavy-weight techniques like DVFS and power-gating are necessary to control the consumption of the chip. These mechanisms use sensors for non-functional properties (temperature, voltage, ...) to control the chip's clock generators and power supply [37, 3]. The power management policy itself is usually implemented in software.

Developing such power management policies requires an execution platform, including models of sensors, actuators, and taking into account the feedback loop between actuators and sensors. Low-level (RTL, gate-level or even SPICE) simulations are possible, but too late in the design flow and their simulation is not fast enough for system-level simulation including non-trivial software. On the other hand, an accurate model of power consumption and temperature requires a realistic model of timing.

Virtual prototyping of Systems-on-a-Chip is a well-established practice. The standard technology for fast and early functional simulation is SystemC/TLM [1], implemented as a C++ library. Several abstraction levels are possible: one can model the timing-behavior precisely, requiring a relatively detailed model of the micro-architecture. Early in the design-flow, more abstract simulations that completely abstract the micro-architecture and with a very loose notion of timing are more applicable.

Estimating power-consumption and temperature on early virtual prototypes is needed to allow early power-aware software development and optimize the power-consumption at the system-level. A solution for system-level power and temperature analysis must satisfy several criteria: it must be fast, to allow simulating non-trivial scenarios on the whole hardware+software system. It must be applicable early in the design flow, hence should require a low development effort, and it must be applicable on early, hence abstract, virtual prototypes.

This paper presents several solutions satisfying these three criteria to augment a functional SystemC/TLM model with non-functional information, and to perform power and thermal analysis using a dedicated solver. The tool LIBTLMPWT [25] uses a tight coupling between the functional simulation and the non-functional solver, and exploits this tight coupling for performance optimizations (`pwt` stands for *PoWer and Temperature*). We also developed an approach that allows using an external solver as black box and in a separate process [10], and showed how the cosimulation approach could be used with loosely-timed models [9], validated by a case-study [16]. The present paper gives a global but detailed presentation of these different approaches. It includes both improvements in presentation of previously published content and unpublished results.

We focus on the *cosimulation* of an existing thermal model (Hotspot [28], ATMI [38] or Aceplorer [31]) with a functional SystemC/TLM simulation. The internals of the thermal model are out of the scope of this paper: we study the coupling of simulators, and the impact of the modeling style in SystemC/TLM on the accuracy of the result.

More specifically, the contributions of this paper are:

- A detailed presentation of the tool LIBTLMPWT. Although the tool itself was published with a technical report [25], it was never presented in a peer-reviewed publication. The tool provides a new method for power instrumentation with low performance overhead. It take into account standard SystemC/TLM performance optimizations like temporal decoupling and direct memory interface (DMI). The annotation mechanism is based on *activity ratios*, and takes into account the frequency changes automatically. It is available as free software [27], including the instrumentation API, the cosimulation engine, and a graphical user interface.
- An analysis of different levels of abstraction, and their consequences on possible cosimulation techniques, with a comprehensive state of the art.

- A method to model abortion following an interrupt in loosely-timed systems.
- New experimental studies, including an analysis of the development cost.

2 Modeling Power Management Policies

2.1 Power Management in Modern System-on-Chips

Power consumption of a silicon chip comes mainly from two sources:

dynamic power comes from switching activity. One can see transistors and wires as small capacitors that are filled-in and emptied when the corresponding value switches from 0 to 1 or 1 to 0. This consumption therefore depends on the capacitance of physical elements (hence, of the physical process), and on the frequency at which values change, i.e. how much computation is being performed by the component.

static power comes from leakage current. For a given hardware component, it depends on the state of the power supply (on or off, and voltage), but is independent of the computation being performed. The leakage current tends to increase when the size of transistors is reduced. Since heating silicon increases its conductivity, the static power also increases with temperature.

A simple way to reduce dynamic power consumption is *clock gating*, which stops the switching activity by stopping the clock. Clock gating can be applied automatically by the synthesizer in many cases, and the problem has been studied extensively for decades (e.g. [2, 6]). Clock-gating, however, is not sufficient in modern Systems-on-Chip, where static power is non-negligible, if not dominant.

To reduce static power, heavier techniques have to be used. *Dynamic Voltage and Frequency Scaling (DVFS)* allows reducing the voltage, hence both static and dynamic power, but requires a lower frequency. Switching from a voltage/frequency couple to another takes time, hence the policy behind DVFS is non-trivial, and is usually implemented in software (for example, the `cpufreq` drivers in Linux count for around 20,000 lines of code). Another technique is *power gating*, which consists in disabling the power supply of unused components. Power gating also takes time, and the transitions from a mode to another may take time and consume a lot of energy (typically, before shutting down a component completely, a part of the internal state may have to be saved to some retention registers), hence the policy to chose whether to power-gate a component or not is again non-trivial. In the sequel, we call *power controller* the set of hardware components that directly control these physical mechanisms, and *power manager* the implementation of the policy, usually done at least partially in software. Bugs related to power management, or *power bugs*, can drastically increase the power consumption of a system, and discharge the battery of a mobile phone in a couple of hours [40]. Worse, serious bugs in the power management policy can lead to deadlocks, e.g., waiting for interrupts from a component which has been completely switched off.

The need for low-power is obvious for battery-powered devices, but it is also important for permanently plugged ones like set-top-boxes: in addition to being environment-friendly, a low-power system needs cheaper packaging and avoids the need for a noisy cooling system. Power consumption cannot be evaluated precisely independently from temperature, as hotter silicon consumes more (and consuming more in turn increases temperature).

Temperature peaks can physically damage the chip. Since high temperature increases silicon conductivity, hence static power consumption, a temperature peak can lead to a consumption peak, and therefore a voltage drop. If the voltage goes below the appropriate threshold, the system may stop working. It is therefore critical to have thermal estimations before manufacturing the real system to dimension the packaging and cooling system appropriately.

Also, high temperature gradients lead to faster aging of the chip [14]. To avoid peaks and minimize gradients, the chips are usually equipped with several temperature sensors (typically

one or more per core on a many-core platform [37, 3]), allowing the implementation of a software control loop to regulate temperature.

2.2 Virtual Prototyping for Power Aware Systems

2.2.1 Non-functional Models

A chip's non-functional aspects can be modeled by specifying the power consumption of individual components, the layout of components (called the *floorplan*), and the physical parameters influencing heat dissipation (from a component to another, and from the chip to its environment).

For system-level simulations, modeling individual transistors precisely is clearly too low-level and too slow. The physical parameters have to be abstracted into a set of operating modes for each component, often called *power-state* [5, 7]. A power-state defines the power consumption (in Watts, or the current intensity in Amperes), possibly as a function of temperature. The energy consumption (in Joules, or the charge transported in Coulombs) is obtained by integrating the power over time. When the power consumption of a power-state is constant, the integration is simply a multiplication by a duration. A power-state is defined by several parameters:

Fixed parameters depend on the platform, but do not vary at runtime (we do not use the word "static" to avoid confusion with "static power consumption"):

- *floorplan parameters:* location of the module on the chip; used to compute the area of the module, and the neighbor relations.
- *technology dependent parameters:* physical values that depend on the technology used to produce the physical chip; this includes capacitance, leakage current, influence of temperature on the leakage, etc. Most of these values are common to all modules.

Runtime parameters depend on the platform's activity and configuration:

- *Electrical state* is the voltage and frequency of the component. It is influenced by DVFS and power-gating, and controlled by the power-controller.
- *Activity* defines the computation performed by the component (e.g., waiting, computing, ...). The total power consumption related to activity can be expressed as $K \times F \times V^2 \times \alpha$, where K is a fixed constant (depending on the number of gates and the capacitance of each gate) taken into account in the *technology dependent parameters*, F is the frequency, V the voltage, and α is the *activity ratio*. The activity ratio expresses the proportion of gates involved. It varies from 0 (no activity) to 1 (all gates are active at each cycle).
- *Traffic* is the amount of data processed by unit of time (e.g. number of transactions routed for a bus, number of reads and writes for a memory, ...).

Note that different parameters correspond to different kinds of models. Electrical state fit very well in the power-state model: one only needs to model transitions from state to state. Activity can be modeled in several ways: either an operating mode is modeled as a power-state with a given activity ratio independently from the functionality, or the activity of the functional model is observed during simulation and used in the non-functional model. Traffic has to be modeled separately, since the traffic of a component usually comes from the activity of another component. We cannot define power-states for traffic a priori and need to take into account the traffic of the functional simulation. The solutions presented in this paper take all these parameters into account.

Depending on the stage in the design-flow, the power consumption associated with a power-state can come from different sources. At early stages of development, they can be target values (i.e., that further development will consider as an objective), or extrapolation from previous

```
// SystemC thread
void compute() {
    while (true) {
        set_activity(0); // Enter IDLE mode
        wait(event);
        set_activity(ACTIVITY_RATIO_RUN); // enter RUN mode
        f();
        wait(100, SC_US);
        send_irq();
    }
}
```

Figure 1 Simple Power-model of a Hardware Component (pseudo-code).

generations of the same component. Later in the design flow, they can be more precise estimations based on RTL or gate-level simulations, and physical measurements when a prototype of the chip is available. Our contribution is not to provide these parameters to the user, but to allow exploiting these per-component parameters for a *system-level* simulation including power and thermal management policy and software. Because of heat dissipation, and of the relationship between power and temperature, the composition of the component's parameters is non-trivial.

Tools like *Docea Power*'s *Aceplorer* [31] allow modeling a chip, taking into account both the power consumption and the thermal aspects. The traditional way to use Aceplorer for simulation is to define scenarios that define the sequence of modes for each component. The tool computes the power consumption and the evolution of temperature (including the feedback of temperature on power described above). Scenarios can be provided by hand, using UML's activity diagrams, or can be produced by a SystemC/TLM or RTL simulation.

In the scenario-based usage, the model does not allow the execution of power-aware software: the SystemC or RTL simulation provides non-functional stimuli (typically dumped in a VCD file for offline power/thermal analysis), but has no access to the result of the power and thermal simulation. A temperature sensor, or battery monitor component could not be modeled. Our contribution is to allow such closed-loop simulation.

2.2.2 Functional and Non-functional Models cosimulation

To illustrate the cosimulation concepts, let us first consider a very simple component, with 2 activity states. The component waits for an event in `IDLE` mode, and on reception of the event performs a computation `f` that lasts 100 microseconds in mode `RUN`. A SystemC thread describing this behavior is provided in Figure 1.

The principle of cosimulation is illustrated in Figure 2. The instrumented functional model computes, at each point in time, the power state of each component. This power state information is transmitted (a) to the power model, which uses this power state information together with the temperature T obtained from the thermal solver (c) to compute the power consumption of each component. The power consumption is used by the thermal solver (b) solver to compute the derivative of the temperature. It is used together with the thermal model to compute the evolution of temperature. The temperature (and possibly other non-functional information) are transmitted back to the functional part and embedded software (d) through models of physical sensors.

Note that the scheme illustrated in Figure 2 contains two loops. We focus on the Functional/Non-functional loop ((a) and (d) on the figure: the behavior is influenced by temperature sensors, but

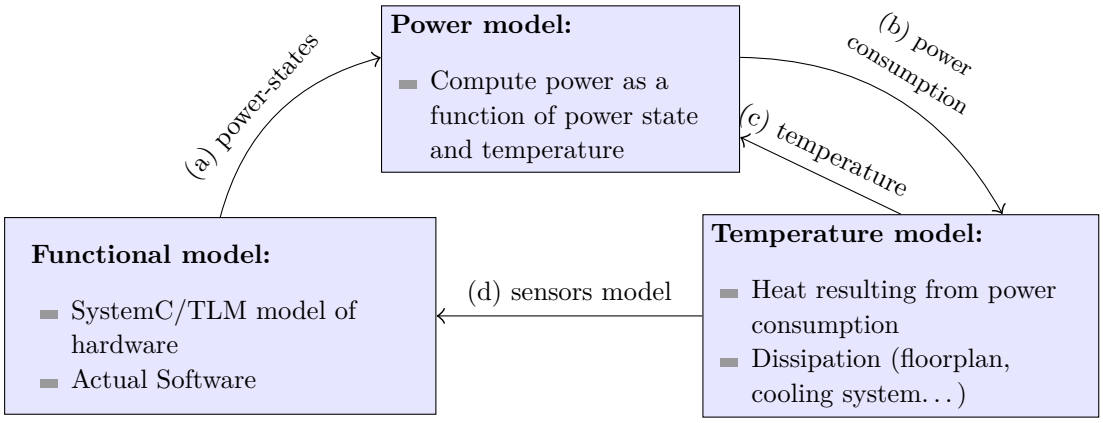


Figure 2 Functional, Power and Thermal Models.

the temperature depends on the behavior). A second loop appears with Power/Temperature ((b) and (c) on the figure: temperature increases as a consequence of power consumption, but static power consumption increases when the temperature increases). This second loop is also managed by our approach: when cosimulating with a black-box power/thermal solver like Aceplorer, the solver deals with it internally. In LIBTLMPWT, the computation of power density can use the temperature (it will actually use the temperature computed at the previous ATMI step).

2.2.3 Power-Aware Software Execution on a Virtual Prototype

To validate these power and thermal managers early in the design flow, one needs virtual prototypes that allow execution of power-aware software. Among these models, various degrees of precision can be achieved:

Purely functional: Even purely functional prototypes need models of temperature sensors: if the embedded software reads a value from one of its registers, then the simulated platform should include a component mapped at this address and returning a sensible value (possibly an arbitrary constant).

Non-functional contracts: Some basic, but yet serious mistakes like reading from or writing to a component which is switched off can be caught by assertions in the model. The assertions form the contract [35] of the components, or of the hardware platform with respect to software.

Scenario-based models: To test some basic power management policies, one may need the non-functional inputs to take different values. For example, if a platform triggers an emergency stop when temperature goes above some threshold, then testing this functionality requires a scenario where the temperature returned by the sensor crosses this threshold. This can be done by returning a pre-defined sequence of values in the temperature sensor model.

Approximate models: When the policy to be tested is non-trivial, manually writing scenarios is not feasible anymore. Not only the scenarios are too complex to be written by hand, but realistic scenarios cannot be generated offline [41]. One needs an automated way to get reasonable sequences of values. A simple thermal model can be sufficient: it will typically let the temperature decrease when the system saves energy and vice versa. For example, a software developer implementing a simple hysteresis policy (switch to power saving mode when temperature is too high, and switch back to normal mode when the temperature crosses a low threshold) can use this model to test that the mode switches are correctly performed. These

models allow detecting some of the non-functional bugs in the embedded software (failure to enter a low-power mode, polling instead of explicit wait for an interrupt, ...).

Precise models: To validate the parameters of a power management policy and get the actual values for maximal temperature peaks and gradient, one needs a precise model. This implies precision in the timing of the platform, and on the thermal model of the chip. Some degree of precision is also needed to compare the efficiency of several power management policies.

We are interested in the last two. The following sections describe new tools to allow power and thermal modeling on SystemC/TLM at different levels of timing granularity. In both cases, the approach enriches a functional model with some power-state information, and cosimulates it with a dedicated power and thermal solver.

The rest of the paper is organized as follows. We review related work in Section 2.3, and then present our tools. Section 3 presents a cosimulation method implemented in LIBTLMPWT, where the thermal solver is embedded in the simulation. Section 4 presents our techniques to deal with loosely timed models without introducing simulation artifacts. In Section 5, we present a prototype distinct from LIBTLMPWT based on similar ideas, that allows cosimulating a SystemC simulation with an external power and temperature solver. We present our experimental results in Section 6, and conclude in Section 7.

2.3 Related Work

2.3.1 Power-state Model

The power-state model we use has already been applied to SystemC in the TLM Power library presented in [34, 55, 23]. TLM Power models a system-on-a-chip in SystemC/TLM, running the actual software on top of a simulated hardware. The objective is to validate a power management policy. We borrowed some ideas from TLM Power, but the latter does not allow temperature management. Another approach using the power-state model on SystemC programs is presented in [24], with advanced techniques for software integration like source-level simulation with back-annotations. We extended the idea to support cosimulation with a thermal solver, including closed-loop cosimulation where the software has access to non-functional values through sensors. Also, [34, 55, 23, 24] target precisely timed models while we allow an analysis on temporally decoupled or loosely timed models.

The power-state model is also used in [18] where the authors perform a thermal analysis to evaluate and optimize the mean time to failure of a chip. [18] uses an abstraction of the software as a task graph, unlike our approach which uses the concrete, actual software (i.e. a cross-compiled binary executable).

The power-state model can also be used at a more abstract level, where the hardware and the software are modeled with automata. Indeed, if the set of power-states is finite, then the underlying formal model is the one of *linear-priced timed automata* [4]. On a simple enough model, formal analysis is possible. For example, [19] formulates the thermal analysis problem as a hybrid automata reachability verification problem, and solves it using model checking. [32] defines a system-level *analytical* model to capture the consumption and thermal behavior of a chip with *Power Variability Curves*, based on the framework of real-time calculus. These approaches cannot be applied on a model precise enough to execute the actual software. Instead, the software is modeled as a set of tasks, and the hardware architecture is not detailed. Power consumption is considered to be a function of the executing software task, hence only processors are considered. [32] computes guaranteed bounds on temperature peaks. The method is intended to be used at a very early stage of the design. As opposed to this, we use SystemC/TLM models that *model* the

hardware, but *execute* the actual embedded software (using either instruction set simulators or source-level simulation [44]).

A concrete study of power-state models in Synopsys Platform Architect and Aceplorer is given in [22]. Unlike the present paper, [22] focuses on the power/thermal analysis and does not consider the cosimulation with a functional simulator nor software integration.

2.3.2 Low-Level Power Analysis

Our approach uses the power-state model, which assumes that the non-functional characteristics like power-consumption are given for each component and each power-state. Other techniques, or physical measurements, have to be used to find these values for each component. The power-state model is used to compose the results at the system-level. The following techniques are lower-level analysis techniques that can be used to calibrate a high-level power-state model.

[39] presents an analytical approach to dynamic power estimation of RTL descriptions. The authors use the concept of entropy (from information theory) to estimate the average activity factor. They consider only the combinatorial parts of a circuit.

The work described in [52] is a simulation method including the functionality and power consumption. It focuses on instruction-level power consumption for software execution, describing in details how to get the parameters of the power-model, with measures. It is limited to power-consumption and does not take temperature into account. A case-study from Intel is described, and the simulation results compared to measures on the real chip. The analysis is based on Hamming distance, and this information is not available in most TLM models because they abstract away details about transaction interleaving, signals switching and arbitration.

[13] provide a methodology for power characterization of the AMBA AHB communication Bus. The resulting power characterization is based on monitoring some parameters of the functional simulation. Accuracy of power estimation is gained at the expense of the functional model abstraction level, and therefore at the expense of the simulation speed. Like for [52], it relies on information that may not be available in TLM.

A technique for instruction level power modeling of processors is described in [51]. The intuitive idea is to run repeatedly one instruction in a loop and measure the average power. The work takes into account inter-instruction effect (switching from an instruction to another). Analysis of assembly code can then associate power values with each basic block of the program. The modeling approach targets software power optimization trough power-aware compilation. A similar approach is described in [42].

[12] describes a methodology for deriving instruction-based dynamic power models from gate-level simulations. The derived power models (in the form of look-up tables) are meant to be used in system-level simulation models to allow for faster power simulation.

Industrial tools like Synopsys's PrimeTime PX [49] allow a very detailed power analysis, but these tools take as input the netlist of the circuit, which is not available at early stages of the design-flow. They perform a detailed analysis hence run orders of magnitude slower than a TLM model.

Several approaches are dedicated to the definition of power-state machines of individual IPs, from data collected with low-level simulations. In some approaches, the structure of the machine has to be known in advance, and the analysis of low-level simulations provides the consumption values to be attached to the predetermined states. [17] proposes to run functional simulations of an IP together with gate-level simulations, in order to synthesize both the structure of the power state machine, and the consumption values attached to the states.

2.3.3 Thermal Analysis

Several thermal solvers are available. Hotspot [28] can be used either as a standalone tool or as a library. It takes as input the physical characteristics of the chip and a power-trace (power consumption of each component as a function of time), and computes a thermal trace. ATMI [38] follows a similar approach, with a focus on simplicity of integration. It is available only as a library. 3D-ICE [48] uses a similar model dedicated to 3D stacked chips. These tools do only thermal analysis: in Figure 2, they can be used for step "Temperature model", but need an additional cosimulation engine to perform the complete simulation.

CTherm [33] proposes a cosimulation including functional (SystemC), power and thermal (3D-ICE) models similar to ours (the thermal solver is triggered periodically from SystemC at a user-defined pace). It does not allow Direct Memory Interface (DMI), temporal decoupling or loose timing. However, it has some features like thermal checkpointing and automatic thermal model generation that would be interesting to add to our tools.

Aceplorer [31] is an industrial tool which, coupled with AceThermalModeler, does both power and thermal analysis (i.e. does both "Power model" and "Temperature model" in Figure 2). Initially, it could get some power-state traces from an external SystemC/TLM model, but not feed non-functional values back to the model. It did take into account the power/temperature loop (arrows (b) and (c) on Figure 2), but did not allow a feedback loop from the non-functional side to the functional model (arrow (d)). The AceTLMConnect [43] extension was added to the tool based on our research prototype (as part of the joint project HeLP[1]).

These thermal solvers solve several problems. They can perform a *steady-state* analysis (i.e., compute the state of the system after a long period of time with constant load), or *transient* analysis (i.e., model the response of a system to a change). The steady-state analysis is useful to analyze the overall characteristics of a chip, but is not sufficient to execute a software power management policy, hence is not sufficient to solve the problem addressed in this paper. A steady-state analysis does not need the detailed behavior hence it does not need our cosimulation technique. We use the *transient* analysis of these tools, and feed it with information produced by the functional simulation.

2.3.4 Standards and Formats

Some common languages/formats (e.g., UPF [29] and CPF [45]) for low power design are emerging and allow to describe power intents (power domains, power network, power switches, etc.) of a circuit. These formats can be the entry point of a code generation flow [36] that avoids hardcoding non-functional values into the SystemC/TLM code, and could use a cosimulation like ours to run the simulation.

2.3.5 System-Level Power and Thermal Simulation

A widely used tool for system-level simulation is gem5 [8]. The principles behind gem5 and SystemC/TLM are very similar: both are event driven simulators, and gem5 can actually cosimulate with SystemC. gem5 is meant to be usable out of the box, and includes several instruction set simulators and memory models, while SystemC only provides the building blocks to write such simulators. gem5 can be configured with different speed Vs accuracy trade-offs, but even the speed-oriented configurations are more precise than the loosely-timed models we are targeting in this paper. For example, gem5 does not support temporal decoupling.

[1] ANR Arpège, 2009-2013, see `http://www-verimag.imag.fr/PROJECTS/SYNCHRONE/HELP/`

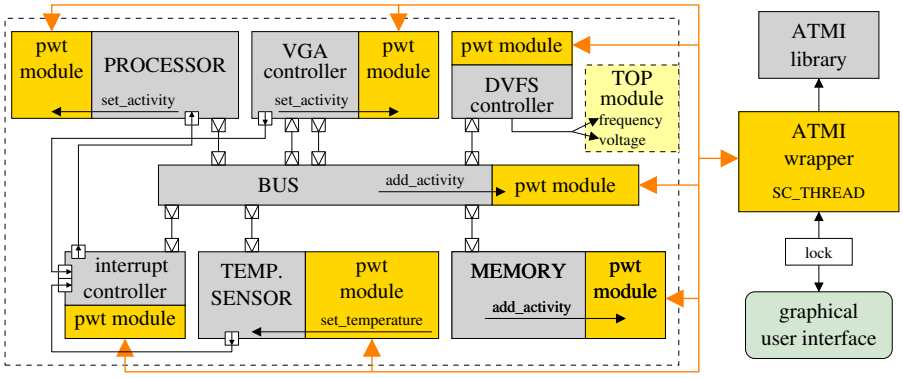


Figure 3 Example of a minimal SoC model with its power and temperature extensions.

gem5 can be augmented with a power model and cosimulate with a thermal solver like Hotspot as done in [50]. Similarly to the present paper, [50] allows executing software that interact with power-related sensors and actuators on the chip. However, the abstraction level is very different. Our techniques allow for common optimizations on loosely-timed models: coarse-grained timing with temporal decoupling (see Section 4.2), direct memory interface (see Section 3.3), including traffic models. Also, to the best of our knowledge, gem5 is not able to cosimulate with an external, black-box, power and thermal solver (see Section 5).

3 Standalone Power and Temperature Modeling

3.1 Architecture Overview

We now present the approach followed by LIBTLMPWT, in which all power and temperature computations are integrated into the SystemC/TLM model. As shown on Figure 3, each module contains code to estimate its power consumption, then a centralized temperature solver gathers power information and evaluates the temperature of each module. An optional graphical user interface (GUI) allows monitoring and controlling the simulation.

Temperature evaluation is done using the ATMI tool [38]. Basically, ATMI takes as input a floorplan, i.e. a list of areas described by their coordinates, and the initial temperature. During simulation, ATMI computes the temperature of each area based on its power consumption, expressed as a power density. This computation is done at a regular pace, such as once every millisecond (SystemC time). The temperature computation must be centralized, because the temperature of any area depends on the temperature of neighbor areas.

Since ATMI is packaged as a C library, it can be directly called from SystemC code. This is done by a SystemC module, called the ATMI wrapper. At elaboration time, we register to this module each SystemC module that is mapped to an ATMI area. This module contains a SystemC thread that calls the ATMI library according to the ATMI pace during simulation.

The ATMI wrapper algorithm is schemed by Figure 4. Given the power densities, computing the module temperatures is just a function call to the ATMI library. The resulting temperature is propagated to the SystemC side by the wrapper after the call to ATMI. For any SystemC module, it is possible to provide a callback method that is called each time the temperature is set. For example, the temperature sensor module defines a callback method that raises an interrupt when the temperature reaches some thresholds (for a concrete example of sensor providing this functionality, see the High/Low-Temperature Interrupt Enable bits of the `IA32_THERM_INTERRUPT` Register on Intel Architecture [30]). The main issue is to estimate the average power density consumed during the last step elapsed. This estimation is done in the new `pwt_module` class.

```
while (true) {
    wait(atmi_step_duration); // SystemC's wait
    for each module, compute its average power density during the last elapsed step.
    atmi_simulator_step(atmi_instance, power_densities) // call ATMI
    set module temperatures and call associated callbacks to trigger non-functional events
}
```

Figure 4 Basic algorithm of the ATMI wrapper.

In our model, the temperature transmitted to the SystemC modules, in particular to models of the thermal sensors, is the temperature at the corresponding instant. If the actual sensor exposes only a sampling of the temperature (e.g. the physical sensor samples temperature only every second), then this sampling can be implemented in SystemC by the temperature sensor. Letting the SystemC module chose the sampling rate makes the approach very flexible.

Each SystemC module that is mapped to an ATMI area must inherit from the `pwt_module` class. This class stores several parameters, categorized following the classification given in Section 2.2.1 (*fixed* parameters: floorplan and technology dependent parameters; *runtime* parameters: voltage, frequency, activity ratio).

In order to define frequency and power domains, we allow the existence of PWT modules not mapped on the floorplan. Such modules only provide the voltage and frequency parameters, which are forwarded to their children modules. Other methods are disabled; in particular, they have no power densities. For example, the chip on Figure 3 has one DVFS (*Dynamic Voltage and Frequency Scaling*) controller and a single power domain. So, in TLM, the model of the DVFS controller is bound to the top module; the DVFS controller TLM module calls the methods `set_frequency` and `set_voltage` of the top module, which in turn calls the `set_frequency` and `set_voltage` methods of all its children PWT modules.

Given these parameters, the `pwt_module` class computes the power density, or more precisely, its average value during the last step elapsed. The power consumption is computed as the sum of the static and the dynamic power:

- The *static* power is due to the leakage current, and it is proportional to the voltage and the leakage current intensity. The intensity itself increases when the temperature increases; in the current implementation, we use a linear approximation of the temperature effect, but a more elaborated physical model would be easy to integrate in the tool.
- The *dynamic* power corresponds to the cost of voltage changes in gates. It is proportional to the frequency, to the number of gates involved (i.e., the activity ratio), and to the square of the voltage. Moreover, it is proportional to a constant that depends on the capacitance per gate and on the gate density.

In consequence, the general formula is of the form:

$$P = P_{static} \quad + \quad P_{dynamic} = V \times K_1 \times (1 + K_2 \times T) \quad + \quad F \times V^2 \times \alpha \times K_3,$$

where V is the voltage, T is the temperature, F is the frequency, α is the activity ratio, and K_i are static parameters depending on the module area and on the synthesis technology. Because PWT modules contain the general formula, which involves explicitly the frequency and the voltage, the power model manages DVFS by construction. Indeed, other approaches [34] let model developers provide the actual power value, using their own function that may or may not take into account that the voltage or the frequency may change. If needed, a module that has a specific power model can also redefine the method that computes its power density and use an arbitrary formula written in C++.

3.2 Setting the Activity Ratio

Using the approach we present in this section, the main task to extend a TLM model with power consumption and temperature estimations is to set dynamically the activity ratio of each module. The `pwt_module` class provides two ways to set this ratio (`sc_time_stamp()` is the SystemC function returning the current simulated time):

1. *level-based: change the activity level, until the next call*
   `void set_activity(float ratio, sc_time now = sc_time_stamp())`
   Set the activity ratio starting from *now* and until another level is set.
2. *action-based: add some extra-activity for a fixed duration*
   ```
   void add_activity(float ratio_increment,
                     unsigned nb_cycles,
                     sc_time now = sc_time_stamp())
   ```
   Add some activity to the current level (more precisely, the level at date *now*), for a short duration defined by a number of clock cycles.

In general, the first method is best suited for initiator modules whereas the second is better for interconnects and target modules. For example, a processor TLM model will call the `set_activity` method when it enters an *idle* state and when it becomes *busy* again. Using this method, we get an activity-state based power model, as in [34]. If more accuracy is needed, one can develop an instruction-based power model (like the one of [20]) on top of this API. The model would use the second method and call `add_activity` for each instruction, with a ratio depending on the instruction kind and the register values. Obviously, the second approach requires additional manpower and will slow down the simulation.

Concerning interconnect and target modules (that receive transaction initiated by others, like a memory component), the best solution is to call the `add_activity` method once per transaction. In general, the *ratio increment* depends on the command (`READ` or `WRITE`) whereas the duration (`nb_cycles`) depends on the transaction size. Note that it would be harder to use the activity-state based method, because there is no local activity state (indeed, the activity depends on the external initiators).

A naive implementation of the `add_activity` method would be to use two calls of the `set_activity` method, as follow: remember the current activity level as `current_ratio`, set the activity level to `current_ratio + ratio_incr`, increase the local date, and finally re-set the activity to `current_ratio`. However, this implementation would not be reentrant, and pretty slow. Reentrancy is mandatory for a bus or memory module: they can receive transactions from several initiators whose non-functional effect overlap. On the contrary, our implementation of the `add_activity` method is reentrant and fast.

The rationale for the parameter "`sc_time now`" of methods `set_activity` and `add_activity` is to ensure the compatibility with temporal decoupling [53], when using the coding rules defined in [1]. When temporally decoupled, the local date of a process is expressed as "`sc_time_stamp() + local_offset`" (instead of "`sc_time_stamp()`"), allowing to advance the local time by executing a low-cost "`local_offset += T`" instead of a costly "`wait(T)`". This local offset is part of all transactions, so it can be used by interconnects and target modules too. Consequently, to set the activity ratio at the right date, the methods `set_activity()` and `add_activity` must be called with the parameter `now` set to "`sc_time_stamp() + local_offset`". Since this parameter has a default value, it can be safely ignored for all processes that are not temporally decoupled.

The "`now`" parameter passed to the activity methods may be further in time than the next ATMI step boundary. Indeed, some TLM modules modeled at a coarse-grained may simulate a slice of time much longer than the ATMI time step before yielding back to the scheduler.

Consequently, each PWT module contains a list of activity counters. The list head contains the activity counter of the current ATMI step, and successive list elements store the activity of future steps. The ATMI wrapper pops the front element once every step. The traffic model of [9] can be implemented on top of this, but it is currently not part of the tool presented here.

3.3 Direct Memory Interface (DMI) Management

Motivated by simulation speed issues, some TLM modules use a technique called *Direct Memory Interface* (DMI). The goal is to accelerate memory accesses. and the idea is to provide the initiator (e.g., an ISS) with a pointer to the memory array. So, when accessing memory, the initiator will directly use the memory pointer instead of generating a transaction. Given that a transaction involves many indirect function calls plus routing in the interconnects, the speed gain is significant. DMI is functionally correct but may bypass some side-effects, because the code related to timing and power into the interconnects and the memory is no longer executed. For the timing issue, the SystemC standard [1] suggests to provide the initiator with two durations: the read latency and the write latency. Thus, the initiator can add the latency to its local offset when a memory access is simulated. Because the latencies depend on the frequency, the *DMI descriptor* must be updated every time the frequency is changed.

We use the same idea for power modeling. However, providing a single activity ratio increment per transaction is not enough, since the activity increment must be added to each module involved in the transaction. Indeed, if the additional activity was assigned to the initiator, then the initiator temperature would be overestimated whereas the target components (e.g. bus and memory) temperatures would be underestimated. Our solution is to add into the DMI descriptor a pointer list of all PWT modules that are on the transaction path. Our DMI manager class provides a method `apply_side_effects(command, size)` that increases the local time offset according to the latency and call the `add_activity` method of all the PWT modules in the pointer list.

3.4 Graphical User Interface

We have developed a graphical user interface, providing basic simulations controls and some monitoring features (see Figure 5). This GUI is implemented using the Qt framework, and run in a POSIX thread distinct from the SystemC simulation. So, the SystemC simulation is slowed down only when the GUI takes the main lock to update its values. Those value updates are done ten times per second (wall-clock time).

On the control side, the GUI allows to pause the simulation (done by keeping the lock), or to reduce the SystemC simulation speed (done by adding Unix `usleep` calls in the Posix thread running the SystemC kernel).

For monitoring, the GUI provides the current temperatures as text values, a graph of the floorplan where each module is coloured with respect to its temperature (from blue for cold, to red for hot module). Additionally, a plot provides either the temperature or the power consumption or the power density of each module depending on the time.

4 Thermal Analysis for Loosely Timed Models

4.1 Loosely Timed Models

Loose timing requires special attention. An obvious limitation of any power-state based model is that if the timing is too imprecise, then a precise power analysis is not possible (as the power-state model intrinsically needs timing to integrate the power values over time). Still, the timing can be loose and be a reasonable approximation of reality.

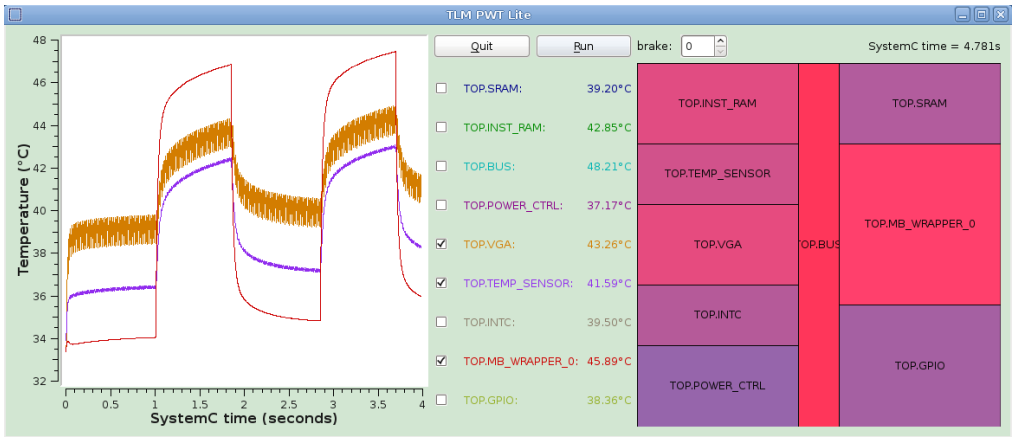


Figure 5 Graphical user interface.

In fine-grained models, the duration of a task execution is usually *computed*. For example, an ISS is a way of computing a software task duration: the real binary (and thus the real algorithm) is simulated instruction by instruction and memory access per memory access; the duration of the task is then a sum of short durations corresponding to each instruction execution and memory access. These short durations are defined in many modules: instruction cost in the ISS module and memory access latencies and bandwidths in interconnect and target modules.

As opposed to this, loosely timed models can abstract durations completely and consider that computations are done in zero-time, or use *calibrated* durations. This happens for example when the algorithm used by the future hardware is still unknown, or when the simulated platform uses an algorithm that is different from the hardware. For an image processing application, for example, a coarse-grained model that considers an image as atomic can be calibrated with some timing values close to the actual ones (e.g. considering that decoding one image takes X ms). This estimated duration may possibly depend on some parameters, such as the image size.

Calibrated models cannot be very precise, as the actual performance can depend on interactions that cannot be taken into account in calibration (e.g. conflicts on a shared bus). However, using calibrated models is not less precise than using hand-written scenarios, which is a common practice for power/thermal modeling in the industry. This section presents a set of techniques to write approximate power and thermal models based on loosely timed functional models. We obviously cannot recover information which is abstracted away by loose timing, but we avoid simulation artifacts that would result from it.

In this section, we consider the case of coarse-grained modules; that is to say, a module which contains abstract tasks whose duration is declared instead of computed. A typical example is an hardware accelerator for graphical computation, such as image encoding or decoding. Such TLM module models the functionality using an algorithm (such as the legacy code algorithm) which is not the same as one used in the hardware.

The power consumption of a coarse-grained module can be defined easily by a discrete set of activity states: for example one for idle and another for busy. However, because a coarse-grained module performs memory accesses, it has an impact on memory and interconnect consumption that must also be evaluated. Figure 6 shows a typical code for a coarse-grained module task.

More generally, the code is usually written as a sequence of computations, each of them being of the form `compute(); wait(...); commit();` . `compute()` is done in zero simulated time. `wait(...)` lets the simulated time elapse. Its argument is the time the computation would take on the real system. `commit()` is the functional effect of the computation, typically an interrupt or a transaction

```
1  void compute() { // SC_THREAD
2      while (true) {
3
4          wait(start_event); // wait for a new request
5
6          in_data = read_block(...); // read all useful data (e.g., 1 image)
7          out_data = compute_task(in_data); // call legacy code
8          write_block(out_data, ...); // write all new data
9          wait(duration_task); // wait the duration task
10         end_event.notify(); // notify the client that the request is done
11
12     }
13 }
```

Figure 6 Example Code with Loose Timing.

to a control register. `commit()` is also called a *synchronization point*. More details can be found in [15].

With a naive implementation, the `read_block` and `write_block` transactions are done at a single SystemC time, thus creating an infinite peak of power consumption in the interconnect and memory, followed by no additional consumption during `duration_task`. As a consequence, in the model, the chip temperature will climb very fast and immediately fall back as fast. This temperate peak may cross a threshold of the temperature sensors, thus generating an event for the temperature manager. This event is an issue, because on the physical system there is likely no such temperature peak, but instead a slow increase of the temperature spread among the whole task execution.

Avoiding unrealistic peaks is relatively easy on the initiator's side. The solution is to instrument the complete `compute(); wait(...); commit();` block of code. In the example of Figure 6 this would mean adding calls to `set_activity()` at lines 3, 5 and 11.

4.2 Transactions from Coarse-Grained Modules

To illustrate the problems caused by loose timing, Figure 7 shows a possible execution of two iterations of the loop. Vertical arrows represent transactions. The first line shows the transactions that the actual system would perform. The second line shows how these transactions are simulated on a loosely-timed model. In a naive model (Figure 7.(a)), the dynamic energy consumption of the bus routing these transactions would be modeled at the simulated time when the transactions are executed, hence we would get an instantaneous energy consumption at SystemC instants (for clarity, the figure shows cumulative energy instead of power, which would be infinite). The temperature model would therefore compute a peak that does not exist on the real system.

If the model uses a precise temporal decoupling, then the local dates of transactions can be used instead of the global SystemC time. However, on a loosely timed system, this local clock only provides a lower bound of the transaction start: the actual time of the transaction may be at any time between the previous timing annotation and the next one, which is not known yet. In the example of Figure 6, the TLM model has a clear read/compute/write separation, but the actual system may use a pipelined algorithm where reads, computations and writes are interleaved. This cannot be reflected by only adding annotations to the code. The best we can do is to assume that transactions will be evenly distributed on the interval of time where they happen. This is illustrated on Figure 7.(b): the analysis counts transactions over each interval of time, and when

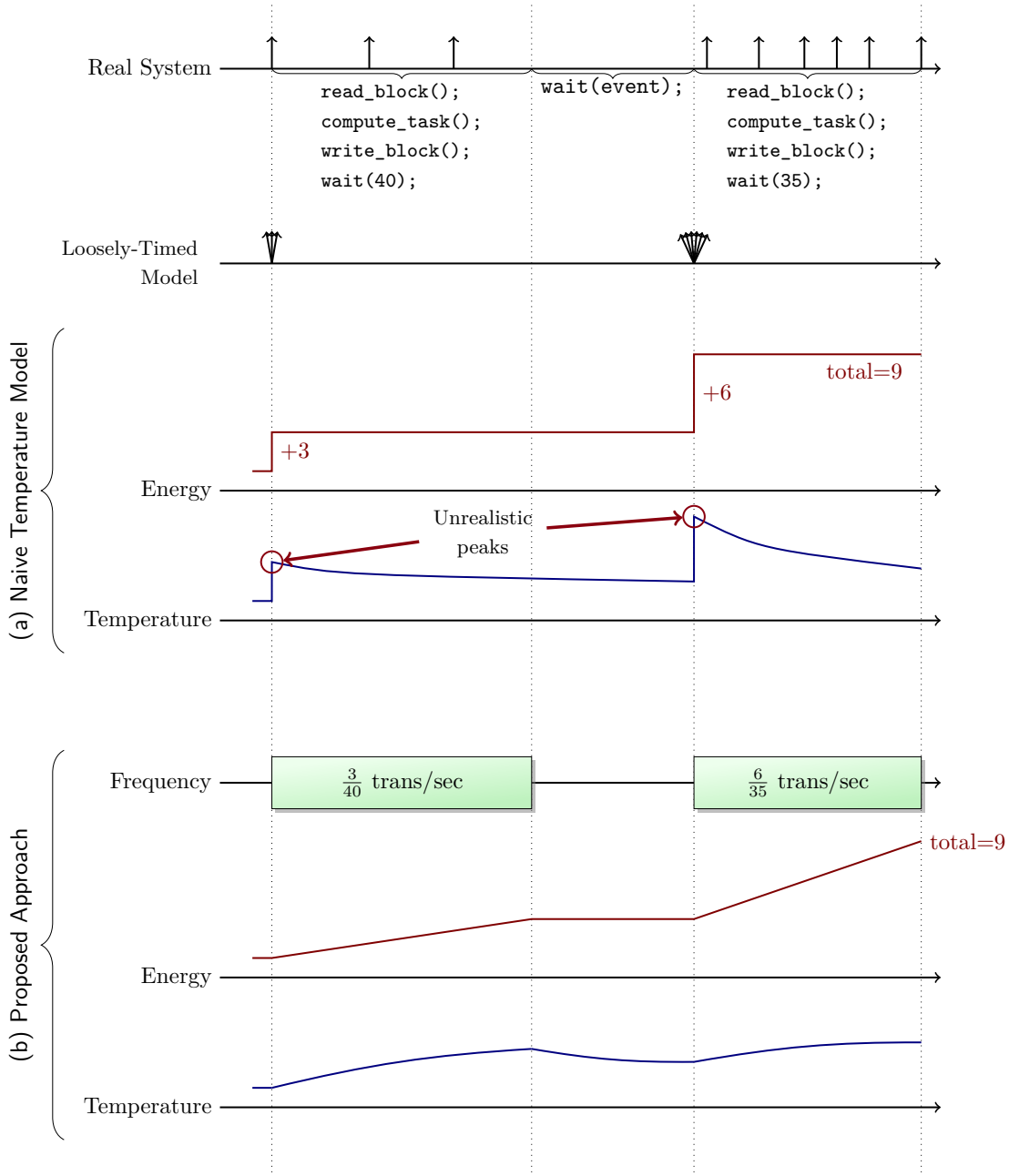


Figure 7 Elimination of Simulation Artifacts.

reaching a synchronization point, a frequency is computed, and this frequency is used instead of individual transactions in the analysis.

We presented a first implementation of this principle in [9]. We describe below our new implementation of this principle integrated in LIBTLMPWT, with an accurate management of classical optimizations like DMI.

Firstly, interconnect and targets modules (mainly memory modules) must be able to recognize such coarse-grained transactions. For this goal, we have defined a new extension type (following the ASI TLM guidelines for ignorable extensions). When a module receives a transaction with this extension, its behavior is changed as follow:

- the code to add a delay to the initiator local date is skipped
- the extra-activity due to transaction is stored until the end of access date is known.
- the current module is registered to a *delayed-access manager*.

Secondly, when the task is done and its execution duration is known, the coarse grain module informs the delayed-access manger, which in turn calls back each module currently storing delayed accesses. Actually, this is done by adding a line `DelayedActivityManager::commit(task_duration)` just before the statement `wait(duration_task)`. When interconnect and target modules are called back, they simply spread the stored activity between the start of access and end of access date.

Additionally, the new extension for coarse-grained transactions allows to define a secondary transaction size. The idea is that because the algorithm is not the real one, the access size in the physical chip may be different from the one of the TLM platform. For a simple example, consider a hardware module computing the next step of the Game of Life. In TLM, the implementation will read then write the whole image. However, a hardware implementation will have to read some pixel many times if it cannot buffer the whole image; the most naive implementation would even read each pixel 9 times. On the write side, if the image is modified in place, then the hardware may decide to write only pixels that change. Thanks to the secondary size provided with the extension, the extra-activity can be computed on a better estimated size without changing the functional model.

4.3 Interrupt Management

As explained in Section 4.1 and [15], a loosely-timed model can still be functionally faithful in the presence of interrupts. For example, in Figure 6, in the execution depicted on Figure 7, if an interrupt is received by the component executing `compute` at time 20, then in the real system, the interrupt may abort any of `read_block()`, `compute_task()` or `write_block()`. In the loosely-timed model, these 3 functions are executed in zero-time. The interrupt is received during the call to `wait(duration_task)`, and a check for pending interrupts is done after this `wait` statement. The interrupt is therefore processed after the `wait` statement terminates. In some sense, it is taken into account later than the real system, but the model is still faithful because the interrupt is received before `end_event.notify()`, hence the functional effect of the computation interrupted is not yet visible to other components. It is therefore acceptable to take the interrupt into account at the next SystemC instant (at time 40 in our case) from the functional point of view.

However, from the non-functional point of view, interrupts in loosely-timed models raise several issues: even though we can manage them in a functionally faithful way, the timing of the SystemC simulation does not match the one of the physical system. Consider a component executing `compute(); wait(50); commit();` starting at time t, to model a computation that takes 50 units of time. Two cases are problematic:

1. If the processor receives an interrupt at time $t + 20$, and executes an Interrupt Service Routine (ISR) `isr()` for 10 units of time at this point. The functional model executes `compute()` and then `isr()`, but we need to model `isr()` as being executed in interval $[t + 20, t + 30]$.
2. If the processor receives an interrupt at time $t + 20$ which aborts the computation, then `compute()` is executed completely in the functional model, and only a part (20/50) of its execution must be taken into account in the non-functional analysis. In other words, we do not need to actually cancel the functional computation because its functional effect is not yet visible, but we must not consider the power consumption of a computation that is not done in the real system.

To solve issue 1, we consider ISR as a special-case of software execution. First, we need to execute the ISR at the right point in time, hence checking for the presence of pending interrupt after the `wait` statement is no longer acceptable. Our approach replaces the call to `wait(time)` between a computation and its functional effect with a `wait(event, time)`, that either completes when enough simulated time has elapsed, or can be interrupted by an event triggered by an interrupt. This way, the interrupt service routine can be executed at the right point it time. The frequency of transactions due to the normal computation is kept unchanged, but is not taken into account during the execution of the ISR.

To solve issue 2, we need to instrument the abortion in the embedded software, and when we encounter a task abortion, we reset the frequency for the processor (technically, this is implemented by instrumenting `longjmp` to write to a magic address that is caught by the SystemC model and transmitted to the power-model). This is implemented in the tool presented in [9] but not yet in LIBTLMPWT.

5 Cosimulation with an External Power and Temperature Solver

5.1 Cosimulation Interface

When using the non-functional solver as a black-box it is no longer possible to execute a SystemC process for each of its internal steps, since we cannot know when the steps should take place. In this case, another cosimulation strategy has to be applied.

We implemented a cosimulation interface, presented in details in [10], that allows running a functional SystemC/TLM platform with power annotations with an external non-functional solver running in a separate process. We did our experiments with wrappers around ATMI [38] and Hotspot [28] as external solvers, but the same interface was used to connect to the industrial tool Aceplorer. This interface is now used by the commercial extension AceTLMConnect [43] for Aceplorer. A small case study using the industrial implementation is presented in [16].

The cosimulation interface uses a simple request/response protocol, using Thrift [46] for interprocess communication. The principle is the following: the simulation starts on the SystemC side, and SystemC's time drives the simulation. From time to time, the SystemC program requests a non-functional simulation on a time interval. The non-functional solver performs the simulation and returns the relevant values at the end of the time interval. In some case, the non-functional simulation may return early, because the temperature crossed a given threshold, and a SystemC event can be generated at the time the threshold is crossed.

5.2 Strategies Using the Interface

The same interface can be used with multiple strategies. In the *lockstep strategy* (Figures 8 and 9), the functional/non-functional synchronization is performed at the end of each SystemC simulated instant. A non-functional simulation is requested for the time interval between the current instant

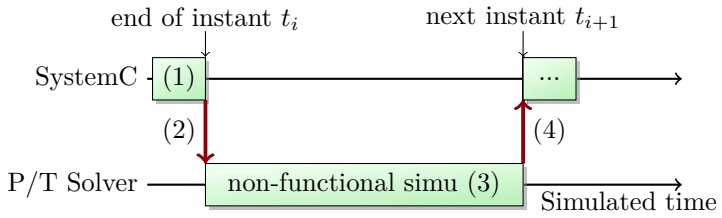


Figure 8 *lockstep* cosimulation strategy (for clarity, simulated instants are represented with a non-null width; boxes on the SystemC line correspond to simulated instants and boxes on the P/T solver line to intervals between instants).

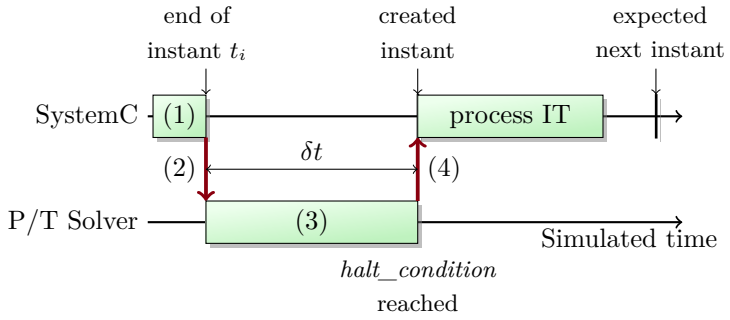


Figure 9 *lockstep* cosimulation strategy with interrupt (IT).

and the expected next simulation instant. In case a non-functional event is triggered (Fig 9), the non-functional simulation stops before the end of the requested time interval, and a SystemC event is notified, which creates a SystemC instant during which the functional part can react (e.g., by triggering an emergency stop if the temperature is too high).

When one can guarantee that no non-functional event is possible (either because the hardware sensors are only passive components and cannot trigger events, or because we know for sure that the condition will never be met), another strategy is possible: the *functional-ahead strategy*, illustrated in Figure 10. In this case, the SystemC side runs without launching the non-functional simulation until an access to a sensor is performed. When this happens, a non-functional simulation is requested up to the current SystemC instant. This strategy is less flexible, but requires considerably less request/response round trips, hence can be faster (especially when SystemC and the non-functional solvers run on different machines).

A *parallel strategy* is also presented in [10].

6 Experiments

6.1 Development Cost

For the development and the evaluation of the LIBTLMPWT approach, we have developed a demonstration platform based on a small FPGA system. The main components are a processor (MicroBlaze) and a VGA controller. There are two memories, one for instructions and the other for data, plus the usual devices: timers, UART, interrupt controller, etc. Compared to the initial FPGA system, we have added in the TLM model a temperature sensor and a DVFS controller. The whole TLM model uses the blocking TLM interfaces of [1], with the generic payload plus an ignorable extension for DMI configuration. We reuse some open-source TLM code from SoCLib [47] and SimSoC [26].

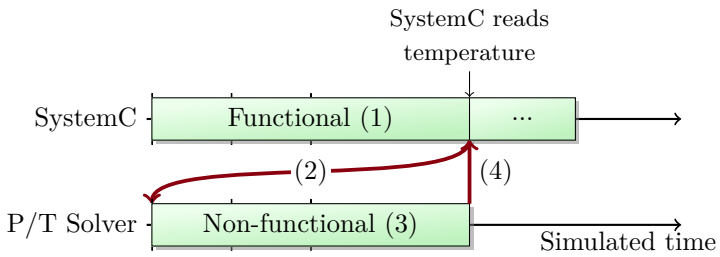


Figure 10 *functional ahead* strategy, in the absence of interrupt.

We have applied some classic optimizations in the TLM model, so that the base simulation speed is similar to the simulation speed of an industrial TLM model. In particular, we use *temporal decoupling* in all places where it is useful; the processor and VGA controller models use the *Direct Memory Interface* mechanism. When the processor is busy, the simulation speed is around 50 MIPS (million instructions per second).

The TLM model without power and temperature counts 5000 lines of code, and uses some general development kits counting in total 1400 lines of code. In this version, the temperature sensor and the DVFS controller modules are included but they have no behavior.

For the instrumentation of the platform itself, we have added about 100 lines of code. Note that this is quite small compared to the platform size, showing that once the tools and data are available, instrumenting an existing TLM model for power and temperature estimations requires a very little cost. One must provide the physical values used in the power and thermal model; this calibration task is out of this paper scope.

The core classes we have developed for power and temperature modeling counts 700 lines of code (not including the ATMI library, which is 2700 lines long). Additionally, the graphical user interface counts close to 600 lines.

6.2 Simulation Speed Overhead

To evaluate the simulation speed, we use our demonstration platform and make it run a benchmark application. In this benchmark, the processor is periodically computing: it waits one second and then computes during about 0.8 seconds (SystemC time). The application computes the "game of life", waiting 1 second between images. Additionally, the VGA controller is active and loads the image buffer 60 times per second. Between two consecutive reloads, the VGA controller remains idle for a few milliseconds.

The first time the model is simulated, the ATMI library computes some data in advance in order to accelerate the simulation itself. Those data are cached in a file for future simulations. Modifying the floorplan or some technology dependent parameters requires to compute this file again. This computation takes about two minutes.

Simulating 10 seconds (SystemC time) takes:

- 3.4 seconds (wall-clock time) for the initial functional TLM model
- 6.4 seconds with power and temperature estimations (6.6 seconds with GUI), assuming that the ATMI cache file was ready.

So, the simulation duration overhead is about **+88%**. We consider that it is a significant but acceptable overhead: the performance remain in the same order of magnitude, and it should be noted that the LIBTLMPWT approach is compatible with the common TLM abstractions and optimizations, which allowed gaining several orders of magnitude compared to lower-level models like RTL.

Table 1 Effect of the ATMI step duration in LIBTLMPWT.

ATMI step	simulation duration	standard deviation (σ)			
		processor	VGA	bus	temp. sensor
0.25 ms	15.3 s (+139%)	*reference*	*reference*	*reference*	*reference*
0.5 ms	8.9 s (+39%)	0.01 °C	0.05 °C	0.09 °C	0.00 °C
1 ms	6.4 s (*ref.*)	0.03 °C	0.11 °C	0.12 °C	0.01 °C
2 ms	5.2 s (-19%)	0.06 °C	0.23 °C	0.15 °C	0.03 °C
4 ms	4.6 s (-28%)	0.13 °C	0.42 °C	0.19 °C	0.05 °C

For example, DMI would need to be disabled if we did not take it into account. As expected, running the PWT simulation with DMI but without the extension is quick (≈5 seconds) but incorrect: we have observed errors of more than 1 degree Celsius. If we disable DMI, then the functional simulation consumes 9.1 seconds and the PWT simulation consumes 12.8 seconds. In other words, without our extended DMI, the total overhead would be 9.4 seconds, i.e. +276% (5.7 seconds for disabling the DMI plus 3.7 seconds for power and temperature computations).

Looking at the profile obtained with `callgrind`+`kcachegrind` [54], we notice that there are two performance-consuming spots: 1. computations internal to the ATMI library (≈28% of total simulation time), 2. applications of transaction side effects when using DMI (≈12% of total simulation time). The `pwt_module` class has been implemented with the optimization of this second performance-consuming spot in mind. These numbers show that the performance overhead comes mainly from the temperature simulation (about 2/3 of the overhead), not from the interfacing (about 1/3 of the overhead).

Concerning the time spent in the ATMI library, the user may optimize it by adapting the ATMI step duration. The values above are given for a step duration of one millisecond. As shown by Table 1, the longer the ATMI step is, the faster the simulation, at the cost of a loss of accuracy. The temperature error is higher in modules whose power density changes at a fast pace.

6.3 Applications

Using the "game of life" benchmark previously presented, we can observe that temperature plots are as expected. Figure 11 shows those plots at different time scales. Looking at a short time range, we see that the VGA temperature fluctuates with an amplitude slightly above 1 °C; as a consequence, the temperature sensor fluctuates too, but with a smaller amplitude. On the second plot, we see that the processor temperature fluctuates at a slower pace, since it is computing about one second every two. Moreover, other module temperatures evolve according to the processor temperature. Finally, the third plot shows that the whole system takes about 100 seconds to reach its maximum temperature. It is one reason why simulators must be fast enough: several minutes of SystemC time can be needed to observe the relevant behavior.

One possible application of our tool is to detect non-functional errors in embedded software, such as polling a device register instead of using idle mode and interrupts. Figure 12 shows what happens if the previous benchmark uses polling instead of interrupts. The functional behaviour is exactly the same, but we immediately see that the temperatures keep increasing and that the real chip would overheat. Note that the bus temperature is higher during polling than during frame computation due to the high polling traffic. The bug is obvious with temperature analysis, even if the analysis is imprecise. It would be much harder to find without it.

Another application is the development and validation of the voltage and frequency management. One simple solution to avoid overheating is to switch between two modes: a default fast mode

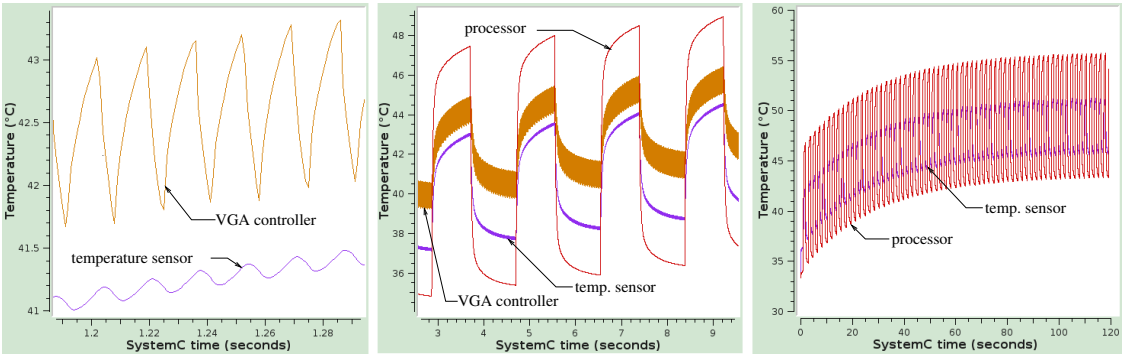


Figure 11 Temperature plots for the "game of life" benchmark, with different time scales.

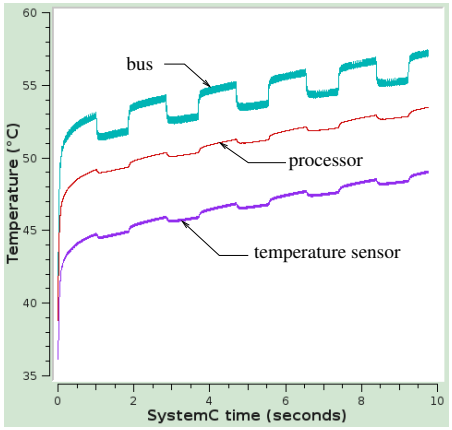


Figure 12 Variant of the "game of life" benchmark using polling.

where frequency and voltage are high, and a backup low-power mode where voltage and frequency are low. The power manager (i.e., a part of the embedded operating system) programs the interrupts of the temperature sensor module according to two thresholds: the high threshold is used to avoid overheating and causes the switch to the low-power mode, whereas the low threshold determines when to switch back to the fast mode. Testing this algorithm on a pure functional TLM model is impossible (see discussion in Section 2.2.3). It would require at least a scenario-based model, but in this case, temperature sensor interrupts would occur at unrealistic dates which would make the test more difficult. On the contrary, using the extended TLM model and its GUI allows to check easily the power manager's behaviour, as shown on Figure 13. Again, note that the simulation must be at least 20 seconds long to be useful, which shows that simulation speed matters.

Such temperature management based on low and high thresholds has some drawbacks; one is that the frequent temperature changes may raise the failure rate of the system [56]. Another approach is to use a PID controller. We have implemented this approach in the embedded software of our demonstration platform. As shown by Figure 14, we see that the PID-controlled temperature curve is smoother than the previous threshold-managed temperature curve. The first plot shows a simulation with a badly tuned PID controller, where the VGA controller temperature oscillations are amplified instead of smoothed, meaning that the gain parameters are likely too high.

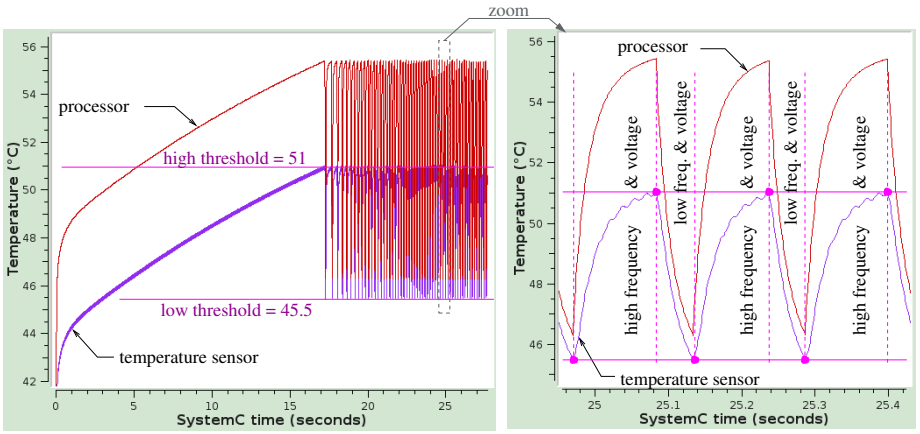


Figure 13 Temperature management with *low* and *high* thresholds: the software manager toggles between low and high power modes according to the temperature.

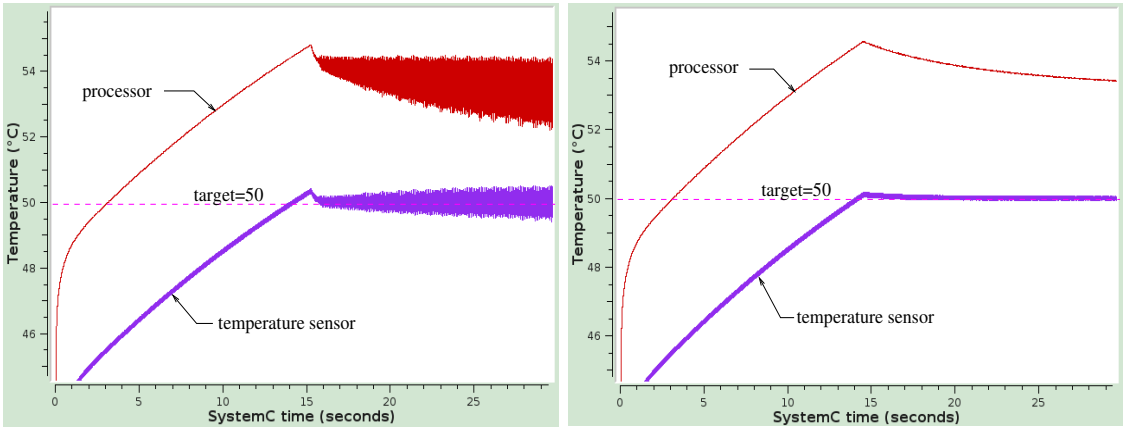


Figure 14 Temperature management using a PID controller, with distinct gain parameters.

6.4 Influence of Loose Timing on Performance and Precision

This section describes cosimulation techniques implemented in the tool presented in [9, 10], but not yet in LIBTLMPWT.

6.4.1 Granularity of the functional models

Figure 15 illustrates another experience with power consumption and temperature of the MEMORY component. This experiment was done using the cosimulation approach presented in Section 4, distinct from LIBTLMPWT, hence the performance results are not directly comparable with the previous sections. The functional behavior of the SoC is such that the memory receives a lot of traffic in the first half of the simulation period represented, and nothing in the second half. Fig. 15-(a) is the instruction-accurate simulation, on which the effect of this two-phase behavior is clearly visible: the temperature increases because of high power consumption, and then decreases slowly. Both Fig. 15-(b) and Fig. 15-(c) are obtained with a coarse-grained simulation, but we can see that only (c) reproduces the profile of (a). The difference is the following: in (b), the simulation is made in one big step (because there is no simulation instant in the middle), and the power consumption is spread over the whole interval; in (c), the simulation is split into two

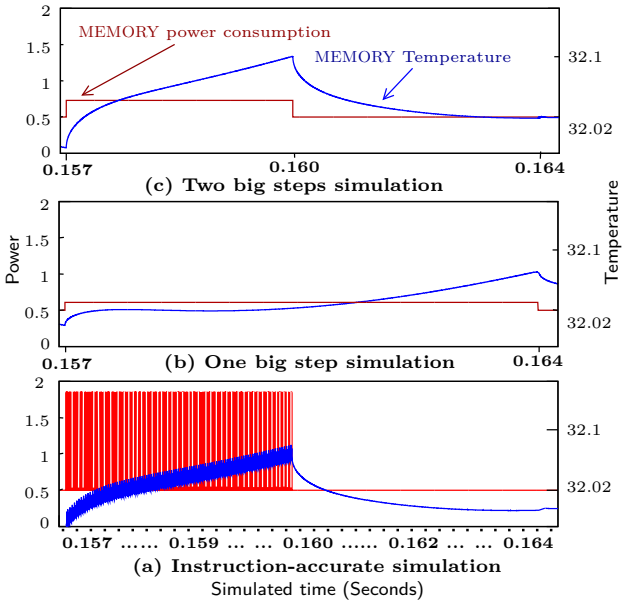


Figure 15 The effect of coarse granularity on memory power consumption.

simulation intervals. What happens is that in (b) the *functional and timed model* itself is too coarse. If the memory receives traffic in two well-differentiated patterns, it is probably because one of the components has two distinct running modes that have been ignored in the model. It means that the modeling of a component (like the software) can benefit from an explicit distinction between running modes, even if the impact is not on the *activity* model of component itself, but on the *traffic* model of another component.

6.4.2 Simulation speed

Table 2 is a summary of simulation speeds of the tool presented in [9, 10]. We distinguish the time taken by the SystemC part (SC), the ATMI part, and the connection between them (between parentheses is the number of exchanges between the simulators). The first line corresponds to an instruction-accurate simulation, the second one to simulation where we execute 100 instructions in a row, and the last one is the coarse-grained simulation based on logical synchronization points and explicit `purge` statements. We observe a factor 7 between the first and the third ones. It is also interesting to see that a lot can be gained on the side of the temperature simulator. ATMI is a fixed-step simulator, and we may obtain better results with a variable-step simulator. Note however that ATMI already uses a well optimized model, pre-computing the temperature response to a pulse, and using a sophisticated optimization called *event compression* [38]. Moreover, ATMI does not allow to ask for the evolution of temperature on time intervals smaller than $1\mu s$. When we execute the SystemC model one instruction at time, it represents a time interval of around 20 ns. It is useless to call ATMI for each of these small steps; a cache mechanism could be implemented in the connection, to call ATMI only when several steps with a total of more than $1\mu s$ have been executed on the SystemC side.

6.4.3 Interrupt Management

In Section 4.3 we explained how to model the abortion of a computation due to an interrupt faithfully with respect to power consumption. Figure 16 illustrates the effect of the method on

Table 2 Execution times and contributions of the simulator parts, for simulating 0.5s of the system.

	Time spent in				
	Total	SC	ATMI	Connection	Number of exchanges
1-inst.	1028s	48.8%	41.2%	11%	15.7E+6
100-inst.	185s	5.2%	94.5%	0.03%	0.15E+6
coarse-grained	139s	5%	95%	$\approx$ 0%	128

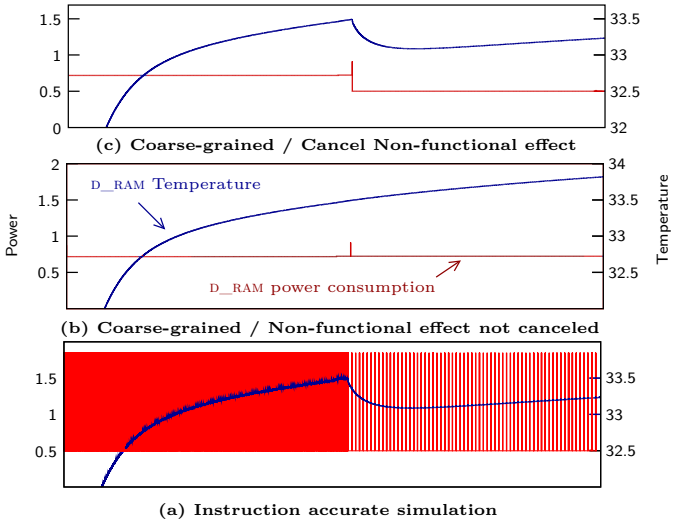


Figure 16 Modeling abortion.

MEMORY's temperature and power, due to the traffic it receives. We use a variant of the case-study where the power management policy just checks whether the temperature reaches the upper threshold and reacts by canceling the current operation of the software; this results in a much lower traffic on the memory. Fig. 16-(a) is the instruction-accurate simulation, which shows a high power consumption before the interrupt, and then a much lower one. Fig. 16-(b) is what we obtain with a coarse-grained simulation if we do not implement our method for modeling abortion faithfully. Fig. 16-(c) is what we obtain if we do implement it: we correctly observe that the temperature decreases when some functional behavior is aborted, resulting in a lower power consumption on the memory.

7 Conclusion

We presented several methods for cosimulation of a functional SystemC/TLM model with a power and a thermal model. LIBTLMPWT is a lightweight, integrated solution, that embeds ATMI as a temperature solver. It is available publicly as free software [27]. The tool presented in [9, 10] allows a cosimulation with an external solver, using inter-process communication to communicate with it.

Obviously, none of these tools allows recovering precision that was lost by raising the level of abstraction. If a component has a complex access pattern to a bus, and this pattern is not modeled, then none of our techniques can accurately model it. This is not surprising, as this would require *guessing* instead of *modeling*. Our contribution is, given a possibly loosely-timed functional platform, to provide modeling tools to create a non-functional model that is as accurate as it can be given the abstraction level of the functional platform.

The validation of these models is a difficult task. Ideally, we should compare the result of the models with a real system, but this is much harder than it seems. Measuring the temperature precisely can be done only on a chip without its packaging, which cannot run at full speed without overheating. This is a non-trivial task for silicon manufacturers, and clearly out of reach for an academic laboratory. Power-consumption measurements of individual components would require an instrumented version of the chip (with more pins than the actual system). Instead of validating our models against the real system, we compared several models at several levels of abstraction, considering the lower-level ones as the reference. Indeed, our contribution is not to provide power and thermal models, but to cosimulate them with a SystemC/TLM model. In other words, we assume the availability and accuracy of a non-functional model, and plug the input and output of this model on a functional simulation.

We experimented on a small but representative platform containing both hardware IPs and a processor. It would be interesting to experiment on a larger platform containing a large number of hardware IPs and/or a large number of processors (like many-core embedded processors or high-end general-purpose processors for data-centers where power and temperature are also important concerns). We do not foresee any fundamental issue with our cosimulation techniques: both the SystemC/TLM functional simulation and the power/thermal solvers we use are already used industrially at very large scales, and the cosimulation itself adds only a small overhead.

Using ATMI limits us to 2D designs. We are currently extending the interface to support Hotspot [28], which is able to manage 3D chips. It would be interesting to test other thermal solvers such as 3D-ICE [48]. Also, we focused on the cosimulation scheme, but a more integrated design-flow can be envisioned, where dedicated tools would be used for individual components, and the result used in the generation of our model. This is currently easy in theory but partly manual.

We are now working on extending the approach to support loose power and temperature annotations: when the precise values are not known, allow specifying an interval instead of possible values for each parameter.

References

1 Accellera Systems Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2011. URL: `http://www.accellera.org/`.

2 Mazhar Alidina, José C. Monteiro, Srinivas Devadas, Abhijit Ghosh, and Marios C. Papaefthymiou. Precomputation-based sequential logic optimization for low power. *IEEE Trans. VLSI Syst.*, 2(4):426–436, 1994. `doi:10.1109/92.335011`.

3 Andrea Bartolini, Matteo Cacciari, Andrea Tilli, and Luca Benini. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *IEEE Trans. Parallel Distrib. Syst.*, 24(1):170–183, 2013. `doi:10.1109/TPDS.2012.117`.

4 Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2001. `doi:10.1007/3-540-45351-2_15`.

5 Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In Anantha Chandrakasan and Sayfe Kiaei, editors, *Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998, Monterey, California, USA, August 10-12, 1998*, pages 173–178. ACM, 1998. `doi:10.1145/280756.280881`.

6 Luca Benini and Giovanni De Micheli. Automatic synthesis of low-power gated-clock finite-state machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(6):630–643, 1996. `doi:10.1109/43.503933`.

7 Reinaldo A. Bergamaschi and Yunjian Jiang. State-based power analysis for systems-on-chip. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 638–641. ACM, 2003. `doi:10.1145/775832.775992`.

8 Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 sim-

ulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011. doi:10.1145/2024716.2024718.

9 Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. System-level modeling of energy in TLM for early validation of power and thermal management. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1609–1614. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.327.

10 Tayeb Bouhadiba, Matthieu Moy, Florence Maraninchi, Jérôme Cornet, Laurent Maillet-Contoz, and Ilija Materic. Co-simulation of functional systemc TLM models with power/thermal solvers. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 2176–2181. IEEE, 2013. doi:10.1109/IPDPSW.2013.206.

11 Dennis D Buss. Technology in the internet age. In *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, volume 1, pages 18–21. IEEE, 2002.

12 M. Caldari, M. Conti, P. Crippa, G. Nuzzo, S. Orcioni, and C. Turchetti. Instruction based power consumption estimation methodology. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 721 – 724 vol.2, 2002. doi:10.1109/ICECS.2002.1046270.

13 Marco Caldari, Massimo Conti, Massimo Coppola, Paolo Crippa, Simone Orcioni, Lorenzo Pieralisi, and Claudio Turchetti. System-level power analysis methodology applied to the AMBA AHB bus. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 20032–20039. IEEE Computer Society, 2003. doi:10.1109/DATE.2003.10234.

14 Chang-Chih Chen and Linda Milor. System-level modeling and microprocessor reliability analysis for backend wearout mechanisms. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1615–1620. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.328.

15 Jérôme Cornet. *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*. PhD thesis, Institut National Polytechnique de Grenoble, 2008.

16 Jérôme Cornet, Laurent Maillet-Contoz, Ilija Materic, Sylvian Kaiser, Hela Boussetta, Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. Co-Simulation of a SystemC TLM Virtual Platform with a Power Simulator at the Architectural Level: Case of a Set-Top Box. In *Design Automation Conference*, page SESSION 10U: USER TRACK, San Francisco, États-Unis, Jun 2012.

17 Alessandro Danese, Graziano Pravadelli, and Ivan Zandona. Automatic generation of power state machines through dynamic mining of temporal assertions. In Luca Fanucci and Jürgen Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 606–611. IEEE, 2016. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7459383.

18 Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):869–884, 2016. doi:10.1109/TPDS.2015.2412137.

19 Dipankar Das, P. P. Chakrabarti, and Rajeev Kumar. Thermal analysis of multiprocessor soc applications by simulation and verification. *ACM Trans. Design Autom. Electr. Syst.*, 15(2), 2010. doi:10.1145/1698759.1698765.

20 Nagu R. Dhanwada, Ing-Chao Lin, and Vijaykrishnan Narayanan. A power estimation methodology for systemc transaction level models. In Petru Eles, Axel Jantsch, and Reinaldo A. Bergamaschi, editors, *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2005, Jersey City, NJ, USA, September 19-21, 2005*, pages 142–147. ACM, 2005. doi:10.1145/1084834.1084874.

21 Mohammad Javad Dousti and Massoud Pedram. Power-efficient control of thermoelectric coolers considering distributed hot spots. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 966–971. ACM, 2015. URL: http://dl.acm.org/citation.cfm?id=2757037.

22 Bernhard Fischer, Christian Cech, and Hannes Muhr. Power modeling and analysis in early design phases. In Gerhard Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014. doi:10.7873/DATE.2014.210.

23 David Greaves and Mehboob Yasin. Tlm power3: Power estimation methodology for systemc tlm 2.0. In *Models, Methods, and Tools for Complex Chip Design*, pages 53–68. Springer, 2014.

24 Kim Grüttner, Philipp A. Hartmann, Tiemo Fandrey, Kai Hylla, Daniel Lorenz, Stefan Stattelmann, Björn Sander, Oliver Bringmann, Wolfgang Nebel, and Wolfgang Rosenstiel. An ESL timing & power estimation and simulation framework for heterogeneous socs. In *XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14-17, 2014*, pages 181–190. IEEE, 2014. doi:10.1109/SAMOS.2014.6893210.

25 Claude Helmstetter, Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. Fast and modular transaction-level-modeling and simulation of power and temperature. Technical report, Verimag Research Report, 2014. URL: http://www-verimag.imag.fr/~moy/?LIBTLMPWT-Model-Power-Consumption.

26 Claude Helmstetter, Vania Joloboff, and Hui Xiao. Simsoc: A full system simulation software for embedded systems. In *Open-source Software for Sci-*

entific Computation (OSSC), 2009 IEEE International Workshop on, pages 49–55, Sept 2009. doi:10.1109/OSSC.2009.5416870.

27 Claude Helmstetter and Matthieu Moy. LIBTLMPWT: Model power-consumption and temperature in systemc/tlm. Distributed under the terms of the GNU General Public License version 2, 2013. URL: http://www-verimag.imag.fr/~moy/?LIBTLMPWT-Model-Power-Consumption.

28 Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. Hotspot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Trans. VLSI Syst.*, 14(5):501–513, 2006. doi:10.1109/TVLSI.2006.876103.

29 IEEE. Ieee 1801-2009 — unified power format (upf), 2009. URL: http://standards.ieee.org/develop/project/1801.html.

30 Intel. *Intel? 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*, order number: 253669-057us edition, December 2015.

31 Sylvian Kaiser, Ilija Materic, and Rabih Saade. ESL solutions for low power design. In Louis Scheffer, Joel R. Phillips, and Alan J. Hu, editors, *2010 International Conference on Computer-Aided Design, ICCAD 2010, San Jose, CA, USA, November 7-11, 2010*, pages 340–343. IEEE, 2010. doi:10.1109/ICCAD.2010.5653615.

32 Pratyush Kumar and Lothar Thiele. System-level power and timing variability characterization to compute thermal guarantees. In Robert P. Dick and Jan Madsen, editors, *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, pages 179–188. ACM, 2011. doi:10.1145/2039370.2039400.

33 Sumeet S. Kumar, Amir Zjajo, and René van Leuken. Ctherm: An integrated framework for thermal-functional co-simulation of systems-on-chip. In Masoud Daneshtalab, Marco Aldinucci, Ville Leppänen, Johan Lilius, and Mats Brorsson, editors, *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 674–681. IEEE Computer Society, 2015. doi:10.1109/PDP.2015.56.

34 Hugo Lebreton and Pascal Vivet. Power modeling in systemc at transaction level, application to a DVFS architecture. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2008, 7-9 April 2008, Montpellier, France*, pages 463–466. IEEE Computer Society, 2008. doi:10.1109/ISVLSI.2008.71.

35 Ons Mbarek, Alain Pegatoquet, and Michel Auguin. A methodology for power-aware transaction-level models of systems-on-chip using UPF standard concepts. In José L. Ayala, Braulio García-Cámara, Manuel Prieto, Martino Ruggiero, and Gilles Sicard, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation - 21st International Workshop, PATMOS 2011, Madrid, Spain, September 26-29, 2011. Proceedings*, volume 6951 of *Lecture Notes in Computer Science*, pages 226–236. Springer, 2011. doi:10.1007/978-3-642-24154-3_23.

36 Ons Mbarek, Alain Pegatoquet, and Michel Auguin. A methodology for power-aware transaction-level models of systems-on-chip using UPF standard concepts. In José L. Ayala, Braulio García-Cámara, Manuel Prieto, Martino Ruggiero, and Gilles Sicard, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation - 21st International Workshop, PATMOS 2011, Madrid, Spain, September 26-29, 2011. Proceedings*, volume 6951 of *Lecture Notes in Computer Science*, pages 226–236. Springer, 2011. doi:10.1007/978-3-642-24154-3_23.

37 Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1137–1142. ACM, 2012. doi:10.1145/2228360.2228568.

38 Pierre Michaud and Yiannakis Sazeides. ATMI: analytical model of temperature in microprocessors. *Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

39 Farid N. Najm. Towards a high-level power estimation capability. In Massoud Pedram, Robert W. Brodersen, and Kurt Keutzer, editors, *Proceedings of the 1995 International Symposium on Low Power Design 1995, Dana Point, California, USA, April 23-26, 1995*, pages 87–92. ACM, 1995. doi:10.1145/224081.224097.

40 Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 29–42. ACM, 2012. doi:10.1145/2168836.2168841.

41 Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 200–209. IEEE Computer Society, 1998. doi:10.1109/REAL.1998.739746.

42 Björn Sander, Jürgen Schnerr, and Oliver Bringmann. ESL power analysis of embedded processors for temperature and reliability estimations. In Wolfgang Rosenstiel and Kazutoshi Wakabayashi, editors, *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009, Grenoble, France, October 11-16, 2009*, pages 239–248. ACM, 2009. doi:10.1145/1629435.1629469.

43 Tanguy Sassolas, Chiara Sandionigi, Alexandre Guerre, Alexandre Aminot, Pascal Vivet, Hela Boussetta, Luca Ferro, and Nicolas Peltier. Early

design stage thermal evaluation and mitigation: The locomotiv architectural case. In Gerhard Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–2. European Design and Automation Association, 2014. doi:10.7873/DATE.2014.327.

44 Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. High-performance timing simulation of embedded software. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 290–295. ACM, 2008. doi:10.1145/1391469.1391543.

45 Si2. Common power format specification (CPF) 2.1, 2014.

46 M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 2007.

47 An open platform for virtual prototyping of multiprocessors system-on-chip. URL: http://www.soclib.fr/.

48 Arvind Sridhar, Alessandro Vincenzi, Martino Ruggiero, Thomas Brunschwiler, and David Atienza. 3d-ice: Fast compact transient thermal modeling for 3d ics with inter-tier liquid cooling. In Louis Scheffer, Joel R. Phillips, and Alan J. Hu, editors, *2010 International Conference on Computer-Aided Design, ICCAD 2010, San Jose, CA, USA, November 7-11, 2010*, pages 463–470. IEEE, 2010. doi:10.1109/ICCAD.2010.5653749.

49 Synopsys. Primetime PX, 2015. URL: https://www.synopsys.com/apps/support/training/primetimepx_fcd.html.

50 Federico Terraneo, Davide Zoni, and William Fornaciari. An accurate simulation framework for thermal explorations and optimizations. In Gianluca Palermo, Daniel Gracia Pérez, Morteza Biglari-Abhari, Daniel Chillet, Smaïl Niar, and Adam Morawiec, editors, *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO@HiPEAC 2015, 21 January, 2015, Amsterdam, The Netherlands*, pages 5:1–5:6. ACM, 2015. doi:10.1145/2693433.2693438.

51 Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. VLSI Syst.*, 2(4):437–445, 1994. doi:10.1109/92.335012.

52 Ankush Varma, Eric Debes, Igor Kozintsev, Paul Klein, and Bruce L. Jacob. Accurate and fast system-level power modeling: An xscale-based case study. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347378.

53 Emmanuel Viaud, François Pêcheux, and Alain Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In Georges G. E. Gielen, editor, *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, pages 94–99. European Design and Automation Association, Leuven, Belgium, 2006. doi:10.1109/DATE.2006.244003.

54 Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In Michael M. Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pages 93–113. Springer, 2008. doi:10.1007/978-3-540-68564-7_7.

55 MY Yasin, C Koch-Hofer, Pascal Vivet, and DJ Greaves. Tlm power 3.0 (cbg) user manual version: Cbg 3.2 alpha draft manual-updated 1q2015-rev f, 2015.

56 Francesco Zanini, David Atienza, Luca Benini, and Giovanni De Micheli. Multicore thermal management with model predictive control. In *European Conference on Circuit Theory and Design (ECCTD 2009)*, volume 1, pages 90 – 95. IEEE Press, 2009. doi:10.1109/ECCTD.2009.5275073.

Optimal Scheduling of Periodic Gang Tasks

Joël Goossens[1] **and Pascal Richard**[2]

1 **Université Libre de Bruxelles (ULB)**
50 av. F.D. Roosevelt 1050 Brussels, Belgium
joel.goossens@ulb.ac.be

2 **LIAS/Isae-Ensma-Université de Poitiers**
1 av. Clément Ader, BP 40109, 86961 Chasseneuil du Poitou, France
pascal.richard@univ-poitiers.fr

Abstract

The gang scheduling of parallel implicit-deadline periodic task systems upon identical multiprocessor platforms is considered. In this scheduling problem, parallel tasks use several processors simultaneously. We propose two DP-Fair (deadline partitioning) algorithms that schedule all jobs in every interval of time delimited by two subsequent deadlines. These algorithms define a static schedule pattern that is stretched at run-time in every interval of the DP-Fair schedule. The first algorithm is based on linear programming and is the first one to be proved optimal for the considered gang scheduling problem. Furthermore, it runs in polynomial time for a fixed number m of processors and an efficient implementation is fully detailed. The second algorithm is an approximation algorithm based on a fixed-priority rule that is competitive under resource augmentation analysis in order to compute an optimal schedule pattern. Precisely, its speedup factor is bounded by $(2 - 1/m)$. Both algorithms are also evaluated through intensive numerical experiments.

2012 ACM Subject Classification Computer systems organization, Real-time systems, Embedded and cyber-physical systems, Software and its engineering, Process management, Scheduling, Multithreading, Theory of computation, Design and analysis of algorithms, Scheduling algorithms

Keywords and phrases Real-time systems, scheduling, parallel tasks

Digital Object Identifier 10.4230/LITES-v003-i001-a004

Received 2015-05-21 **Accepted** 2016-05-05 **Published** 2016-06-29

1 Introduction

We consider the preemptive scheduling of real-time tasks on identical multiprocessor platforms (see [13]). We deal with *parallel* real-time tasks, the case where each job may be executed on different processors *simultaneously*, i.e., we have *job parallelism*. Nowadays, the design of parallel programs is common thanks to parallel programming paradigms like Message Passing Interface (MPI [26, 28]) or Parallel Virtual Machine (PVM [40, 23]). Even better, sequential programs can be parallelized using standards like OpenMP application programming interface (see [8] for details).

Contributions. We define and prove correct a technique to schedule *optimally* periodic implicit deadline rigid gang tasks (see Definition 1 for details) upon multiprocessors. The algorithm is based on linear programming (LP) and runs in polynomial time for a fixed number m of processors. The second proposed method is a fixed-task priority rule with a performance guarantee of $(2-\frac{1}{m})$ under resource augmentation analysis. These algorithms are compared through numerical experiments.

Organization. Section 2 presents the studied scheduling problem and the related work. Section 3 presents basic results about DP-Fair scheduling of parallel tasks. Section 4 presents the LP-based optimal method and its implementation. Section 5 presents a gang heuristic and its worst-case

performance analysis under resource augmentation. Section 6 presents numerical results comparing both methods on randomly generated task systems. Then, Section 7 concludes the paper and presents some future work.

2 Model, Problem and Related Work

2.1 Parallel Task & Job Models

We deal with jobs and tasks which may be executed on different processors at the very same instant, in which case we say that *job (or task) parallelism* is allowed. Various kinds of parallel task models exist. Goossens et al. [25] adapted parallel computing terminology [7] to *recurrent* (real-time) tasks and jobs as follows.

▶ **Definition 1** ([25] Rigid, Moldable and Malleable Job)**.** A *job* is said to be (i) *rigid* if the number of processors assigned to this job is specified externally to the scheduler a priori, and does not change throughout its execution; (ii) *moldable* if the number of processors assigned to this job is determined by the scheduler, and does not change throughout its execution; (iii) *malleable* if the number of processors assigned to this job can be changed by the scheduler during the job's execution.

A recurrent task is said to be *rigid* if all its jobs are rigid, and the number of processors assigned to the jobs is specified externally to the scheduler; a recurrent task is said to be *moldable* if all its jobs are moldable; malleable if all its jobs are malleable.

Additionally at task level the literature distinguishes between at least three kinds of parallelism:

- *Multithread [12, 32, 39, 38, 1].* Each task is sequence of phases (multiphase in the following), each phase is composed of several threads, each thread requires a single processor for execution and threads *can* be scheduled simultaneously [38]. A particular case is the *Fork-Join* (see e.g. [36]) task model where the phases are an alternate sequence of sequential and parallel segments; task begins as a single master thread that executes sequentially until it encounters the first fork construct, where it splits into multiple parallel threads which synchronize/join on their terminaison and so on.
- *Dag task model [2, 6, 34].* The model generalizes the fork-join model, each task is represented as a directed acyclic graph, which is a set of precedence-constrained sequential jobs. Any group of jobs that are not constrained *may* execute in parallel.
- *Gang [11, 4, 25, 30].* Each task corresponds to $e \times k$ rectangle where e is the execution time requirement and k the number of required processors with the restriction: the k processors *must* execute task in *unison* (i.e., at the exact same time). This model is very representative to real world parallel applications where, at the submission time, users or scheduler select the number of processors for the task [14, 16] and consequently the number of generated threads corresponds to the number of used processors like MPI [41] and OpenMP [9] tools do. The threads communicate each other, they must be ready to communicate at the same time which imposes the synchronous threads execution.

2.2 Our Task/Job Model and Scheduling Problem

At job-level we consider the preemptive scheduling of parallel jobs on a multiprocessor platform upon m identical processors. We will focus on the problem of scheduling a set of rigid gang parallel jobs, each job $J_j \stackrel{\text{def}}{=} (r_j, v_j, e_j, d_j)$ is characterized by a release time r_j, a required number of processors v_j, an execution requirement e_j and an absolute deadline d_j. The job J_j must execute

for e_j time units over the interval $[r_j, d_j)$ on v_j processors. We consider the scheduling of *rigid* jobs since v_j is fixed externally to the scheduler.

Our main scheduling problem concerns *periodic* (and sporadic —see discussion Section 5.3) preemptive *hard* real-time systems. Let $\tau \stackrel{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$ denote a set of n periodic *implicit-deadline* rigid gang parallel tasks. Each task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $((k-1) \cdot T_i, v_i, C_i, k \cdot T_i)$. In the following $u_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$ denotes the utilization factor of task τ_i. This utilization is not related to the number of required processors (v_i) this is an *horizontal* notion. The execution requirement of a job of τ_i corresponds to a $C_i \times v_i$ *rectangle*.

This Research. We study the scheduling of preemptive periodic implicit-deadline rigid *gang* real-time tasks. We address the feasibility and the schedulability questions by designing an optimal scheduler. The proposed approach exploits the deadline partitioning of the schedule. Precisely, in every slice delimited by two subsequent deadlines in the schedule, a static schedule pattern is stretched according to the slice length. Two algorithms are proposed to define such a pattern. The first algorithm is based on linear programming and is the first one to be proved optimal for the considered gang scheduling problem. Furthermore, it runs in polynomial time for a fixed number m of processors and an efficient implementation is fully detailed. The second algorithm is an approximation algorithm based on a fixed-priority rule that is competitive under resource augmentation analysis for computing a static schedule pattern. Precisely, its speedup factor is bounded by $(2 - 1/m)$. Both algorithms are also evaluated through intensive numerical experiments.

For the sake of simplicity we consider periodic tasks and we assume that C_i is the *exact* duration of the task τ_i. Consequently, we consider an offline scheduling problem. The discussion of Section 5.3 extends the scope of our result to the scheduling of *sporadic* tasks with C_i as the *worst-case* execution time.

2.3 Related Work

Optimal solutions. To the best of our knowledge there is a single work which provides an *optimal* solution for the scheduling of recurrent *hard* real-time *parallel* computing: the work of Collette et al. [11] which considers sporadic implicit-deadline *malleable* gang scheduling. The authors provide an optimal scheduler and an exact feasibility/schedulability test. The work we report in this document provides an optimal scheduler and an exact feasibility/schedulability test as well, except we consider a more realistic parallel task model since our task are *rigid* jobs—the degree of parallelism is specified at design time and does not change at run-time—while the authors of [11] consider *malleable* jobs.

Non optimal solutions. The other contributions to the scheduling of recurrent *hard* real-time *parallel* computing consider *non* optimal schedulers and schedulability tests (sufficient or exact). [30] considers the EDF (Earliest Deadline First [35]) scheduler for rigid gang sporadic tasks and proposes a sufficient schedulability test. [25, 4] consider FTP (Fixed Task Priority, such as Rate Monotonic [35]) periodic gang scheduling and provide an exact/sufficient schedulability tests. [12] considers FTP and periodic multithread tasks and proposes an exact schedulability test. [32] consider Deadline Monotonic scheduling of fork-join tasks, they provide a competitive analysis for that suboptimal scheduler. [39] considers EDF and FTP scheduling for multiphase multithread periodic tasks and provides a task decomposition technique and a competitive analysis. [38] considers optimal schedulers for sequential tasks (e.g., DP-Fair described Section 3.4), implicit

deadline multiphase multithread recurrent tasks, for which the authors propose a decomposition technique and a competitive analysis. More recent works [6, 34] consider DAG tasks model (which generalizes fork-join model). The authors study the competitiveness of global EDF and global RM.

3 Basic Results

3.1 Hardness for arbitrary number of processors

Assuming that the number of processors is an input in the problem model, it is easy to show that the preemptive Gang scheduling problem is equivalent to the Bin Packing problem. Such a result was observed (without proof) for the non preemptive scheduling problem of parallel (Gang) tasks having unit processing times [5]. Hereafter, we provide a basic proof sketch to clearly exhibit the reducibility among both problems.

▶ **Theorem 2.** *Preemptive Gang scheduling is* NP*-hard in the strong sense for problem instances with an arbitrary number of processors.*

Proof. (Sketch) We transform from Bin Packing [22]: Finite set A of items, a rational size $s(a) \in \mathbb{Q}^+$ for each $a \in A$, a positive integer bin capacity B, and a positive integer K; is there a partition A into disjoint $A_1, A_2, \ldots, A_K$ such that the sum of the sizes of the items in each A_i is no more than B?

Without loss of generality we assume that all $s(a)$ have a common denominator and B is scaled accordingly, we define a Gang scheduling instance as follow:

- $m = B$ processors,
- each item $a \in A$ is model as a unit-length task of period K, $\tau_a = (v_a, e_a, K)$, such that $e_a = 1$ and $v_a = s(a)$.

The Bin Packing decision problem is equivalent to determine if there is a feasible schedule of period K? (i.e., a schedule with no more than m processors are simultaneously used at any time). ◀

The previous transformation shows that preemptive Gang Scheduling with an arbitrary number of processors contains the bin packing problem as a particular case (i.e., with all task execution times equal to 1). It is also known that the Bin Packing problems can be solved in polynomial time for any fixed B by exhaustive search [22]. Such a property will be also exploited for defining our optimal polynomial time algorithm for scheduling periodic Gang real-time tasks.

3.2 Maximum Rectangle Utilization Bound

In our task model (see Section 2.2), the task utilization $u_i \stackrel{\text{def}}{=} C_i/T_i$ represents an *horizontal* utilization. In this section we introduce the notion of total *rectangle* task set utilization which is by definition $\sum_{i=1}^{n} \frac{C_i \times v_i}{T_i}$.

The maximum rectangle utilization bound U_b of a scheduler guarantees that every system of tasks whose total utilization is smaller than or equal to U_b will be correctly scheduled. Beyond this utilization limit, and if the bound is said to be tight, then there exist systems of tasks which are not schedulable.

First notice that, since our scheduling problem of parallel tasks is a generalization of the popular scheduling problem of periodic *sequential* tasks, we have to report a negative result: the maximum rectangle utilization bound is $1/m$.

▶ **Theorem 3.** *The maximum rectangle utilization bound for the scheduling of periodic Gang tasks is* $1/m$.

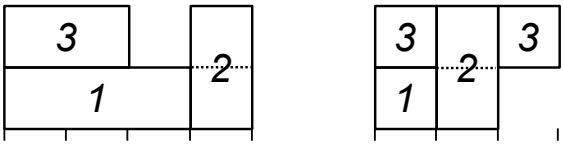


Figure 1 Non-predictability of gang FJP schedulers. Job 1 has the highest priority, job 3 has the lowest one and job 2 in the middle, and they all arrive at time 0.

Proof. The result will be established using a simple parallel task set with two tasks[1]: $\tau_1(1,1,1)$ and $\tau_2(m,\epsilon,1)$, where ϵ is an arbitrary small positive infinitesimal number. This system is trivially infeasible and the total task rectangle utilization is $1/m$. ◀

3.3 Scheduling Anomalies

We have to report a second negative result concerning our scheduling problem: FJP (Fixed Job Priority, such as Earliest Deadline First [35]) and consequently FTP gang scheduling are not predictable [25] [2]. Here is an example task system, on 2 processors and three jobs[3] (see Figure 1):

$$J_1 = (0,1,3,3), J_2 = (0,2,1,4), J_3 = (0,1,2,2) \ .$$

Using the priority assignment $J_1 > J_2 > J_3$, Gang FJP schedules the set of jobs (J_3 completes at time-instant 2). Unfortunately, if the actual duration of J_1 is 1, J_2 will preempt J_3 at time $t = 1$ and J_3 will complete *later*, at time-instant 3. Then, J_3 does not miss its deadline in the "worst case" scenario, but misses it if J_1 uses less than its worst case execution time C_1. Thus, reducing an execution time can delay the completion of another job.

3.4 DP-Fair Scheduling for Sequential Tasks

While this research concerns *parallel* tasks we will introduce a scheduling technique defined for *sequential* tasks (in the next section —Section 3.5— we will show how to apply the very same technique to gang tasks). Consequently, we assume in this section the scheduling of n *sequential* and implicit-deadline tasks upon m identical processors. Each task $\tau_i \stackrel{\text{def}}{=} (C_i, T_i)$ is characterized by a worst-case execution duration C_i and a period T_i. Seminal optimal multiprocessor scheduling techniques were based on the notion of *proportionate fairness*, it is the case for instance of the PF (Proportionate Fairness) scheduler [3]. This type of algorithm assumes that time is discrete.

The quantum-by-quantum construction of the scheduling *is not necessary* in order to define an optimal algorithm [42, 20, 24]. DP (Deadline Partitioning) scheduling techniques do not decompose the tasks into single-time unit (sub)-tasks. The construction of the scheduling is done over time intervals delimited by two *consecutive deadlines* called blocks. In each block, every task receive a workload that is proportional to its utilization so that the fairness property is satisfied at each deadline in the schedule.

Let L_j be the length of the block j delimited by two subsequent task deadlines, every implicit deadline periodic task τ_i receives an amount of processor equal to $L_j \times u_i$. Consequently, a task τ_i has received an execution times equal to $u_i \times T_i = C_i$ for each of its deadlines.

---

[1] (v_i, C_i, T_i)

[2] a scheduling algorithm is predictable if reducing an execution requirement cannot increase the completion of tasks.

[3] (r_j, v_j, e_j, d_j)

The previous algorithms assume that time is by its nature discrete: the times at which the scheduler can be activated are integers (in other words correspond to the clock ticks of the real-time operating system). Discrete time is by its nature a source of complexity in multiprocessor scheduling and for this reason, algorithms which exploit the continuous nature of time have been defined.

We will now detail the simplest algorithm in this category: the DP-Wrap algorithm.

The DP-Wrap algorithm is a very simple deadline fair algorithm which is optimal for tasks with implicit deadlines [20]. Contrarily to the previous algorithms, DP-Wrap considers that the time is continuous. The scheduling is broken down into blocks delimited by deadlines/periods. The distribution of the tasks into each interval is equal to the length of the interval multiplied by the utilization of the task, i.e., $u_i \stackrel{\text{def}}{=} C_i/T_i$. Thus, in the interval $[s_j, s_{j+1})$, each task τ_i is given $C_i(j) \stackrel{\text{def}}{=} u_i \times (s_{j+1} - s_j)$. Consequently, at each deadline, the tasks have received an execution time equals to $u_i \times T_i = C_i$. The scheduling (distribution) in each block is done by McNaughton's algorithm, which has been proposed in 1959 [37].

In the next section, we show how to reuse the deadline partitioning technique in order to define an optimal static gang scheduling algorithm.

3.5 DP-Fair and Gang Scheduling

The next result (Theorem 4) shows that concerning our parallel scheduling problem (gang scheduling defined Section 2.2) we can, without loss of generality, consider DP-Fair scheduling, i.e., schedule where the same pattern is replicated (and stretched) in each interval delimited by deadline/period. In Section 4 we will define a technique to build such pattern optimally.

▶ **Theorem 4.** *Every feasible parallel task set is schedulable with a DP-Fair schedule.*

Proof. Assume we have a feasible schedule, then we show how to define a feasible DP-Fair schedule. Since parallel tasks are strictly periodic and are simultaneously released, the whole schedule is periodic and has a period equal to the hyperperiod. Within the hyperperiod H, every task is executed for $\frac{H}{T_i}C_i = Hu_i$. To define a DP-Fair schedule:

- Schedule pattern: stretch the complete schedule within unit time slots.
- Stretch the pattern accordingly in every block of the schedule.

The corresponding schedule is DP-Fair. At every block boundary t, every task $\tau_i, 1 \leq i \leq n$ receives exactly $t \cdot u_i$. Hence, for every time instant t corresponding to a deadline of task τ_i (i.e., $t = kT_i$) the task τ_i receives exactly $kT_iu_i = kC_i$ and thus has been executed to completion by its deadline. ◀

4 Optimal Pattern Definition

4.1 Research Method

Firstly, we will revisit a non real-time scheduling problem and its solution (the work of Błazewicz et al. [5]) where the main goal is to *minimize* the schedule length of non recurrent rigid gang jobs. We will show how that technique can be *optimally* adapted to our hard real-time scheduling problem.

4.2 The work of Błazewicz et al. revisited

Principles. In [5] the authors consider the scheduling of n rigid gang jobs, each job J_i is characterized by the couple (u_i, v_i), i.e., a duration u_i and a required number of processors v_j, all

these jobs are released simultaneously at time origin. Upon an identical multiprocessor platform the scheduling problem is to find a schedule which minimizes the schedule length, the first instant where the jobs are completed or equivalently to minimize the makespan.

The authors present a polynomial time algorithm for a fixed number of processors m, based on linear programming, for computing an optimal schedule in the general case. Particular cases are also considered but not useful in our framework. Notice that the problem is NP-hard for arbitrary number of processors (i.e., m is an input of the problem) [15].

The method decomposes the schedule as a sequence of slices. Remember, a feasible allocation of jobs is the one that uses no more than m processors. Each slice σ_i is characterized by the set S_i of feasible jobs and the duration x_i of their execution. The algorithm computes the length for every feasible allocation of jobs. As a result, the slices having a positive length are sequenced in arbitrary order. Moreover the method minimizes the $\sum_{i=1}^{M} x_i$, i.e., the makespan to define an optimal schedule.

▶ **Definition 5** (Feasible allocation)**.** A *feasible allocation* of jobs is a subset s of job indexes that can be processed *simultaneously* on the platform: $\sum_{i \in s} v_i \leq m$. By definition we have a *finite* number of different feasible allocation sets. In the following M is the number of different feasible allocation sets. Thus the set of all feasible allocation subsets is denoted $S = \{S_1, \ldots, S_M\}$.

▶ **Example 6.** For instance, consider three jobs J_1, J_2, J_3 with $v_1 = 1, v_2 = 2, v_3 = 1, u_1 = 3, u_2 = 1$ and $u_3 = 2$. For $m = 2$ the feasible allocations are $S_1 = \{1\}, S_2 = \{2\}, S_3 = \{3\}$ (jobs can be executed alone); $S_4 = \{1, 3\}$ (J_1 and J_3 can be executed in parallel). Remark that J_2 cannot be executed with J_1 nor J_2.

Notice that S has a cardinality of $M \stackrel{\text{def}}{=} |S| \leq \sum_{k=1}^{m} \binom{n}{k} \leq (n)^m$. It is important to notice that the number of subsets M (i.e., number of variables in the linear program) is in $O(n^m)$ that is *polynomial* for fixed values of m.

Let Q_j be the set of those subset indexes which contain job J_j. Let x_i be the processing time of the subset S_i in the schedule and used to define variables in the linear program. The linear program computes the schedule as a set of slices of length x_i. Every value x_i such that $x_i > 0$ defines a slice in the schedule in which the jobs of S_i are executed. Slices are executed in an arbitrary order without any inserted delays between them. Hence, the computed makespan is $\sum_{i=1}^{M} x_i$.

The objective function is to minimize the makespan: $\sum_{i=1}^{M} x_i$; and the linear program must enforce that all jobs are completed. The corresponding constraint is: $\sum_{i \in Q_j} x_i = u_j, j = 1 \ldots n$. Hence, the schedule with the smallest length (i.e., makespan) is defined as follows:

Algorithm 1: Optimal Schedule Pattern construction by Linear Programming.

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{M} x_i \\ \text{subject to} & \sum_{i \in Q_j} x_i = u_j \qquad j = 1 \ldots n \end{array}$$

A solution for Example 6 is $x_1 = 1$ (duration of the execution of J_1 only), $x_2 = 1$ (duration of the execution of J_2 only), $x_3 = 0$ and $x_4 = 2$ (duration of the joint execution of J_1 and J_3) which corresponds to the schedule of Figure 2.

In the previous linear program, there are: M variables and n constraints. It can be solved in polynomial time using for instance Khachiyan's algorithm [31]. This is pretty much what Błazewicz et al. [5] did to solve optimally and polynomially their scheduling problem. We first show how

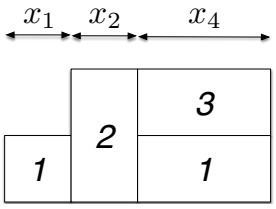


Figure 2 A solution for Example 6.

that technique can be used to solve our hard real-time scheduling problem (Section 4.3). Then, we will show how to speedup the resolution time by defining an efficient problem construction before calling the LP solver (Section 4.4).

4.3 Minimizing the Makespan vs. Meeting Hard Real-time Deadline of Recurrent Tasks

The next property establishes an equivalence between the Błazewicz et al. optimal solution and an optimal scheduler for our real-time scheduling problem thanks to the DP-Fair scheduling theory [33].

▶ **Theorem 7.** *A periodic implicit deadline rigid gang scheduling system* $(v_i, C_i, T_i)_{i=1\cdots n}$ *is feasible* $\Leftrightarrow$ *the set of synchronous jobs* $(u_i, v_i)_{i=1\cdots n}$ *has a minimum makespan not larger than the unity.*

Proof. $\Leftarrow$ Assuming the makespan of the set of synchronous jobs $(u_i, v_i)_{i=1\cdots n}$ is not larger than the unity we have a schedule *pattern* which executes each task for a duration of u_i in a unit-length interval. Using deadline partitioning fairness theory (DP-Fairness, see [33, 20, 10, 18, 21, 19]) we can schedule our original *periodic* task set. The main idea of that kind of schedule is the deadline partitioning of the timeline: the time is divided in *slices* bounded by two successive job deadlines [33, 20]. All tasks are assigned a local execution time which is the length of the current slice times the task utilization u_i. As basically DP-Wrap [20] we use an identical pattern (the solution of the LP) in each time slice, i.e., the unitary pattern is *stretched* according to the length of the slice. Thanks to the DP-Fair theory, we know that for each time interval $[kT_i, (k+1)T_i)$ the task τ_i executes during $u_i T_i = C_i$ time units on v_i processors simultaneously. Consequently, the periodic gang task is feasibly scheduled.

$\Rightarrow$ We will show the contra-positive, i.e., assuming that *all* schedules of the synchronous job set have a length larger than one. Since DP-Fair is optimal for periodic implicit-deadline systems and since the technique of Błazewicz et al. determines the minimal makespan we can conclude that it is *necessary* for the schedulability of periodic implicit-deadline tasks that in each slice the active task τ_i executes for u_i times the slice length. Hence, slices where all tasks are active (like the first one in the synchronous case) cannot execute u_i times the slice length since the solution of the LP is larger than one, consequently (because DP-Fair is optimal), the periodic system is not schedulable on m unit-speed processors. ◀

4.4 LP implementation issues

Efficient generation of the set of all feasible allocations S (Definition 5) is the main combinatorial problem in the linear program construction (Algorithm 1) in order to setup the linear program that will compute the optimal schedule pattern. Even if the size of the problem is polynomially bounded for fixed values of m, the brute force definition of the set of all feasible allocations S

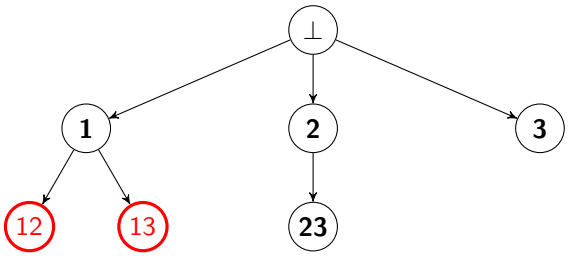


Figure 3 Search tree with nodes defined by task indexes in subsets; black nodes are feasible subsets whereas red nodes are infeasible subsets.

requires a huge amount of time for $n > 20$. This stage is the bottleneck of the approach since the linear program is solved quickly as it will be shown in the experimental section.

A simple way to implement a Brute Force generation of all subsets of tasks is to represent every feasible allocation by an integer in which the binary encoding represents the tasks selected in a subset. There are 2^n subsets of a set with n elements, exactly as there are 2^n different ways to write numbers with n bits. Let s denote such an integer, if i^{th} binary digit is 1 in s, then it indicates that task τ_i is in the feasible allocation represented by s. With such a binary encoding, the brute force enumeration of all feasible allocations of n tasks simply consists in counting from 1 to $2^n - 1$ and defining subsets from the binary encoding. Subsets corresponding to feasible allocations are those that do not use more than m processors.

Using the same binary encoding principles for feasible allocations, we define an efficient generation using Depth First Search with lexicographic ordering of enumerated subsets. Tasks are sorted in non increasing order of v_i. As a consequence, vertices corresponding to infeasible allocations are efficiently pruned. A branch in the search tree is pruned if the current vertex in the tree corresponds to subset of tasks that use more than m processors. The search tree is illustrated in Figure 3 for three tasks: $\{\tau_1, \tau_2, \tau_3\}$ which respectively use 3, 2 and 1 processors. The considered platform has 3 processors. Nodes define indexes of tasks in a subset. Black nodes are feasible subsets whereas red ones are infeasible (i.e., requires more than 3 processors). During the search, the node $\{123\}$ is not defined since $\{12\}$ is already infeasible (i.e., the branch is pruned or fathomed). Using a Depth First Search, the search tree simply consists in the list of unexplored subsets (i.e., encoded as one integer each) and its size is upper-bounded by $\mathcal{O}(n^2)$ subset entries. Our Matlab implementation of this algorithm, denoted DFSLex hereafter, is limited to 64 tasks (i.e., due to Matlab 64-bit integers). Brute Force and DFSLex methods will be compared in the section dedicated to numerical experiments. The performance of the LP for optimally solving Gang scheduling problems will be also presented in Section 6.

In the next section, we propose an heuristic that avoids the previous combinatorial problem. This heuristic has a performance guarantee in terms of resource augmentation (i.e., speedup factor).

5 Gang heuristic

This section presents a scheduling heuristic for defining the schedule pattern. As for the optimal solution presented in the previous section, we consider a DP-Fair schedule in which the pattern will be stretched in every block delimited by two subsequent deadlines of tasks.

The heuristic algorithm is a fixed-task priority scheduling rule that runs in $\mathcal{O}(n \log n)$ for an arbitrary number of processors. We provide a resource augmentation analysis to compare its worst-case performance against the optimal method.

5.1 Fixed-Task Priority Scheduling gang-h

[29] presents an heuristic for minimizing the makespan of preemptive parallel jobs. We shall reuse the basis of the algorithm for defining a fixed-task priority algorithm, denoted gang-h hereafter. As in the previous section, the algorithm is used to define the pattern of tasks to schedule in every time slot (i.e., block) of the DP-Fair schedule. As in the optimal method, the pattern is defined as a unit-length schedule in which utilization factors of tasks play the role of execution requirements. For each block in the schedule, the rule is simultaneously used to:

- allocate the portion of each task to processors.
- sequence tasks within the block.

gang-h is a fixed-task priority scheduling rule that works as follows [29]:

1. Priorities are assigned in non-increasing order of the number of requested processors (i.e., non-increasing order of v_i); ties are broken arbitrarily.
2. Scheduling decisions are taken every time a job is released or completed. At such event, all tasks are preempted and the priority list is used to allocate ready tasks to the processors greedily as feasibly while the current job can be scheduled on the remaining available processors.

The complete algorithm is described in Algorithm 2. This algorithm considers synchronous jobs and will be used to define the pattern of jobs to be scheduled in every block of the schedule (as in the optimal algorithm).

5.2 Optimality under resource augmentation

The scheduling rule gang-h is obviously not optimal for minimizing the pattern makespan in our DP-Fair approach. Nevertheless, we next prove that it is as powerful as an optimal algorithm if it is allowed to use a faster processor than the optimal algorithm executed upon a unit-speed processor. Such a performance guarantee quantifies the price being paid for using gang-h. Precisely, we establish that the speedup factor is bounded by $2 - 1/m$. We first recall the speedup factor metric.

▶ **Definition 8.** (Speedup factor) A scheduling algorithm A has a speedup factor f, $f \geq 1$, if it can schedule any task set that can be scheduled on a given platform by an optimal algorithm, provided that A is able to schedule the same task set upon a platform in which each processor is f times as fast as the processors available to the optimal algorithm.

For proving the resource augmentation performance guarantee for our real-time scheduling problem, we reuse the following results that establish the competitive ratio for the makespan minimization problem.

▶ **Lemma 9.** *(Theorem 3.1 in [29]) Let w_L be the makespan computed by gang-h and w_0 be the optimal makespan, then:*

$$w_L \leq (2 - \frac{1}{m})w_0 \tag{1}$$

The approximation bound of $(2 - 1/m)$ can be easily proved to be tight by using the same task set that has been proposed in [27]: $m^2 - m + 1$ jobs. This job set is defined by $(m^2 - m)$ unit-length jobs and one job of length m. Every task uses exactly one processor. Since ties are broken arbitrarily, assume that gang-h assigns the lowest priority to the job of length m. Consequently, gang-h defines a schedule that first allocates all unit-length jobs to the m processors and lastly the last job. This schedule is of length $m - 1 + m = 2m - 1$, whereas the optimal makespan is m by allocating the long job to a dedicated processor and by scheduling the unit-length jobs on $m - 1$ remaining processors.

Algorithm 2: gang-h (synchronous jobs).

input :
n jobs $J_j(u_j, v_j), 1 \leq j \leq n$;
m: number of processors;
output:
Slice lengths $S(s), s = 1, 2, \ldots$;
Scheduled jobs j in slice s: Sched(s,j) $\in \{0, 1\}, 1 \leq j \leq n$;

```
List=Sort(J_1, ..., J_n) ;        /* Jobs are sorted in non increasing order of v_j */
s = 0 ;                                                    /* Number of slices */
r_j := u_j   ∀j = 1 ··· n ;                  /* Job remaining execution times r_j */
while ∃j, r_j > 0, 1 ≤ j ≤ n do
    /* create a new slice                                                      */
    s = s + 1 ;                                            /* Number of slices */
    K = m ;                                          /* Remaining processors */
    ℓ = ∞ ;                                        /* Slice length upper bound */
    Sched(s,j)=0   1 ≤ j ≤ n ;                                  /* Empty Slice */
    foreach j ∈ List do
        /* For each job j in priority List                                     */
        if v_j ≤ K then
            /* the job j is schedulable in current slice                       */
            Sched(s,j)=1;
            ℓ = min(ℓ, r_j);                              /* update slice length */
            K = K − v_j ;                                /* remaining processors */
        end
    end
    S(s)=l;                                                /* Slice length */
    for j = 1 ... n do
        /* Update remaining execution times                                    */
        if Sched(s, j) then
            r_j = r_j − ℓ;
        end
    end
end
```

The following lemma states that gang-h always produces the same distribution of tasks among identical processors whatever the platform speed.

▶ **Lemma 10.** *Let P be the schedule pattern computed by gang-h on m unit-speed processors and w be its makespan, then gang-h produces a pattern P' of length w/s if a platform is of speed $s > 0$.*

Proof. All jobs in a pattern are simultaneously available at the beginning of the schedule. Thus, scheduling decisions are only taken by gang-h when a job is completed. Using m speed-s processors will stretch execution times to the value C_i/s for every task τ_i. Hence, the pattern is of length w/s. ◀

▶ **Theorem 11.** *gang-h has a speedup factor not exceeding* $(2 - \frac{1}{m})$.

Proof. Consider a task set τ on a platform Π defined by m unit-speed processors. Assume that τ is feasible upon Π and let w_0 be the length of the pattern computed by an optimal algorithm, then

by Lemma 9, gang-h produces a pattern of length not exceeding $(2 - \frac{1}{m})w_0$ on Π. Now consider a platform Π' where each processor is of speed $s = 2 - \frac{1}{m}$. The task set corresponding to τ, denoted τ', has an execution requirement defined by C_i/s for every tasks. By Lemma 10, the length of the schedule defined by gang-h upon Π' is not exceeding $(2 - \frac{1}{m})w_0/s = w_0$. Hence, the pattern defined by gang-h is feasible. ◀

Thus, a processor speedup of $2 - 1/m$ is an upper bound on the price being paid for using the presented gang heuristic for defining the schedule pattern to be stretched in every block of the DP-Fair schedule.

5.3 Extending the scope of the results

For the sake of simplicity we considered implicit-deadline periodic tasks and we assumed that C_i is the *exact* duration of the task τ_i. In this section we discuss straightforward extensions (constrained-deadlines, asynchronous systems and C_i as the worst-case execution requirement) and possible extensions which are left as future work (sporadic and arbitrary deadlines).

Constrained-deadlines. We considered in this work *implicit*-deadline rigid gang parallel tasks. Constrained-deadline tasks are characterized by an additional parameters $D_i \leq T_i$ the relative deadline. Each constrained-deadline task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i, D_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $((k-1) \cdot T_i, v_i, C_i, (k-1) \cdot T_i + D_i)$. DP-Fair techniques can be obviously extended for constrained-deadlines by considering the task *density* $\delta_i \stackrel{\text{def}}{=} C_i/D_i$ instead of task utilization. DP-Fair is not longer optimal for constrained-deadline and sequential tasks, but if the makespan of gang jobs $\{(\delta_i, v_i) \mid i = 1, \ldots, n\}$ is not larger than the unity our method schedule feasibly constrained-deadline gang tasks. Funk et al. extended for instance DP-Wrap for constrained- deadlines [20].

Asynchronous periodic tasks. Asynchronous periodic tasks are characterized by an additional parameters O_i the release time of the first job of τ_i. Each task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i, O_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $(O_i + (k-1) \cdot T_i, v_i, C_i, O_i + k \cdot T_i)$. Once again the technique can be used for that asynchronous system: we define the pattern for the synchronous job scenario and we apply the deadline partitioning method and stretching accordingly that pattern.

Early completion. We assumed in this work that C_i is the *exact* duration of the task τ_i. Meanwhile, from applicative perspective this is incorrect, at design time we determine the worst-case execution time (C_i is the WCET) for each task. At run-time the actual duration of any job of τ_i can be smaller than C_i. Again DP-Fair techniques can be obviously extended for that case, a task might not use all the capacity reserved for it, but because of scheduling anomalies reported Section 3.3 we have to respect the stretched pattern, in other words it is forbidden to schedule another task earlier.

Sporadic tasks. Sporadic tasks are quite similar to periodic tasks, the only difference being that the period of a sporadic task denotes the *minimum* inter-arrival time instead of the *exact* one. While Funk et al. show that handle arrivals within a time slice is fairly straightforward (see [20], Section 6.) for sequential tasks we consider that extension to parallel gang tasks is no direct and that extension is left as future work.

6 Numerical experiments

Intensive numerical experiments have been performed using Matlab. The used LP solver is an interior point method (i.e., solver linprog included in Matlab). Source codes of all algorithms and experimental results are available at the project page[4] including a wiki page for detailing the file organization. We next detail the task set synthesis and the numerical results.

6.1 Task set synthesis

Input parameters for the task set synthesis are m, i.e., the number of processors in the platform, its total utilization U, and the number of tasks n. Stafford's algorithm is used for generating utilization factors u_i of gang tasks $\tau_1, \dots \tau_n$ to meet a total utilization of $m \times U$. As shown in [17], the method is suitable for task set synthesis for multiprocessor systems. The utilization factors of tasks are picked up by Stafford's algorithm in the interval $[0.02, m]$. The number of used processors for every gang task is generated using uniformly distributed pseudo random integers in the interval $[1, m)$. We do not allow a task to simultaneously use m processors since such a situation is not interesting from scheduling perspectives. Precisely, such tasks can be removed from the optimization problem in order to compute an optimal pattern, and added afterwards in the previously computed optimal pattern.

In DP-Fair scheduling, task individual periods and execution requirements are not useful since between two subsequent deadlines, d_j and d_{j+1}, the execution requirement to be scheduled is exactly $(d_{j+1} - d_j) \times u_i$ for every task τ_i, $1 \leq i \leq n$. Furthermore, the presented algorithms build up the pattern that will be stretched in every block in the DP-Fair schedule. The length of the interval is basically set to one hundred to avoid small decimal numbers that can lead to numerical problems while using a LP solver.

6.2 LP-based method evaluation

As previously mentioned, the optimal LP-based algorithm must handle two different combinatorial problems: the feasible subsets construction stage and the optimization stage. From the computational time point of view, the optimization stage is quite fast in comparison with the construction of all feasible subsets (i.e., setting up the matrix of constraints).

Figure 4 presents the computation times of Brute Force v.s. Depth First Search with Lexicographic order (DFSLex) for this problem construction for $m = 16$. In the following plots, every point corresponds to 1000 runs (i.e., simulation with replication factor equal to 1000). As commonly observed, Brute Force is still manageable until $n = 20$, but cannot be used beyond whereas the DFSLex algorithm runs quite efficiently. The drawback of the DFSLex algorithm is that it is limited to 64 gang tasks due to the binary encoding of feasible subsets as 64-bit integers. The DFSLex results for larger task sets are depicted in Figure 5. We also implement a similar version of that algorithm relaxing the 64-bit constraint by using variable integer arithmetic routines but it runs quite slowly in our Matlab implementation (e.g., it is as slow as the Brute Force algorithm for small task sets). All these implementations are available in the project page.

Figure 6 depicts the resolution times of the optimal algorithm (with both stages) for several numbers of processors and for global utilization equal to 50% and 90%. As depicted, the utilization factor has a moderate influence on average resolution times. Problems become harder to solve when first, the number of gang tasks increases, and second, when the number of processors increases;

[4] http://www.lias-lab.fr/forge/projects/multiprocessorgangscheduling

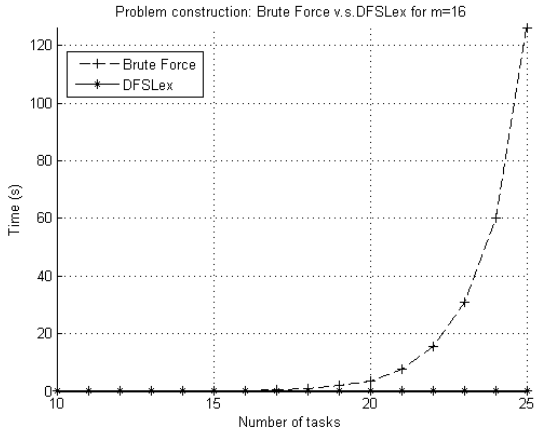


Figure 4 Brute Force v.s Depth First Search enumeration of feasible allocations in the linear program OPT.

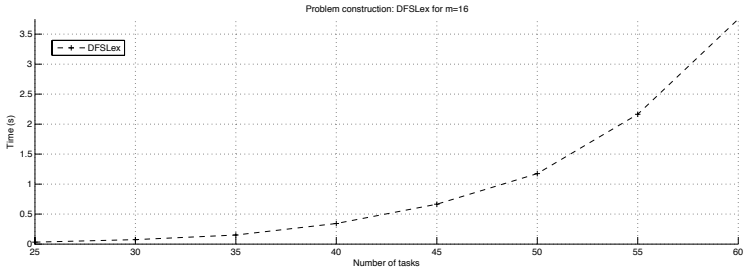


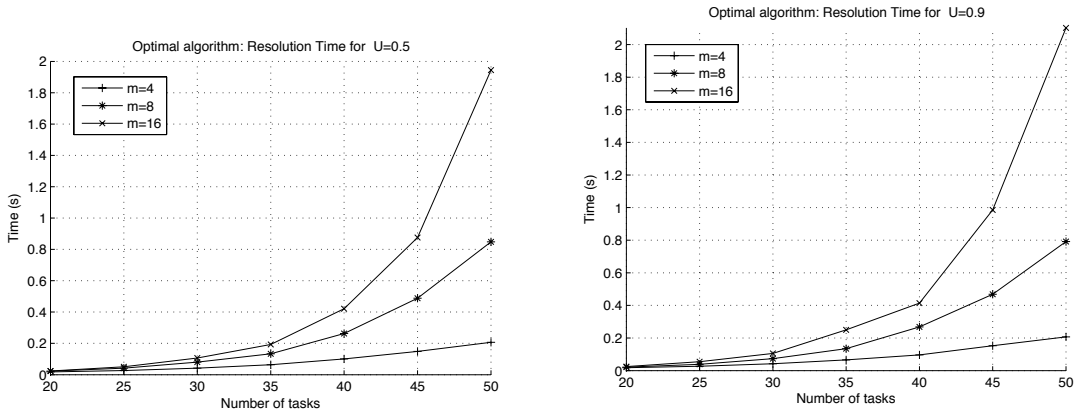
Figure 5 Depth First Search enumeration of feasible allocations for larger task sets.

but require few seconds in the average. In both cases, the feasible subsets construction requires an important amount of computations when the problem size increases.

6.3 Acceptance ratio

We compare the optimal algorithm (OPT) and the heuristic (gang-h) for computing a schedule pattern according to the average acceptance ratio for a platform with 16 processors and varying utilization factors. The used schedulability tests are Theorem 7 for the optimal algorithm and its sufficient version for the heuristic (i.e., if (gang-h) generates a schedule pattern of length not larger that 1, then the task set is schedulable). Figure 7 depicts the results for 20 and 40 tasks, respectively. The replication factor during the simulation is set to 10000 (i.e., every point in graphs is the average of 10000 results).

For the optimal algorithm, when the number of tasks increases in the experiment for $m = 16$ processors, then every task has relatively smaller individual utilization due to the task set synthesis method. As a consequence, there are more feasible subsets and consequently more feasible schedules. As depicted in Figure 7, the acceptance ratio for the optimal algorithm doubles for $U = 0.95$ when the number of tasks doubles. Such a benefit is not observed for the gang heuristic that achieves quite poor results when the total utilization becomes high.



(a) Resolution time for $U = 0.5$

(b) Resolution time for $U = 0.9$

Figure 6 Resolution times of the LP-based optimal algorithm.

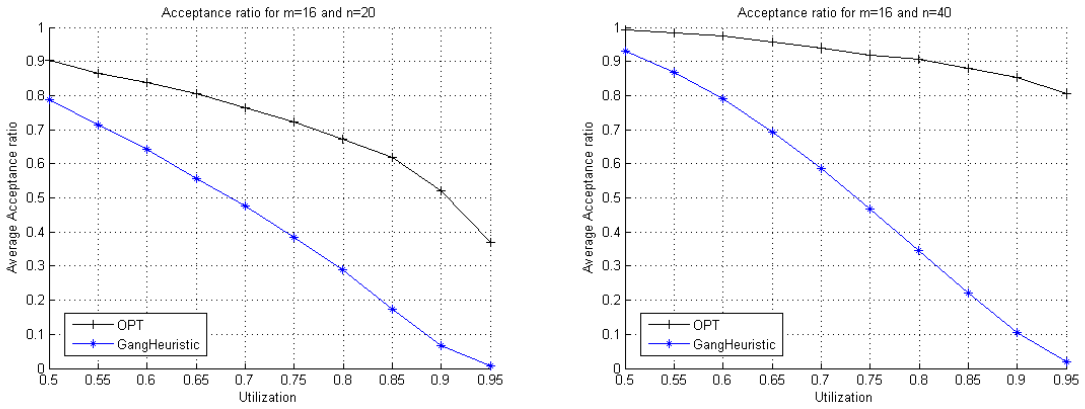
6.4 Average error

We also compare the optimal and heuristic algorithms according to the average error in comparison with the length of the schedule pattern. Let OPT be the length of the pattern computed by the optimal algorithm and UB be the corresponding upper bound computed by the gang heuristic, the error is defined by: $\text{err} = (\text{UB} - \text{OPT})/\text{OPT}$. Due to Theorem 11, we verify that: $\text{OPT} \leq \text{UB} \leq (2 - \frac{1}{m})\text{OPT}$. Hence, the error is bounded by $\text{err} \leq (1 - 1/m)$.

We perform comparison of algorithms for several numbers of tasks and processors that are depicted in Figure 8. As for the acceptance ratio, simulations for computing the average error have been replicated 10000 times. In these graphs, the y-axis is delimited by the worst-case error of $(1 - 1/m)$. First, the average error is not sensitive to the utilization factor but only to the number of tasks. Precisely, the average ratio compare the schedule lengths of the patterns computed by OPT and Gang-h. Varying utilization factors of synthetic task sets will define quite similar pattern shapes that lead to quite similar average error. Second, when the number of tasks increases, the average error also increases and but is still under half of the worst-case error.

7 Conclusion

In this paper we considered the preemptive scheduling of implicit-deadline periodic gang task systems upon identical multiprocessors. We proposed two algorithms which define static patterns that are stretched at run-time in a DP-Fair way. The first one is optimal and runs in polynomial time for a fixed number of processors; the second one is a sub-optimal fixed-priority rule but it is competitive under resource augmentation analysis. Precisely, the speedup factor of the heuristic is bounded by $(2 - \frac{1}{m})$. Our numerical experiments show that the optimal pattern can be computed efficiently up to 60 tasks and ensures a high acceptance ratio when the number of tasks is not too small. For larger systems ($m >> 16$ or $n > 64$), computing an optimal pattern becomes a hard combinatorial problems. In these cases as for most hard combinatorial problems, we think that heuristics (e.g., gang-h) must be used rather than an optimal algorithm. Concerning the proposed gang heuristic, the experiments show that the acceptance ratio decreases quasi-linearly according to the platform utilization factor, but the average error with respect to the optimal pattern length is less than 40%.



(a) Acceptance ratio for $n = 20$

(b) Acceptance ratio for $n = 40$

Figure 7 Acceptance ratio of the LP-based optimal algorithm and Gang Heuristic.

Future Work. Future work will concern the definition of a pattern schedule that aims to reduce the number of preemptions. This latter problem seems to be hard to cope with but still significant for allowing practical applications of real-time scheduling methods. As we said in the discussion Section 5.3 the case of sporadic task is left as future work.

References

1 Björn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In Roberto Baldoni, Paola Flocchini, and Binoy Ravindran, editors, *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, volume 7702 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2012. doi:10.1007/978-3-642-35476-2_2.

2 Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*, pages 63–72. IEEE Computer Society, 2012. doi:10.1109/RTSS.2012.59.

3 Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. doi:10.1007/BF01940883.

4 V. Berten, P. Courbin, and J. Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *5th Junior Researcher Workshop on Real-Time Computing*, pages 9–12, 2011. URL: http://arxiv.org/abs/0805.3237.

5 Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Computers*, 35(5):389–393, 1986. doi:10.1109/TC.1986.1676781.

6 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 225–233. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.32.

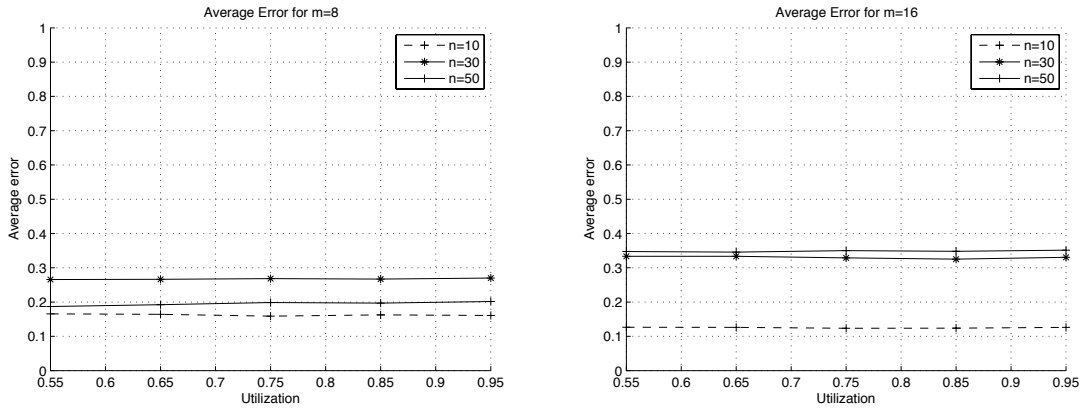
7 Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, chapter Scheduling Parallel Jobs on Clusters, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

8 Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. URL: http://dl.acm.org/citation.cfm?id=355074.

9 Rohit Chandra, D Leonaldo, Kohr Dave, Maydan Dror, M Jeff, and Menon Ramesh. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001. URL: http://lib.mdp.ac.id/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf.

10 Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 101–110. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.10.

11 Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time



(a) Average error for $m = 8$

(b) Average error for $m = 16$

Figure 8 Average error on the schedule pattern length for the LP-based optimal algorithm and gang heuristic.

scheduling theory. *Inf. Process. Lett.*, 106(5):180–187, 2008. doi:10.1016/j.ipl.2007.11.014.

12 Pierre Courbin, Irina Iulia Lupu, and Joël Goossens. Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks. *Real-Time Systems*, 49(2):239–266, 2013. doi:10.1007/s11241-012-9173-x.

13 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

14 M. Drozdowski. *Handbook of Scheduling Algorithms, Models and Performance Analysis*, chapter Scheduling Parallel Tasks — Algorithms and Complexity. Chapman & Hall/CRC, 2004.

15 Maciej Drozdowsli. On the complexity of multiprocessor task scheduling. *Bulletin of the polish academy of sciences*, 43(3):381–392, 1995. URL: http://www.cs.put.poznan.pl/mdrozdowski/txt/BullPAS95.pdf.

16 P.-F. Dutot, G. Mounie, and Denis Trystram. *Handbook of Scheduling Algorithms, Models and Performance Analysis*, chapter Scheduling Parallel Tasks — Approximation Algorithms. Chapman & Hall/CRC, 2004.

17 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor task sets. In *In proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010. URL: http://retis.sssup.it/waters2010/waters2010.pdf.

18 Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 13–22. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.15.

19 Shelby Funk. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Systems*, 46(3):332–359, 2010. doi:10.1007/s11241-010-9109-2.

20 Shelby Funk, Greg Levin, Caitlin Sadowski, Ian Pye, and Scott A. Brandt. Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47(5):389–429, 2011. doi:10.1007/s11241-011-9130-0.

21 Shelby Funk and Vijaykant Nanadur. LRE-TL: An optimal multiprocessor scheduling algorithm for sporadic task sets. In *17th International Conference on Real-Time and Network Systems*, pages 159–168, 2009. URL: https://hal.inria.fr/inria-00442002/document.

22 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

23 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. URL: http://www.netlib.org/pvm3/book/pvm-book.html.

24 Joël Goossens and Pascal Richard. *Real-time Systems Scheduling*, volume Fundamentals, chapter Multiprocessor Architecture Solutions. Wiley-ISTE, September 2014. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1848216653.html.

25 Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. *CoRR*, abs/1006.2617, 2010. URL: http://arxiv.org/abs/1006.2617.

26 Sergei Gorlatch and Holger Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998. doi:10.1142/S0129626498000456.

27 R L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:416–426, 1966. URL: http://www.jstor.org/stable/2099572.

28 William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MIT Press, 2nd edition, 1999. URL: http://mitpress.mit.edu/books/using-mpi.

29 Berit Johannes. Scheduling parallel jobs to minimize the makespan. *J. Scheduling*, 9(5):433–452, 2006. doi:10.1007/s10951-006-8497-6.

30 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 459–468. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.42.

31 Leonid G Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980. URL: http://www.sciencedirect.com/science/article/pii/0041555380900610.

32 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, pages 259–268. IEEE Computer Society, 2010. doi:10.1109/RTSS.2010.42.

33 Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott A. Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 3–13. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.34.

34 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Outstanding paper award: Analysis of global EDF for parallel tasks. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 3–13. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.12.

35 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.

36 Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luís Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 3. ACM, 2014. doi:10.1145/2659787.2659815.

37 R McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1), 1959. URL: http://www.jstor.org/stable/2627472.

38 Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 321–330. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.37.

39 Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 217–226. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.27.

40 Vaidy S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency - Practice and Experience*, 2(4):315–339, 1990. doi:10.1002/cpe.4330020404.

41 David W Walker and Jack J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

42 Dakai Zhu, Daniel Mossé, and Rami G. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 142–151. IEEE Computer Society, 2003. doi:10.1109/REAL.2003.1253262.

A Survey on Static Cache Analysis for Real-Time Systems

Mingsong Lv[1], Nan Guan[2], Jan Reineke[3], Reinhard Wilhelm[4], and Wang Yi[5]

1 **Northeastern University, China**
lumingsong@cse.neu.edu.cn
2 **Northeastern University, China**
guannan@ise.neu.edu.cn
3 **Saarland University, Germany**
http://orcid.org/0000-0002-3459-2214
reineke@cs.uni-saarland.de
4 **Saarland University, Germany**
http://orcid.org/0000-0002-5599-7560
wilhelm@cs.uni-saarland.de
5 **Uppsala University, Sweden**
yi@it.uu.se

Abstract

Real-time systems are reactive computer systems that must produce their reaction to a stimulus within given time bounds. A vital verification requirement is to estimate the Worst-Case Execution Time (WCET) of programs. These estimates are then used to predict the timing behavior of the overall system. The execution time of a program heavily depends on the underlying hardware, among which cache has the biggest influence. Analyzing cache behavior is very challenging due to the versatile cache features and complex execution environment. This article provides a survey on static cache analysis for real-time systems. We first present the challenges and static analysis techniques for independent programs with respect to different cache features. Then, the discussion is extended to cache analysis in complex execution environment, followed by a survey of existing tools based on static techniques for cache analysis. An outlook for future research is provided at last.

2012 ACM Subject Classification Surveys and overviews
Keywords and phrases Hard real-time, cache analysis, worst-case execution time
Digital Object Identifier 10.4230/LITES-v003-i001-a005
Received 2015-04-23 **Accepted** 2016-05-11 **Published** 2016-06-29

1 Introduction

Real-time embedded systems not only exist in industry domains, such as automotive electronics, avionics, telecommunication, medical systems, etc., but are deeply immersed in our everyday life due to the rapid progress of mobile and embedded technology. A real-time system should not only provide logically correct functionality, but moreover, it must meet *timing requirements* as stated in the system specification [21]. In hard real-time systems, such as aerospace systems, a timing error may result in catastrophic consequences. A major task in real-time system verification is to analyze the timing behavior of the system before deployment in order to guarantee that no timing violation occurs at run time.

A real-time system is typically composed of many tasks that cooperate to provide the required functionality. To verify the satisfaction of the timing requirements of the system, one must first know how long each task (or program) may execute. However, this is not an easy problem, because

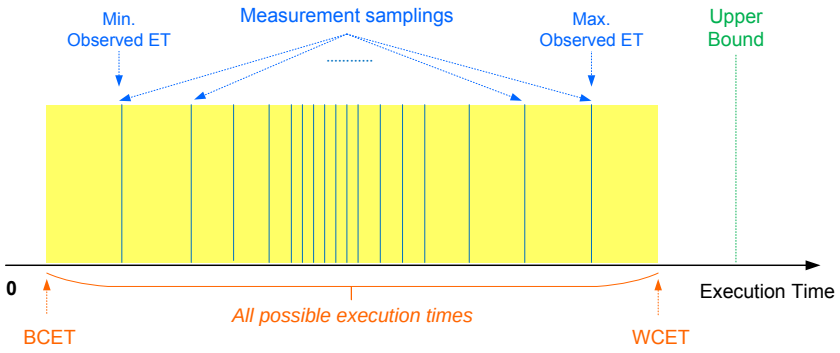


Figure 1 The distribution of execution times of a program.

the execution time of a program may vary widely as a result of many complex factors, such as data inputs, hardware features, execution contexts, etc. Among all the possible execution times (represented by the yellow range in Figure 1), the minimum and the maximum are called the Best-Case Execution Time (BCET) and the Worst-Case Execution Time (WCET), respectively. The main objective of program-level timing analysis is to estimate the WCET [130], which is then used in system-level timing analysis, such as a schedulability analysis [34, 115].

The common practice in industry has been, and partly still is, to *measure* the end-to-end execution latency of a task [127], by sampling its executions in different scenarios (depicted as the blue vertical lines in Figure 1). The maximal observed execution time increased by a safety margin is used as the WCET estimate of the measured program. This approach is called *dynamic timing analysis* in the real-time community. However, the worst case is not guaranteed to be covered by measurements. Thus, the observed WCET is in general an *underestimation* of the actual WCET. Analytical methods that cover all possible execution scenarios (without executing the analyzed program) and provide safe upper bounds on the WCET are desirable for hard real-time systems. They are usually called *static timing analysis*.

Unfortunately, such upper bounds cannot be easily estimated due to both the complexity of the program itself and the uncertainty from the execution environment. The program may execute different control flow paths depending on input, and these different paths may need different execution times. The execution platform may exhibit a dependence of the execution time of instructions on the execution state of the platform. This *execution state* consists of the occupancy of the platform resources. For example, an instruction may exhibit very different execution times depending on whether instruction or operand fetches hit or miss the cache. The execution environment, finally, may interfere with a program's execution by preempting its execution and thereby increasing the program's response time. Hence, these three factors all have an impact on the program's execution time.

Exhaustive exploration of the combined space of control flow paths and paths through the architectural state space is infeasible due to the size of this space. A conservative abstraction of the execution platform is typically used in static timing analysis to increase efficiency. This abstraction may adversely lead to an *overestimated* WCET. Efforts have to be exerted to reduce the overestimation as much as possible, to avoid the need to over-provision system resources.

All approaches to static timing analysis compute bounds on the execution times of a program starting with bounds on the execution times of individual instructions occurring at points in the program. Their execution times typically depend on the execution state of the platform. Depending on this state, an instruction's execution may suffer from *timing accidents*, which may increase the execution time by their associated *timing penalties*. For example, a memory

access may suffer from a cache miss, which increases its execution time by the cache miss penalty. The actual execution state is the result of the execution history. Different control flow paths through the program, in general, result in different execution states and may thus exhibit different execution times. A classification of an occurrence of a memory access as a cache hit or a cache miss must hold for all executions of this memory access.[1]

Static timing analyses determine an invariant for each program point that describes all possible execution states when control reaches this program point. Such an invariant allows excluding many timing accidents, such as cache misses, pipeline stalls, etc., and safely allow subtracting the associated timing penalties from the worst-case upper bound on the execution times.

Among the hardware features to consider in timing analysis, *caches* have the biggest influence on execution time [59]. A precise analysis of the cache behavior does therefore have a great impact on the precision of the overall WCET estimation.

Cache is a small on-chip memory to bridge the speed gap between the processing unit and the much slower off-chip memory by storing a portion of the content from main memory. If a data request *hits* in the cache, it takes only very few processor cycles to deliver the data from the cache to the processing unit; otherwise, in the case of a *miss*, the CPU has to fetch the data from main memory, which nowadays consumes hundreds of processor cycles.

The role of cache analysis for WCET estimation is to predict the behavior of a program on the platform's caches. For example, cache analysis may provide a safe bound on the number of cache hits or misses when a program executes on some given platform; it may also categorize the accesses to memory blocks in programs as definite hits or misses.

In [130], WCET analysis techniques and tools are surveyed. Due to its importance in timing analysis and its complexity, cache analysis alone deserves an in-depth discussion.

The rest of the article is organized as follows. First, we give background knowledge on WCET estimation and caches in Sec. 2. Then, we present the problems and solutions for the intensively researched LRU caches in Sec. 3. A survey of the results on non-LRU caches is provided in Sec. 4. In Sec. 5, the discussions are extended to cache analysis in multi-tasking and multi-core environments where programs interfere with each other on shared caches. A summary of WCET analysis tools based on static cache analysis is given in Sec. 6. We present an outlook for future research at last.

2 Background Knowledge

We first present an established static WCET analysis framework to exhibit its main work flow and where cache analysis steps in. Then, the basic concepts on cache organization, behavior and analysis are introduced.

2.1 A Classical WCET Analysis Framework

The objective of static timing analysis is to compute safe lower and upper bounds on the execution times of programs. These are also called BCET and WCET estimates, respectively. The WCET is observed in a particular execution scenario with some execution context, such as data input and initial hardware state. Theoretically, the WCET is not computable; otherwise, one could solve the halting problem. In this article, we assume that all real-time programs terminate so that their WCET can be computed.

[1] Note that this is a slight simplification to ease understanding. Later on we will explain why it may make sense to partition sets of memory accesses by *contexts*. The distinction between the occurrence of a memory access or an instruction and one, several, or all of their executions is of utmost importance.

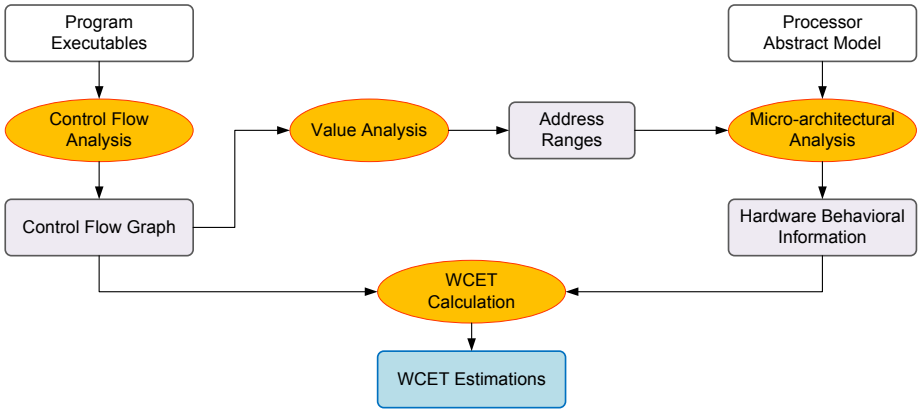


Figure 2 The separated path and cache analyses framework for WCET estimation.

Most static analyses are performed on the binary code rather than the source code of the program, because the two need not have the same control flow due to compiler optimizations, and the source code does not determine the precise location of instructions and program variables in memory, which are needed for instruction- and data-cache analysis.

A naïve, straightforward analysis would enumerate all possible executions to find the largest execution time. However, this method does not scale. Consider a loop with a conditional branch inside. If we do not know whether or not the branch is taken in each iteration, the number of program paths to be explored is exponential with respect to the number of loop iterations. To tackle the complexity, the state-of-the-art analysis techniques adopt the framework in Figure 2.

The first step is to reconstruct the Control Flow Graph (CFG) of the program. A CFG is a directed graph, with each vertex representing an instruction and each edge representing the control flow. We say there is a *program point* right before each vertex in the following discussion. A CFG typically has a single entry and a single exit corresponding to the start and the end of the program. The analysis is then conducted on the CFG. In some work [93], WCET analysis is conducted in a modular way on program functions to reduce analysis overhead.

This step is followed by a *Value Analysis*, which computes enclosing intervals for all potential values of registers and local variables and also determines loop bounds. This is a more or less standard *Interval Analysis* as invented by P. and R. Cousot [30]. The next step is to compute an upper bound on the execution time of each instruction (C_i in Equation 1), which heavily depends on the underlying hardware features, such as pipelines [69], branch predictors [18, 28] and caches. *Cache analysis* is an important part of this step, which is often referred to as *micro-architectural analysis* or *low-level analysis*.

With the above results, the final task is to find the execution path that exhibits the longest execution time, typically referred to as WCET calculation. An established approach is the *Implicit Path Enumeration Technique* (IPET) [71], the main idea of which is to transform the problem of searching the worst-case execution path into searching the execution counts for each instruction such that the execution time is the largest. This can be formally modeled as an integer linear programming (ILP) problem, in which the execution time of a program is represented by the sum of execution latencies of all instructions. Thus, the WCET can be obtained by maximizing the execution time (the objective function of the ILP problem):

$$WCET = max \sum_{i=1}^{N} C_i \times X_i \tag{1}$$

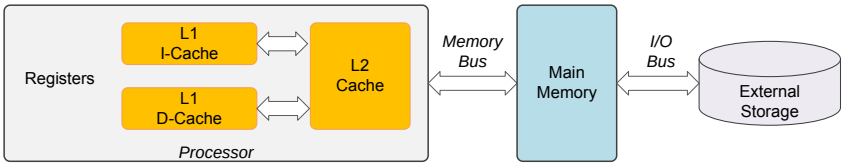


Figure 3 A common memory architecture.

In Equation (1), N refers to the total number of instructions, which is a constant obtained from the CFG; C_i is the WCET bound for the i^{th} instruction, which has been computed in the third step; the variable X_i stands for the execution count of the i^{th} instruction. X_i is subject to constraints induced by the program structure. In the following, the variable d_{i_j} captures how often the edge from instruction i to instruction j is taken. Then, X_i must be equal to both the total execution counts of all its incoming edges and those of all outgoing edges[2], which can be expressed as follows.

$$\forall i, X_i = \sum_{all\ incoming\ edges} d_{*\_i} = \sum_{all\ outgoing\ edges} d_{i\_*} \tag{2}$$

Other program behavior can be constrained as well. For example, the loop iterations should be bounded in advance either manually or by automatic analysis [49]. They can be modeled as linear functions relating the execution counts of the loop body and the loop entry. All available constraints are expressed in one ILP problem, whose maximal solution bounds the WCET from above. To improve analysis efficiency, sequences of instructions (with no branch along the path) are combined into *basic blocks* and represented by a single vertex in the CFG.

The key feature of this framework is the separation of micro-architectural analysis from WCET calculation. In general, this approach is pessimistic. However, the sacrifice of precision is rewarded by a significant improvement in analysis efficiency.

2.2 Cache Organization, Behavior and Analysis

Cache is a small, high-speed memory residing on the processor chip (shown in Figure 3) that stores a copy of a portion of the instructions and/or data in main memory. Each access to the cache results in either a *hit* or a *miss*. One can distinguish two types of cache misses. A *cold miss* occurs when a data element, absent from the cache, is loaded for the first time. If the cache is full and a cache miss occurs, a data element needs to be evicted. A *replacement miss* occurs when an evicted element is reused. Cache hits are the result of *memory reuse*.

2.2.1 Cache Organization and Behavior

In most processors, a cache line (the unit for cache access) contains multiple data elements. An access to one element causes the whole cache line to be loaded into the cache. As a result, a following access to another element of the same cache line also results in a cache hit. Besides, consecutive accesses to the same data element result in cache hits as well, an example of which is the execution of a loop. The above two types of reuses are commonly referred to as *spatial reuse* and *temporal reuse*, respectively, the pervasiveness of which is expressed by the well-known *locality principle*.

[2] For either the entry or the exit instruction, one can simply constraint the execution count to be exactly 1.

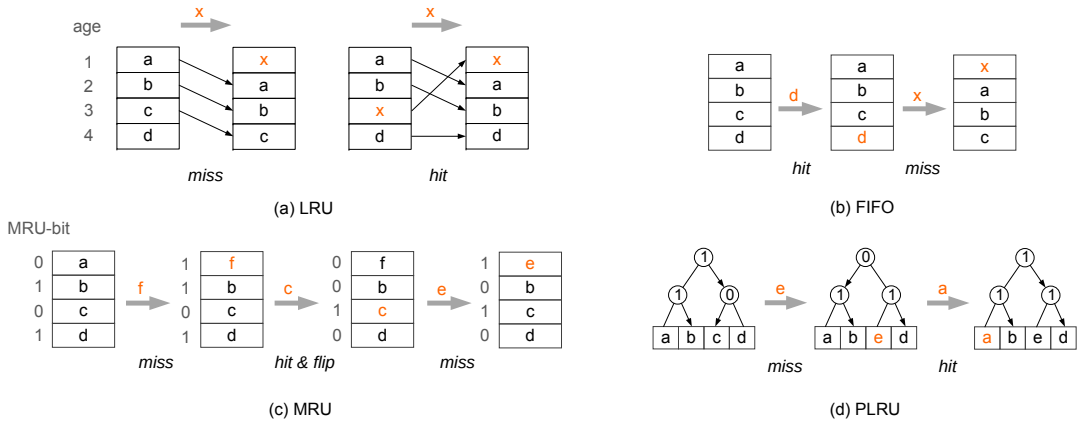


Figure 4 Common cache replacement policies.

Some processors are equipped with two or more levels of caches, as a fine-grained trade-off between cost and speed. The lowest level[3] (namely L1 cache) is usually divided into a private instruction cache and a private data cache, each of which is typically no larger than 32 KB and has an access latency of $1-2$ cycles. If a memory access misses in the L1 cache, the L2 cache is queried. The capacity of L2 caches may range from hundreds of KB to several MB, with an access latency of around 10 cycles. In some high performance multi-core processors, an L3 cache may also be deployed to further expand cache capacity. Misses in the last level cache trigger accesses to the main memory via the *off-chip* memory bus, causing a delay in the order of hundreds of cycles.

Like other storage devices, addressing is an important feature of the cache design. Some processors adopt the *set-associative* organization, in which the address space is partitioned into independent *sets*. Every set has a fixed number of *ways*, each of which refers to a single cache line in every cache set. The total number of ways within a cache set is called *associativity*. To load a memory block, the processor first determines which cache set the block maps to. Then a lookup into the target set is performed for a free cache way. If all the cache ways are occupied, a *replacement policy* determines which old block to evict to make room for the new block. In this article, we consider four common policies illustrated in Figure 4, assuming a 4-way cache set.

The least-recently-used (LRU) policy replaces the block that has been used least recently. The illustration of LRU in Figure 4(a) is a first abstraction from the actual hardware implementation. Each cache way in an LRU cache set is associated with a fixed age, which is received by the block in the corresponding cache way. Figure 4(a) illustrates how the positions of the blocks are reordered upon a cache hit and a cache miss.

However, most commercial processors do not employ LRU, because it requires complex hardware implementation and further leads to higher power consumption. Non-LRU replacement policies, such as First-In-First-Out (FIFO), Most-Recently-Used (MRU) and Pseudo-LRU (PLRU), are adopted instead since they are simple to implement and still have similar average performance as LRU [58].

Figure 4(b) shows how the FIFO replacement policy works. A cache hit does not change the cache state. Upon a cache miss, all the memory blocks shift one position downwards, evicting the block in the bottom cache way; then the new block is installed in the top-most cache way. Again, this representation is an abstraction from the actual hardware implementation, which does not

[3] A cache level is lower if it is closer to the processing unit; the highest level is typically called the last level.

shift memory blocks from one cache line to another, but rather maintains a modulo-4 counter to determine the next block to replace.

The MRU cache (shown in Figure 4(c)) maintains a bit for each cache way (called MRU-bit) to approximate the recency of access. Bit 1 means the block was visited recently. Upon a hit, the MRU-bit of the hit block is set to 1. Upon a miss, the top-most way with MRU-bit 0 is taken by the new block, and its MRU-bit is set to 1. Eventually there is only one cache way with MRU-bit 0. When this way is visited—in the example the access to c—the MRU-bit is turned to 1, and all the other MRU-bits are set to 0. This is called a *global flip*.

PLRU is a tree-based approximation of LRU (Figure 4(d)). It arranges the cache ways at the leaves of a binary tree with $k-1$ bits, where k is the cache associativity. Bit 0 and 1 on the branches indicate the left and the right subtrees, respectively. Following the bits downwards from the root, the cache line to be replaced or refilled can be found. After an access (either hit or miss) to a cache way, all tree bits along the path from it to the root are set to point away. It is possible that a cache set contains invalid cache lines. We assume the *tree-fill* policy, by which the line to be filled or replaced is always determined by the tree bits.

In most architectures, cache sets are completely independent of each other. This makes the independent analysis of programs' behavior on different cache sets possible. Throughout this article, we focus on the cache behavior in one set, and may use *cache* to refer to a cache set to simplify the presentation when appropriate.

2.2.2 Cache Analysis

The objective of cache analysis is to statically determine the cache behavior of a program. Its results can be used for performance analysis and optimization. The results may be of several types: one is the *classification of individual memory accesses in a program as hits or misses*. Such a classification of memory accesses can be used in a cooperating pipeline analysis to determine whether the pipeline may have to stall on an instruction or operand fetch. Another is a *bound on the number of cache loads in a segment of the program*. This allows, under certain conditions, just adding an accumulated penalty to the execution-time bound for memory accesses that could be neither classified as cache hits or misses. The former type of cache analysis could be called a *classifying cache analysis*, one instance of the latter, relevant for practice, is known as *persistence analysis*.

A typical use of the results of a cache analysis in real-time systems is estimating the BCET and WCET of programs. The bigger the percentage of hits that will happen during execution it can predict, the tighter the WCET estimation is. On the other hand, predicting a higher percentage of actual misses leads to tighter BCET estimation.

The designer of a cache analysis faces several questions: the first one is whether the analysis is to be a classifying or a persistence analysis. The second question is by which method cache behavior should be analyzed. Associated with the second question are the questions of the granularity of the analysis and the representation of the cache behavior properties.

Let us illustrate this rather abstract discussion with the example of classifying cache analyses. The most precise analysis would predict each *executed memory access* to be either a hit or a miss—here it is vital to make the difference between an *executed memory access* and the *occurrence of an instruction involving a memory access* in a program. We have assumed that all real-time programs terminate. Hence, any program would execute only finitely many memory accesses, so that such an analysis would in principle be possible. However, the corresponding analysis would, in general, not scale. On the other end are cache analyses that would classify *occurrences of memory accesses* as *always hit* or *always miss*, where *always* means for all executions of this occurrence of the memory access. However, experience has shown that the actual execution times of memory

accesses associated with one occurrence of a memory access may vary widely. This means that just taking their upper bounds may largely overestimate the memory-access costs. Precise and efficient analyses should attempt to classify subsets of executed memory accesses corresponding to one occurrence, such that the accesses in the subsets have some homogeneous timing behavior. The subsets would be characterized through control flow criteria, in the following called *contexts*. The most important examples for contexts are different iterations of loops.

To approach the second question raised above, one needs to identify the information about cache contents—in the following mostly called *concrete cache states* (CCS)—to be computed by a classifying cache analysis. This information would provide answers to the question, *are all memory accesses belonging to this occurrence (in this context) hits or are they all misses?* One solution would be to collect at each program point the set S of all concrete cache states that are possible when program control reaches this program point (in this context). Such an analysis would again not scale. Instead, one can represent sets of concrete cache states by *abstract cache states* (ACS). Each abstract cache state (compactly) represents a set of concrete cache states. As we will see later, two types of such abstract cache states are of interest. Consider the set S of all concrete cache states that are possible at a program point. An *abstract Must cache state* will represent the information: which memory blocks will be in each of the possible concrete cache states in S. This is obtained by some kind of intersection applied to the elements in S. Likewise, an *abstract May cache state* will represent the information: which memory blocks may be in one of the concrete cache states in S. This is obtained by some kind of union applied to the elements in S.

3 Analysis of LRU Caches

For decades, a majority of research on cache analysis has focused on caches with LRU replacement strategy. In this section, we survey the main analysis techniques with an emphasis on the approach based on Abstract Interpretation (AI) [38]. This technique is realized in the aiT tool of AbsInt [57], which is widely used in industry. Since programs spend most of their execution time in loops, a sub-section is dedicated to the analysis of the cache behavior in loops. The big picture on LRU caches is completed with further discussions on data cache and multi-level cache analyses.

3.1 Abstract-Interpretation-Based Approaches

The first cache analysis based on abstract interpretation (AI) was proposed by Ferdinand and Wilhelm in the 1990s [1, 38]. The overall approach works in two phases:

1. An AI-based cache analysis computes abstract cache states at all program points as part of a fixed-point solution;
2. These abstract cache states are queried in order to classify memory accesses.

3.1.1 A Short Introduction to Abstract Interpretation

Abstract interpretation [30] is a static program analysis method based on a semantics of the considered programming language. Instead of executing the program on the concrete domain of values, it executes an abstracted version of the program on an abstract domain of descriptions of values. In the case of cache analysis, the program abstraction only describes the memory-access behavior of the program, i.e., it performs all memory accesses that the program would execute. This abstracted program works on *abstract cache states*, which are descriptions of sets of concrete cache states. One abstract cache state is associated with each program point. Whenever the analysis encounters a memory access, it updates the abstract cache state in a way induced by the update that the processor would perform on the concrete cache states. Whenever the control flow

Figure 5 An example to show the $\sqsubseteq$ relation w.r.t. the Must domain.

of the program merges, e.g. at the end of a conditional or at the header of a loop, it combines the incoming abstract cache states in a sound way.

Gary Kildall [64] has recognized that the abstract domains of typical data-flow analyses are lattices, i.e., partially ordered sets where all subsets have least upper bounds. The partial order reflects the relative information content of two lattice elements. By convention, elements lower in the lattice represent more information than information higher in the lattice, i.e., an element a below or equal to an element b in the lattice, $a \sqsubseteq b$, contains no worse information than b. The domain of abstract cache states together with a partial order reflecting the amount of knowledge about cache contents, in this sense, also forms a lattice.

Consider two abstract Must cache states $\hat{a}$ and $\hat{b}$ (as shown in Figure 5). Abstract cache state $\hat{a}$ represents just one concrete cache state, containing memory blocks $\{u, x, y, z\}$ with the ages 1, 2, 3, 4. Abstract cache state $\hat{b}$ represents the set of concrete cache states with memory block u having age 1, x having age at most 3, z having age at most 4, and possibly one more (unknown) block at age 2, 3, or 4. $\hat{a} \sqsubseteq \hat{b}$ means that all the concrete cache states represented by $\hat{a}$ are also represented by $\hat{b}$. In particular, this implies that all the memory blocks known to be contained in the concrete cache states described by $\hat{b}$, in the example above u, x, z, are also known to be contained in the concrete cache states described by $\hat{a}$. Furthermore, $\hat{a}$ additionally tells us that, (1) block y is guaranteed to be in the cache while $\hat{b}$ does not; (2) the age upper bound estimated for block x in $\hat{a}$ is smaller than that in $\hat{b}$. Clearly, the abstract cache state $\hat{a}$ contains better information than $\hat{b}$. As stated above, at control flow merge points cache analysis must combine the incoming information in a sound way. The operation applied to the incoming abstract cache states is the least upper bound, $\sqcup$, of the lattice. This is shown in Figure 6. As said above and made more precise later, it is some form of intersection. It determines (the best) safe information holding for all incoming paths.

The lowest element in the lattice of abstract cache states, $\bot$, called *bottom*, describes the empty set of concrete cache states. It is the initial analysis information at all program points but program entry. If we do not have any information about the cache contents at program entry, the highest lattice element, $\top$, called *top*, is used as initial analysis information. It describes the set of all concrete cache states, and thus the absence of information about cache contents.

The update functions are, in general, monotone, so that information, once computed, is not lost again. A fixed-point iteration over the control flow graph of the program is guaranteed to terminate and deliver the least fixed point as solution. Essential for termination is the finiteness of the lattice.

Let us summarize this short introduction to AI by listing the main ingredients of a particular abstract interpretation. The designer needs to choose or define an *abstract domain*, a lattice of abstract values, which are descriptions of (sets of) concrete values. The *partial order* defines the relative information content of two lattice elements. The *least upper bound* is the operation to join abstract values flowing to a program node through different control flow graph edges. *Abstract update functions* for an instruction reflect the instruction's effect on the incoming abstract values.

Table 1 Cache Hit/Miss Classification.

Classification	Cache Access Behaviors Described	Analysis
Always Hit (AH)	Block is guaranteed to be in the cache upon each memory access	Must
Always Miss (AM)	Block is guaranteed not to be in the cache upon each memory access	May
Not Classified (NC)	Cannot be classified by any of the above classifications	/

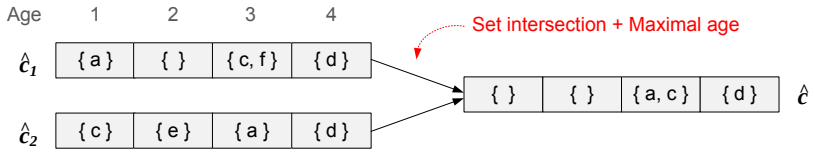


Figure 6 An example to demonstrate the Must join function.

3.1.2 Classification of Memory Accesses

Querying an abstract cache state, resulted from a cache analysis, for an accessed memory block may yield qualitative properties as listed in Table 1. To determine whether the memory access to m is *always hit* (AH), one simply checks the existence of m in the abstract Must cache reaching its program point. Similarly, to determine whether the memory access to m is *always miss* (AM), it suffices to know that m is not in the abstract may cache reaching its program point. If a memory access can be neither classified as AH nor as AM, it is classified as NC. An NC classification can have two reasons: (1) some of the executions of a memory access hit in the cache and others miss in the cache, or (2) the analysis method overapproximates the set of concrete cache states and thereby fails to deliver the correct classification. Research results show that these properties are able to cover most access behaviors for LRU caches [38]. The classifications can then be expressed as linear constraints on the execution cost of each instruction (basically each instruction generates a single memory access) and later integrated into a WCET estimation. In architectures without timing anomalies [22, 106], if the classification of a memory access is NC, it is safe to treat it as AM. The properties AH and AM in Table 1 are explored by independent analyses, which are now described in detail.

3.1.3 Must Analysis

The objective of a Must analysis is to compute a Must-ACS at each program point, *which represents the common cache contents in all possible executions leading to this program point.* An age is associated with each memory block in the Must-ACS, which is an upper bound of its ages in all CCS. We use a graphical representation to show a Must-ACS, in which blocks are grouped according to their ages, e.g., in Figure 6. The set of CCS represented by a given Must-ACS is formally defined by the *concretization function* below, where c and $\hat{c}$ denote concrete and abstract cache states, respectively, and $age(c, m)$ refers to m's age in cache state c (applies to both concrete and abstract states).

$$conc^{Must}(\hat{c}) = \{c \mid \forall m \in \hat{c} \ : \ m \in c \ \wedge \ age(c, m) \leq age(\hat{c}, m)\} \tag{3}$$

The Must-ACS at the program entry is initialized with $\top$ representing all concrete cache states if the initial cache content is unknown; all other nodes are initialized with $\bot$ representing the empty set of concrete cache states. A fixed-point computation is employed to compute the Must-ACS at each program point, during which two main operations over the Must-ACS are involved.

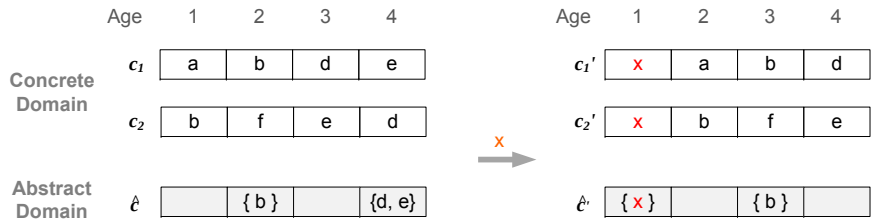


Figure 7 An example to demonstrate the Must update function.

A *join function* combines several Must-ACS into a single Must-ACS when the control flow merges. The resulting Must-ACS takes the intersection of the sets of blocks in all the incoming states and assigns to each block its maximal age from the incoming states, as shown in Figure 6. An *update function* $\hat{U}(\hat{c}, x)$ defines how an abstract state $\hat{c}$ is changed due to an access to memory block x, specifically, how the age of each block in the ACS is updated. Figure 7 shows an example. A correct Must update function guarantees that the age of each block in the computed ACS is a safe age upper bound for all possible represented CCS. For example in Figure 7, the age of d in $\hat{c}$ implies that there could be a CCS represented by $\hat{c}$, in which d has an age of 4, such as c_2. By loading x, we can no longer guarantee that d still stays in any resulting CCS. Thus, d has to be removed from the computed abstract state $\hat{c}'$.

3.1.4 May Analysis

May analysis computes a May-ACS at each program point, which represents all potentially cached contents in all possible executions leading to this program point. If block m does not exist in the May-ACS at the reaching program point, we can guarantee the access to m is AM. Unlike the Must-ACS, the age of each block in a May-ACS is the lower bound of its ages in all represented CCS, as expressed by the May concretization function below, with the same notions as in function (3).

$$conc^{May}(\hat{c}) = \{c \mid \forall m \in c \ : \ m \in \hat{c} \ \wedge \ age(\hat{c}, m) \leq age(c, m)\} \tag{4}$$

The May join function takes the union of the sets of blocks in all incoming May-ACS and assigns each block the minimal age in all incoming states. The May update function is exemplified in Figure 8, where $\hat{c}$ is the May-ACS representing the concrete states c_1 and c_2. Take memory block d for example. d's age in the May-ACS is the minimum of those in c_1 and c_2. After the access to x, d is evicted from c_2. However, d still remains in the resulting May-ACS $\hat{c}'$, because $\hat{c}'$ must soundly represent the other resulting CCS c_1' in which d remains. To determine whether the memory access to m is AM, it suffices to know that m is not in the May-ACS reaching its program point. A block m is not in the final May-ACS at a program point because either m has never been loaded or enough different blocks have been loaded to evict m from the cache. May analysis does not directly help with tighter WCET estimations, however, predicting more misses results in better estimations on BCET.

3.2 Improving Precision by Using Contexts

In practice, merely relying on classifying analyses, such as Must and May analyses, may still largely overestimate the memory-access cost and thus the WCET. Methods to improve the knowledge about the cache behavior are proposed by taking program structures into consideration, in particular loops. The cache behavior of programs in loops is somewhat special: the first iteration typically loads the contents into the cache; later iterations profit from the first iteration since

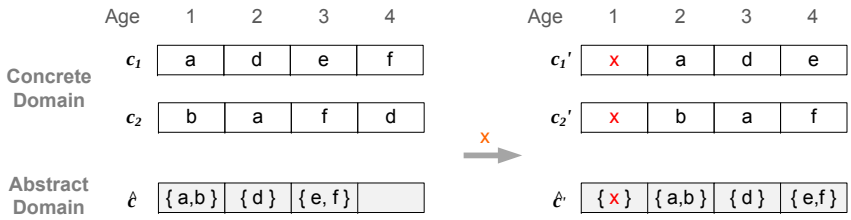


Figure 8 An example to demonstrate the May update function.

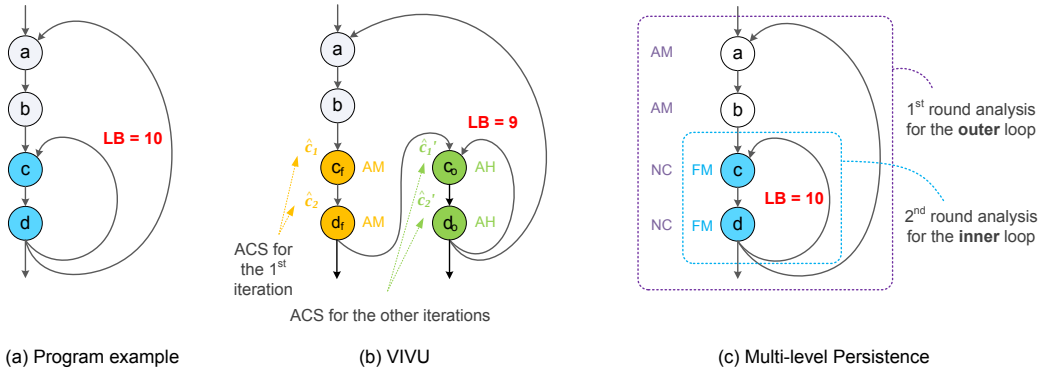


Figure 9 The ideas of VIVU and multi-level Persistence analysis.

accesses to the cached contents are hits. The concrete state on return to the loop header may thus be very different from that reaching the loop from the outside. This is also reflected in cache analysis where the abstract state upon return from the first iteration may be very different from that on the entrance to the loop. Naïvely applying the join function to these two abstract cache states would produce very bad information about the cache behavior of the loop. In such cases, a large percentage of the accesses to the memory blocks in the loop cannot be classified as either AH or AM by the previously introduced Must and May analysis. However, there are two alternative ways to tighten the WCET estimation. The first one exploits the above observation by virtually unrolling each loop followed by a Must analysis. An access to a memory block that in the first iteration would be classified as AM, and that in the other iterations would be classified as AH would then be classified as FM (first miss). The other alternative would be to bound the number of cache misses for all the accesses to a memory block within a certain program scope. These two analysis techniques will be now introduced. Note that the first analysis still is a classifying analysis for memory accesses, albeit with a new classification, FM, and the second is a bounding analysis for memory blocks in a scope.

3.2.1 Virtual Inlining & Virtual Unrolling (VIVU)

VIVU [85] can be used to improve cache analysis precision for loops. The idea is to analyze the first loop iteration separately from all other loop iterations. This is done by virtually unrolling the first iteration of the loop body[4], so as to distinguish the behavior between the two contexts. Then, a Must analysis is applied to the program with the unrolled loop to find the AH memory

[4] (1) The unrolling is called *virtual* since it is done by maintaining separate abstract cache states for the first iteration and the remaining iterations (e.g., $\hat{c}_1$ and $\hat{c}_1'$ in Figure 9(b)). For ease of understanding, we use a physically unrolled CFG to show the effects. (2) VIVU allows to unroll more than one iteration of the loop since iterations other than the first may have vastly different behavior and thus execution times. Here we assume only unrolling the first iteration to simplify presentation.

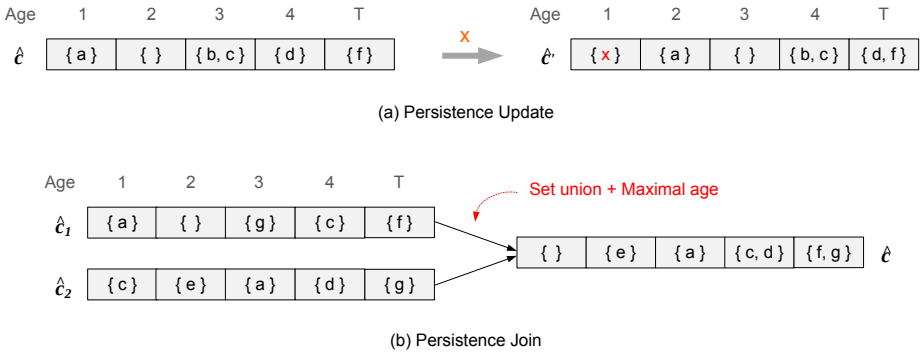


Figure 10 Operations for Persistence analysis.

accesses in all but the first loop iterations. Figure 9(b) shows the results of unrolling the inner loop of the program in Figure 9(a). In the new CFG, c_f and c_o refer to the first and the other iterations of the inner loop, respectively (similar for d). Assume the cache is 2-way associative. Must analysis on the new CFG is able to classify c_o and d_o as AH, and thus c and d as FM.

3.2.2 Persistence Analysis

One can aim at the same analysis objective by a Persistence analysis. There are several possible notions of persistence of memory blocks one could aim at. These are:

persistence execution causes at most one miss for the memory block,

first miss only the first access is a miss, all others are hits,

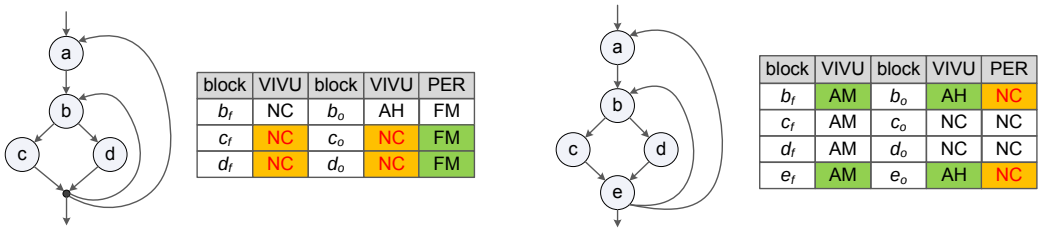
no eviction the block is never evicted after a possible miss.

For a memory block that is persistent in a program fragment, timing analysis can assume a bound of one cache load for all accesses within that program fragment.

The first Persistence analysis was proposed by Mueller et al. [88, 89] for computing First Miss classification of memory references. For set-associative caches, the basic idea of Mueller's approach is to check whether all conflicting instructions in a loop fit into the cache. Later, Ferdinand and Wilhelm [38] proposed a Persistence analysis based on Abstract Interpretation. This analysis employs abstract cache states, Per-ACS, as do the Must and May analyses. The fact that a block has been visited and already evicted from the cache is modeled by assigning an age $\top$ to the block, where $\top$ is larger than the cache associativity. The Per-ACS at each program point represents the cache contents that are potentially visited and then guaranteed to remain in the cache. If a memory block exists in the Per-ACS at the end of the scope, then one can guarantee that at most one cache miss may occur for all the accesses to this block.

The concretization of a Per-ACS is a set of traces satisfying the persistence condition, i.e., at most one miss for each block with a non–$\top$ age in the Per-ACS. More precisely, a Per-ACS captures upper bounds on the ages of memory blocks, assuming that they have already been accessed at least once in the execution of the program.

Figure 10(a) gives an example for Persistence update. All blocks in Per-ACS $\hat{c}$ are potentially visited during program execution. By accessing x, d is no longer guaranteed to be in the cache since it has age 4 in $\hat{c}$, which is maintained by putting d in the $\top$-age line. Block f, already evicted from the cache before accessing x, remains unchanged in the $\top$-age line. The update function for Persistence analysis mainly needs to guarantee the maximal age for each block in the ACS is soundly maintained.



(a) Multi-level Persistence outperforms VIVU

(b) VIVU outperforms multi-level Persistence

Figure 11 Comparing VIVU with multi-level Persistence analysis.

At a control flow merge point, several Per-ACS are merged by the join function, which takes the *union* of the blocks in all the incoming Per-ACS and assigns each block the *maximal* age from the incoming states, as shown in Figure 10(b). Intuitively, the set union operation guarantees the resulting Per-ACS does not lose track of any potentially visited memory block; the maximal age ensures that we can safely predict whether a block is definitely persistent after its first access.

The Persistence analysis by [38] was recently found to be unsafe, due to an error in the update function, which may incorrectly underestimate the age of a block. The error was corrected by Cullmann [31] and Huynh [62]. Several different ways were proposed to restrict the set of memory blocks in a Per-ACS to the actual capacity of a cache set. The simplest way, yielding the least precise results, marks all memory blocks in a Per-ACS as non-persistent, if the Per-ACS contains more blocks than the associativity allows. Others check the number of conflicts between members of the Per-ACS; Huynh's analysis employs fixed-point computation to collect for each block m a set of potentially conflicting blocks (blocks that are mapped to the same cache set with m and thus may age m). If the total number of conflicting blocks is no larger than $A-1$, where A is the cache associativity, then one can safely draw the conclusion that m, once loaded, will persist in the cache. A similar idea was also applied in the Persistence analysis [90] by Mueller. Cullmann presents a number of similar persistence analyses [31]. The most precise of Cullmann's analyses relies on a May analysis to make correct decisions on age update.

3.2.3 Analysis Scope

A bounding analysis, such as the Persistence analysis, is designed to investigate cache behavior within a *program scope*, in most cases a loop body. It is common for a program to have nested loops, where a block in an inner loop also belongs to the outer loop. A natural question would be: does the block have a different cache behavior for different loop levels? To distinguish a block's behavior, the relevant loop nest(s) (the inner loop, the outer loop, or both) is/are unrolled in the VIVU approach. Multi-level approach were proposed by Mueller et al. [128] and Ballabriga and Cassé [10]. Ballabriga and Cassé [10] apply the Persistence analysis of [38] on the relevant scope, here specifically the relevant loop nest, to explore local cache behavior. Figure 9(c) illustrates the basic idea. Persistence properties regarding different loop nests for a memory block can be encoded as linear constraints (or other forms) and integrated into WCET computation for tighter estimations.

3.2.4 Comparing VIVU and Persistence Analysis

Since both VIVU and Persistence analysis are able to bound cache misses for a program scope, a straightforward question would be: which one is more precise? In fact, the two techniques are generally incomparable. For the program in Figure 11(a), c_o and d_o cannot be classified as AH

by Must analysis [38] of the VIVU approach, as neither c_o nor d_o is guaranteed to be accessed in any loop iteration (they belong to two respective conditional branches). On the other hand, multi-level Persistence analysis is successful in this example for c and d. In Figure 11(b), both b_o and e_o can be classified as AH by VIVU. However, none of b, c, d or e can be classified as FM by the multi-level Persistence analysis in the static cache simulation framework [90]. This is because the Persistence analysis [90] counts the conflicting blocks in a loop (mapped to the same cache set); if the number is larger than the cache associativity, none of the blocks in the loop can be classified as FM. If, otherwise, a different Persistence domain is adopted in the multi-level analysis, such as [31], b and e can be locally classified as persistent.

VIVU and Persistence analysis can also be compared in terms of analysis cost. On the one hand, VIVU may result in a more expensive micro-architectural analysis, having to distinguish multiple contexts. On the other hand, the results of Persistence analysis need to be encoded into constraints during implicit path enumeration. The influence of the two effects on analysis times has not yet been compared empirically.

3.3 Other Techniques

3.3.1 Static Cache Simulation (SCS)

There are essentially two approaches preceding AI-based approaches. Mueller et al. [88] developed *static cache simulation* to categorize memory references as *Always hit, First hit, First miss, or Always miss*. They were the first to propose to use *abstract cache states (ACS)*, starting for direct-mapped caches [88].

They later attempted to extend this approach to set-associative caches [89, 90, 128]. For set-associative caches, abstract cache states are defined to hold potentially cached memory blocks at all possible positions, i.e., ages. This results from using set union as join operation at control flow merge points. They also give an update scheme for abstract cache states, which, unfortunately, computes incorrect age lower bounds [89, 128]. The abstract cache states resulted after fixed-point iteration are then used to derive Always hit, First hit, First miss, or Always miss categorizations. It is far from trivial to derive Must information from information contained in their abstract cache states. Spatial locality can be easily exploited. Beyond that there is no obvious way to compute sound and precise Must information. In [90], Mueller employed dominator information in addition to abstract cache states to compute correct Must information.

3.3.2 Cache State Transition Graphs (CSTGs)

Also before AI-based approaches, Li et al. presented a technique that uses *Cache State Transition Graphs* (CSTGs) to model cache behavior [72]. A CSTG, built out of the CFG, models the cache-state transitions for a given cache set. A vertex in the CSTG stands for a possible concrete cache state, and each edge in the CSTG represents a possible transition from the source state to the destination state due to a memory access in the program. Instead of exploring qualitative properties, such as AH, AM and FM, the analysis tries to find a lower bound on cache hits for each memory block. The bounds can be modeled as linear constraints and combined into the ILP to obtain the WCET for the program. By explicitly enumerating the concrete cache states, the CSTG approach can provide good analysis precision. However, it does not scale with program size. Assume that there are M memory blocks mapped to each cache set with associativity K, the number of states in an CSTG can be calculated by $\sum_{i=0}^{K} \frac{M!}{(M-i)!}$ [72]. Note that the number of linear constraints is of the same scale as the number of CSTG states. In practice, the analysis efficiency is low due to the complexity of the resulting ILP problem.

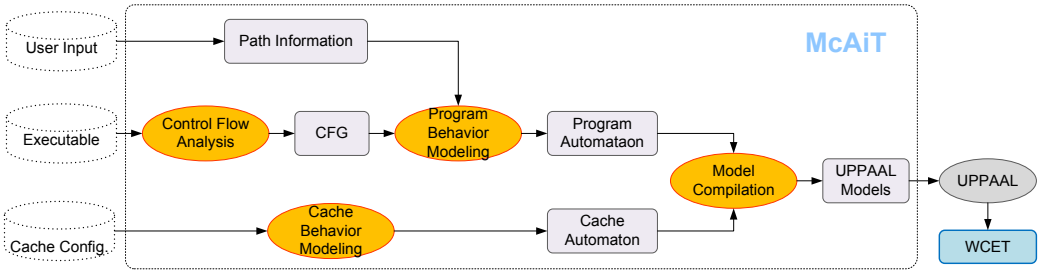


Figure 12 The analysis framework of McAiT.

3.3.3 Model-Checking-Based Methods

Model checking [26] is a powerful technique widely used in system-level timing analysis of real-time systems. Timed automata [6] have been used to model the cache behavior of programs, and a model checker has been employed to find the WCET. Existing work includes the McAiT tool [80], the METAMOC approach [32], and Gustavsson et al.'s analysis [50]. These works use the model checker (UPPAAL in their cases) as a black-box tool. They express the cache access behaviors of a program by the modeling language of the model checker, and verify whether the WCET of the program is bounded by a specified value as a reachability problem.

Figure 12 shows the architecture of the McAiT tool. McAiT first constructs the program automaton out of the CFG, which fully simulates the behavior of the program, such as the control flow and how the program accesses caches. For a given cache configuration, McAiT builds a timed automaton to model each cache. The execution of an instruction causes the program automaton to issue messages to the cache automaton via UPPAAL's channel mechanism, and the cache automaton updates the cache state accordingly. The timed automata models for both the program and the cache are then explored using the UPPAAL model checker to find the WCET.

Essentially, the estimated WCET by a model checker is the actual WCET of the program, since all the possible executions are explored. Cache hits and misses for *each execution of memory accesses* are precisely reported. The major difference between this approach and the CSTG approach is that the possible cache states are not explicitly modeled in the automaton, but rather explored by the model checker. The main drawback of model-checking-based approaches is their lack of scalability, since an exponential state space has to be explored.

3.4 Data Caches

Modern processors are typically equipped with *data caches* to improve the performance of data accesses. Instructions are fetched from known addresses; so instruction fetching can be accurately analyzed. In contrast, data accesses are less predictable [76, 123].

3.4.1 Main Challenges

Before predicting hits/misses for data accesses, the set of data addresses accessed by each instruction needs to be determined, referred to as value analysis or address analysis [9, 129]. The threat to precision is that an imprecise value analysis may not be able to eliminate memory accesses that do not occur in a real execution. The problems are the following: Firstly, data manipulations using redirectable pointers make it hard to statically determine the data items actually accessed. Secondly, in the presence of dynamic data structures on the heap, the data addresses can only be determined at run-time (due to this problem, dynamic data structures are typically avoided in hard real-time systems). Lastly, value analysis may work with abstractions of memory addresses, such

```
1  for (i = 0; i < N; i++)
2    for (k = 0; k < N; k++)
3      for (j = 0; j < N; j++)
4        C[j][i] += A[k][i] * B[j][k];
```

Figure 13 An example of matrix multiplication.

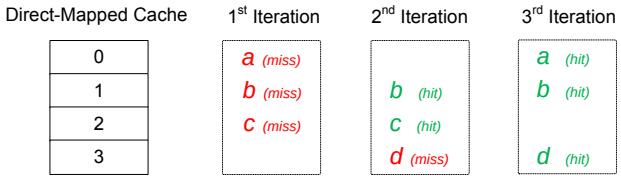


Figure 14 Data cache analysis based on the pigeonhole principle.

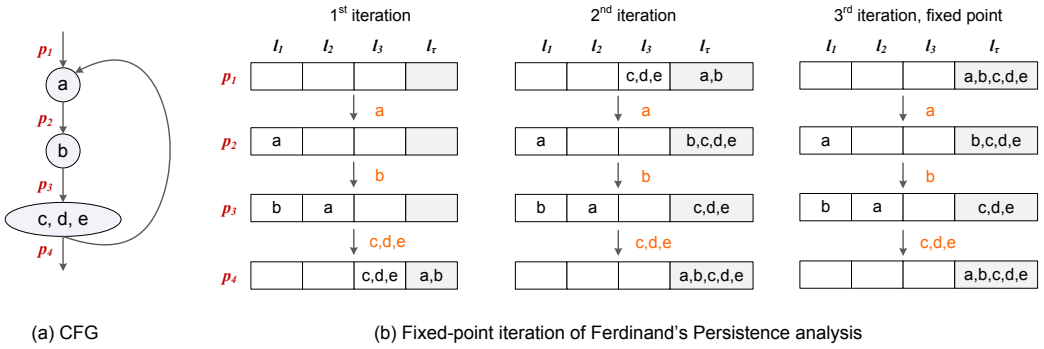
as intervals. As a result, the address range they compute may be overapproximated. Considering non-feasible data accesses in cache behavior analysis increases the probability of not being able to classify memory accesses as cache hits or misses. In addition, a memory access without a precisely determined address pollutes the information contained in an abstract data cache since the update function has to be applied to all potential concrete addresses.

Besides that, data cache analysis is challenged by another problem: executing an instruction may generate accesses to *multiple* data addresses. Figure 13 depicts a program with a matrix multiplication, in which line 4 generates accesses to different matrix elements (in different loop iterations). For this simple program, one can easily determine the data items accessed in each loop iteration. But, in general, data accesses could be very unpredictable due to input dependence of the array indexes. Consider accesses to array $A[x][y]$: if the values of x or y are not clear, one has to conservatively assume that any address in the whole array could be accessed. Furthermore, classifications of memory accesses as used for instruction-cache analysis (AH, AM, FM) may not be sufficient to describe data cache behavior.

3.4.2 Analysis without Input Dependence

Early work, such as the Cache Miss Equation (CME) framework [39], focused on analyzing programs with predictable data accesses. The underlying idea is to set up mathematical formulas (Linear Diophantine equations specifically) to precisely capture both spatial and temporal memory reuses by relating data addresses, loop induction variables and cache parameters. From the solution of the equations, one can check if a memory block is evicted from the cache before it can be reused. An upper bound on the number of misses can thus be obtained for WCET estimation.

However, only a small set of programs can be analyzed by the CME framework: (1) loops must be rectangular loops and perfectly nested; (2) array subscript expressions and the bounds of the loop index must be affine combinations of the enclosing loop indices; (3) no data/input-dependent conditions may exist. The CME framework has been later extended to allow function calls [124], conditionals only depending on the loop induction variables [124], and multiple loop nests [97]. Unfortunately, none of these methods can deal with input dependence. Clauss presented an approach of solving cache miss equations through the mapping to Ehrhart polynomials [27]. Still, the complexity of solving these polynomials is high. Another approach is the Presburger Arithmetic framework [23], which has similar restrictions and is computationally expensive.



(a) CFG

(b) Fixed-point iteration of Ferdinand's Persistence analysis

Figure 15 Ferdinand's Persistence analysis for data caches.

3.4.3 Analysis with Input Dependence

Earlier research to handle input dependence focused on direct-mapped caches. Kim et al. proposed an analysis method based on the pigeonhole principle [65]. Figure 14 shows 3 iterations in the execution of a loop in which a, b, c, and d can be accessed. If in total 9 memory accesses are generated, then at least $9 - 4 = 5$ among them must be cache hits. The 4 cache misses are cold misses. This work was later extended by Staschulat and Ernst to handle programs with unpredictable input dependency [112], in which cache misses are bounded according to data access types: (1) cache misses from predictable accesses are bounded by the pigeonhole principle; (2) cache misses from unpredictable accesses are tracked down by a miss counter and expressed with linear constraints. Unfortunately, these methods are still too restrictive. First, they only work for direct-mapped caches. Second, the loops must fit into the cache to utilize the pigeonhole principle. Essentially, these approaches correspond to simple Persistence analysis for the special case where programs fits into the cache.

AI-based analysis techniques are extended to analyze set-associative data caches with input dependency. Ferdinand extended the Persistence analysis [37] with a new update function to handle multiple memory accesses by one instruction. The basic idea and its drawback can be explained by the example in Figure 15.

Figure 15(a) gives the CFG of a loop in which p_1 to p_4 are program points. Note that each time the instruction after p_3 is executed, one of the blocks from $\{c, d, e\}$ could be accessed. Figure 15(b) shows the fixed-point iteration process, given a cache size of 3. The last column l_τ of each abstract state is used to collect the blocks that have been evicted from the cache (a common structure for most Persistence abstract domains). On the transitions from p_3 to p_4, since it is not clear which of the three blocks (or their combination) is actually accessed, they pessimistically assume that all blocks in $\{c, d, e\}$ could be accessed and cause other blocks to age. Thus, both a and b are evicted from the cache (collected in l_τ). Moreover, since there is no knowledge on the access sequence of c, d and e, they receive an age of 3 when they are brought into the cache state in the 1^{st} iteration. As a result, no cache hit can be predicted for this loop. As only one from c, d, e may be accessed in every iteration, slightly better results would be possible with a different transfer function.

Sen and Srikant developed a Must analysis for data caches [110]. The analysis can be combined with VIVU to discover the persistence property of data accesses. Despite some small differences, the age manipulation in Sen's Must analysis are similar to Ferdinand's Persistence analysis [37], and thus may lead to very pessimistic estimations.

Ferdinand's and Sen's analyses show that without modeling data access patterns, the abstract domain has to do very conservative age maintenance. Again, for the program in Figure 15(a), if by some means we know that the lifetime of c, d, and e do not overlap, then the analysis can be improved. For example, if the loop iterates for 30 times, c is only accessed in iterations 1 to 10, d

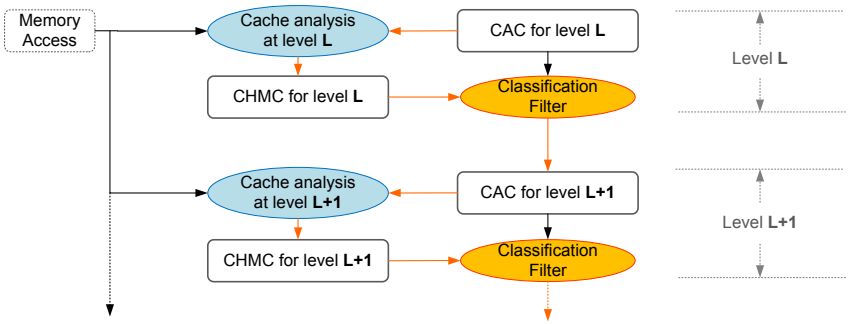


Figure 16 The separate analysis architecture.

only in iterations 11 to 20, and e only in iterations 21 to 30, then c, d and e cannot evict each other in their lifetime. They are actually *persistent* (given a cache size of 3) once they are loaded into the cache, since any of them can only be aged by a and b. Based on this observation, Huynh et al. proposed scope-aware data cache analysis [62]. Each memory access is now associated with a *temporal scope* to model its lifetime, as an augmentation to the traditional AI-based analysis. In the update function, memory accesses that have no overlapping temporal scopes do not cause each other to age. In consequence, some non-existing access conflicts are excluded, and more persistent data accesses can be identified (such as c, d and e).

Hahn and Grund observe that cache analysis does not require knowledge of *absolute addresses* of memory accesses. Instead, it is sufficient to know about the *relation* between the addresses of different memory accesses: do they refer to the same cache block, a different cache block but the same cache set, or different cache blocks in different cache sets? Based on this insight, they developed *relational cache analysis* [51], which can classify accesses as cache hits even if the absolute address of the access is unknown.

To summarize, input dependence makes data-cache analysis a challenge. The main causes are imprecise address analysis and the inability to model and analyze data access patterns. Imprecision of the results may indicate that two memory accesses compete for the same cache set, while in reality they always go into different sets. The success of data cache analysis depends on whether *temporal and spatial locality* of data accesses can be precisely captured and analyzed.

3.5 Multi-Level Caches

Most modern processors adopt a multi-level cache design. Upon a memory access, the processor queries the memory hierarchy from the L1 cache down to main memory until the requested data or instruction is found. Regardless of the number of levels, the highest level cache is generally much faster than main memory, since the latter is accessed via the *off-chip* memory bus. To produce precise WCET estimations, cache analysis should be conducted all the way to the highest level, instead of merely on the L1 cache.

3.5.1 Separate vs. Integrated Approaches

Two major analysis frameworks for multi-level caches are the *separate analysis*, which analyzes caches level by level, and the *integrated analysis*, which deals with the cache hierarchy as a whole.

The separate analysis framework was first proposed by Mueller [87] and refined by Hardy and Puaut [55], who corrected a soundness problem. Figure 16 [55] shows the main work flow. In the analysis of all but the L1 cache, a key information is whether a data request actually leads to an access to this level. For example, if a memory access is predicted *always hit* at L1, then the

Table 2 Computing CAC for level L and memory block r [55].

$CAC_{r,L-1}$ \ $CHMC_{r,L-1}$	AM	AH	FM	NC
A	A	N	U	U
U	U	N	U	U
N	N	N	N	N

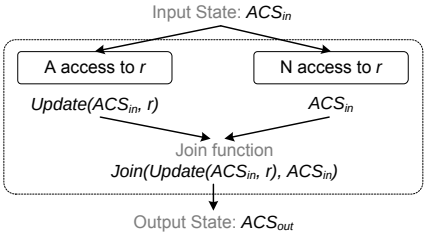


Figure 17 The update function for U access [55].

L2 cache will not be visited. An interface across cache levels, called *Cache Access Classification* (CAC) is introduced to describe this information. The notion $CAC_{r,L}$ denotes the access property of block r to level L, which is evaluated to one of the following three cases:

- N (Never): the access to r is never performed at level L;
- A (Always): the access to r is always performed at level L;
- U (Uncertain): the access to r at level L can neither be excluded nor predicted.

The CAC values for level L are computed from both the Cache Hit/Miss Classification (CHMC) and the CAC for level $L-1$, which is shown in Table 2.

Trivially, if $CAC_{r,L-1} = A$ (or N), then the access to memory block r is always (or never) considered in the analysis of level L. However, handling the U classification requires special attention. In Mueller's multi-level analysis [87], memory accesses with U classification are "conservatively" treated as *always access* in the analysis of the current cache level. However, this treatment is demonstrated to be unsafe [55], since it may underestimate block ages, and thus incorrectly predict cache misses as hits. Hardy and Puaut corrected the problem by considering both possibilities for the U accesses in the update function (shown in Figure 17), which guarantees that the worst-case scenario is never missed.

However, Hardy and Puaut's analysis may suffer from precision problems. Note that in the above CAC computation, the FM classification is treated the same as NC, which means the information obtained by Persistence analysis at level $L-1$ is never leveraged in the analysis of level L. Actually, a block classified as FM at level $L-1$ causes at most one access to level L on its first access. Mueller in an earlier practice tried to solve this problem by unrolling the loop bodies [87], but this approach does not scale.

The component-wise separate analysis has several advantages. First, the analyzer has the flexibility to apply a different analysis method for each cache level, as long as the method produces hit/miss classifications as the interface across adjacent levels. This is desirable for architectures with different replacement policies for different cache levels, such as in the IBM Power 5 processor. Second, the overall analysis is scalable as long as the adopted single-level analysis is scalable.

However, the separate analysis may be pessimistic due to imprecise transfer of cache access information across cache levels. In contrast, *integrated analysis* [111] tries to build a holistic abstract domain for all cache levels, aiming to collect the information lost by the separate analysis.

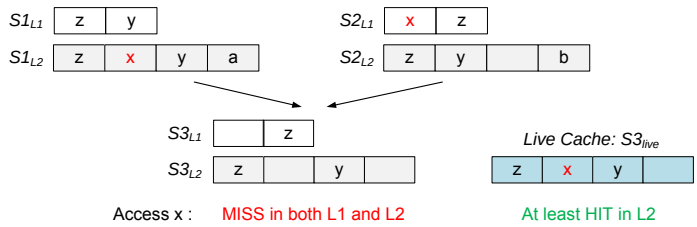


Figure 18 How a live cache helps to obtain a more precise join operation [111].

Consider a Must join operation of a separate analysis with a 2-way L1 cache and a 4-way L2 cache, shown in Figure 18. Si_{Lj} represents the abstract state Si at Level j. Consider block x, which appears in $S1_{L2}$ on the left branch and in $S2_{L1}$ on the right branch. By a separate analysis, x does not appear in the joined state $S3$ of any level. If a subsequent access to x occurs, a cache hit can not be predicted. However, by evaluating both levels together, it can be seen that x does show up in *every incoming path*, so a subsequent access to x should be a cache hit, either in L1 or in L2 depending on the execution history. This is to say, x is guaranteed in the cache hierarchy at the joined program point. Unfortunately, this information is lost in the separate analysis.

Based on this observation, Sondag and Rajan introduced a new component called *live cache* into the traditional abstract domains. At a join, a block is added to the live cache if it appears in some cache level in every input cache state, as depicted by $S3_{live}$[5] in Figure 18. With live-cache information, one can now safely predict that x at least hits in the L2 cache. As reported in [111], the extra overhead by introducing live cache is acceptable for a 2-level cache hierarchy. However, analysis overhead increases with the number of cache levels, since an independent live cache is maintained for every pair of cache levels.

3.5.2 The Impact of Inclusiveness

The relationship between cache levels is a key design feature. In some processors, all data in level L must be contained in level $L+1$. Such caches are called *(strictly) inclusive* caches. For example in Figure 19, the access to e causes a to be evicted from L2, so a is forced to be removed from L1 to guarantee inclusion. Inclusive caches are favored in multi-cores: any data update in the shared L2 cache is automatically synchronized to the private L1 caches of all cores due to inclusion enforcement, which also achieves data coherency. Other processors adopt *exclusive* caches, in which data is guaranteed to be in at most one cache level. Exclusive caches are desirable for resource-limited systems since there is no data duplication in the cache hierarchy. The type adopted in previous discussions is called *mainly-inclusive*, which neither enforces inclusiveness nor exclusiveness. Inclusive caches are harder to analyze than exclusive caches, because the update to one cache level may cause further changes to both lower and higher cache levels. Such behavior may preclude the separate analysis flow.

Hardy et al. adapted the separate analysis [55], originally designed for mainly-inclusive caches, to both inclusive and exclusive caches [56]. The main idea is to first conduct an analysis assuming a mainly-inclusive cache, and to then modify the CHMC results to guarantee that the effects of cross-level cache updates are safely considered. For example in Figure 19, the analysis assuming

[5] x has an older age in $S1_{L2}$ than in $S2_{L1}$. This is because Sondag's analysis assumes the write back policy. If x is evicted from the L1 cache, it is installed in the youngest position of the L2 cache. This means x can still suffer another $k-1$ evictions in L2, where k is the cache associativity. The age of a block in the live cache describes how long it can stay in the *whole* cache hierarchy.

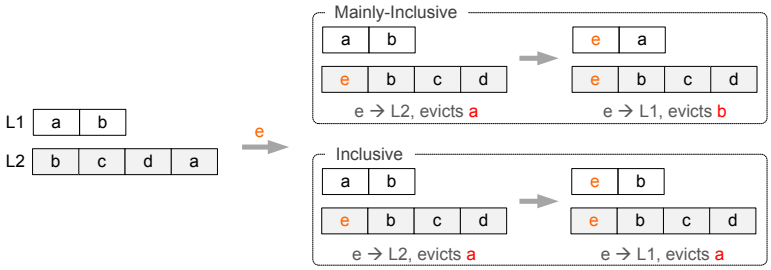


Figure 19 Handling new behavior of inclusive and exclusive caches.

mainly-inclusive cache reports that a is AH at L1 but may be evicted from L2. To adapt this result to an inclusive cache, one must consider the possibility that a can be removed from the L1 cache due to an update in the L2 cache. As a result, a's CHMC at L1 is modified from AH to NC. Similar problems exist in the May analysis as well. As a consequence to CAC computation, accesses to all cache levels except L1 are changed to Uncertain ($CAC_{r,L} = U$ for $L \geq 2$). All these modifications severely degrade the analysis precision.

Sondag and Rajan extended their integrated analysis to both inclusive and exclusive caches [111]. The resulting update and join functions for inclusive caches are very complex since once a block is accessed on some level L, the corresponding changes in other cache levels must be correctly considered. Analysis of exclusive caches has similar problems. To summarize, the inter-dependent updates among cache levels to enforce inclusion/exclusion brings new difficulties regardless of the analysis framework.

3.5.3 The Impact of Write Operations

A write to the cache occurs when a data variable receives a new value. Two levels of policies determine when and where to conduct the writing of data back to memory. The *write-through* policy requires that the new value is updated synchronously both in the cache and in main memory. In contrast, the *write-back* policy only marks the modified data as *dirty*, and performs the actual update of memory only when the data is evicted from the cache. A *write miss* occurs if the data to write are not in the cache. Under the *write allocate* policy, missed data are first loaded into the cache, and then updated with the new value, resulting in a cache miss followed by a cache hit. For the *non-write allocate* policy, the data are directly written to main memory, bypassing the caches.

The write-through policy is generally easy to handle in cache analysis, since data writes to a certain cache level incur no change to other cache levels. However, for the write-back policy, evicted dirty data are written to higher cache levels. Second, for the write allocate policy, a write operation always causes cache accesses regardless of hit or miss, which makes no difference from the read operation. However, for non-write allocate caches, a write miss never causes a cache access, so one cannot simply assume that each write operation changes the cache state, as is the case for reads. Like the inclusiveness enforcement, complex write operations cause cross-level cache updates, which is a challenge to the analysis.

Hardy's separate analysis framework has been extended to multi-level data caches by Lesage et al. [67] and to multi-level unified caches by Chattopadhyay and Roychoudhury [24]. However, both analyses assume a write-through policy. The abstract domains adopted in these methods are not able to handle write-back. Sondag and Rajan modeled write-back behavior in their integrated analysis [111]. It is shown that modeling write-back is easier by an integrated abstract domain, but one still has to carefully distinguish *possibly evicted blocks* from *definitely evicted blocks* at any level during the analysis to guarantee soundness. Definitely evicted blocks are identified from the

May information in Sondag's method. Due to the access uncertainty of possibly evicted blocks, conservative update and join operations have to be employed, causing a loss in precision.

4 Analysis of Non-LRU Caches

In the past two decades, most research on cache analysis in the real-time domain was focused on the LRU replacement policy. The analysis of non-LRU replacement policies, i.e., those widely adopted in real-life processors, is still immature. In this section, we look into the challenges for non-LRU analysis and survey existing techniques.

4.1 Why Are Non-LRU Replacement Policies Hard to Analyze

To answer this question, we explore why it is hard to design *precise* and *efficient* abstract domains and the corresponding operations for non-LRU caches. We identify multiple challenges discussed in the following paragraphs.

4.1.1 Unsuitability of AH, AM, and FM Classifications

Under LRU replacement most memory accesses can be classified as AH, AM, or FM. Are these classifications equally suitable for non-LRU replacements? If not, what are alternatives that are better suited to characterize other policies' behavior?

Unfortunately, these classifications are not as suitable for non-LRU replacements as they are for LRU. As shown in Guan et al.'s analysis [47], under FIFO replacement memory accesses may exhibit alternative hit and miss behavior so that none of the traditional classifications (AH, AM or FM) applies. Similarly, under MRU many cache accesses exhibit the *K-Miss* property [45]. An access classified as *K-Miss* suffers several misses (bounded by $K \leq$ *cache associativity*) upon the first few accesses, and then persists in the cache. This kind of persistence property, however, is not captured by the FM classification. This demonstrates that one needs to better understand the specific cache behavior under different policies to come up with proper classifications.

4.1.2 Irregular and Non-Monotone Cache Update Behavior

The abstract domains and the corresponding transfer functions are designed to compute cache behavior invariants. An abstract domain is precise and efficient if (1) abstract states can compactly represent many concrete states, while preserving the information required for classification, and at the same time (2) transfer functions precisely capture the effect of a memory access on the concrete cache states. For example in the AI-based analyses for LRU, the block age bounds in the abstract states capture precisely the information required to classify blocks as cached or not. Further, this information can be precisely maintained by the transfer functions due to LRU's regular cache update: a) whether or not a block's age depends solely on its age relative to the accessed block's age. Upper and lower bounds (in Must and May analyses) on the ages of blocks can be precisely updated due to the monotonicity of the operation, b) regardless of its previous age, and whether it was cached or not, the accessed block is always assigned the youngest age.

Unfortunately, most non-LRU replacement policies do not possess such monotone behavior. Take the FIFO replacement in Figure 4(b) for example. After a hit to d, d remains in the original position and is immediately evicted by the next access to x. The fact that d is *recently* accessed is not reflected by the update rules. An example of irregular behavior under MRU is shown in Figure 4(c). After f is installed into the cache, a subsequent hit to c followed by a miss to e evicts f out of the cache. However, block b, which is older than f, remains in the cache even after f is

evicted. The problem of PLRU is shown in Figure 4(d). In the state before a is accessed, the oldest block that will be evicted next is b. However, after a hit to a, the block to be evicted is changed to d.

To build efficient abstract domains for such replacement policies is very difficult. For example, in a Must analysis for FIFO [41], early determination of cache misses is helpful to better predict cache hits later. However, a very complex May analysis has to be designed to determine miss information as early as possible. For PLRU, a precise analysis must model the tree bits. The Must analysis for PLRU [40] by Grund employs a far more complex abstract domain, compared to LRU, to express information in the tree and predict cache hits.

4.1.3 The Influence of Initial States

Cache behavior heavily depends on execution history. In [104], it is shown that program performance under non-LRU replacement policies are very sensitive to the initial cache state, i.e., what remains in the cache before a program starts. This presents a challenge to obtain precise estimations. To illustrate the problem, assume currently m is accessed in a FIFO cache and we want to precisely estimate m's lifetime. There are many possible situations: Case 1: m hits, but it has been in the cache for the longest time among the cached blocks, and thus it will be evicted upon the next cache miss. Case 2: the access to m is a miss and due to first-in, first-out behavior m will withstand another $k-1$ (where k is cache associativity) cache misses without being evicted. Obviously, m's remaining "life expectancy" in the two cases is rather different. If no knowledge on the initial cache states is available, a safe analysis (to predict hits) has to assume the worst case, i.e., *Case 1*. To be more precise, one may try to distinguish *Case 2* from *Case 1* by investigating whether the current access to m is a miss or not. Then, the analysis needs to know there are enough cache misses to evict any previously accessed m out of the cache, which again relies on the initial states. If a replacement policy can remove uncertainty from the initial states quickly, it will be easier to analyze.

To analytically model the effects of unknown initial states, Reineke et al. proposed a metric, *evict*, for a replacement policy [105], as listed in Table 3[6]. Intuitively, the value of $evict(k)$ tells us after how long a sequence of pairwise different memory accesses, we can conclude that the cache only contains blocks from the access sequence, or, in other words, how long a sequence of pairwise different accesses is needed to evict all unknown cache contents from the cache.

The $evict(k)$ results show that generally longer sequences have to be observed for non-LRU replacement policies. This property directly corresponds to the achievable precision by a May analysis to predict misses, and indirectly affects Must analysis, the precision of which partly depends on how much May information can be obtained during the Must analysis [41]. Similarly, the $fill$ metric captures the number of pairwise different memory accesses required to reach a single cache state independently of the initial cache state. As can be seen in Table 3, the gap between LRU and other policies is even bigger for the $fill$ metric.

The two metrics *evict* and $fill$ discussed above relate to the precision of *classifying* analyses. In other work, Reineke and Grund [104] determined how strongly the *number* of cache misses may vary depending on the initial state, which is related to the precision of *bounding* analysis in the presence of uncertainty about the initial state. Their analysis demonstrates that the number of cache misses may vary strongly depending on the initial state under FIFO, PLRU, and MRU, while it may not vary much under LRU replacement. Further, it is shown that the empty cache

[6] Table 3 extracts the HM case for *evict* and $fill$ with $k > 2$ from the full results in [105], where k is cache associativity.

Table 3 Predictability metrics [105].

Policy	$evict(k)$	$fill(k)$
LRU	k	k
FIFO	$2k-1$	$3k-1$
MRU	$2k-2$	$3k-4$
PLRU	$\frac{k}{2}log_2k+1$	$\frac{k}{2}log_2k+k-1$

Table 4 Generalized predictability metrics [105].

Policy	$mls(k)=mls'(k)$	$evict'(k)$
LRU	k	k
FIFO	1	$2k-1$
MRU	2	$2k-2$
PLRU	$\log_2 k+1$	∞

state is not necessarily the worst initial state for non-LRU policies. This presents severe problems for measurement-based WCET analysis approaches.

4.2 Predicting Cache Hits

The more hits can be predicted, the better the WCET bound. Must and Persistence analyses discussed earlier are used for this purpose. To predict a hit for a memory access to m, an analysis needs to ensure m has not been evicted since its last access. Intuitively, the more pairwise different blocks have been accessed since the last access to m, the higher the chance that m has been evicted from the cache. To capture this information, we introduce the following definition:

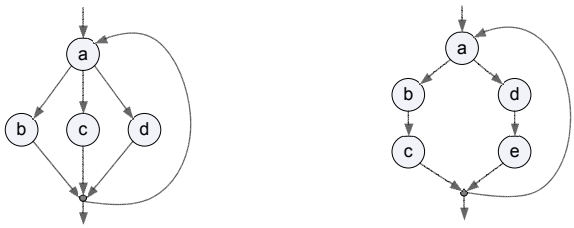
▶ **Definition 1** (Reuse Distance)**.** Let p be a memory access sequence that ends with an access to memory block m. The *reuse distance*[7] of m, denoted by $\mathbf{rd}_p(m)$, is the number of distinct blocks accessed along p since the previous access to m in p.

The notion of reuse distance coincides with the ages of memory blocks that are used in the analysis of LRU caches. For example, let $p_1 = \langle beabcda \rangle$ and $p_2 = \langle abccdcccba \rangle$, we have $\mathbf{rd}_{p_1}(a) = \mathbf{rd}_{p_2}(a) = 4$. Due to branches or input-dependent memory accesses, there can be multiple access sequences leading to the same access to block m in a program. So we define the *maximal reuse distance*, denoted by $\widehat{\mathbf{rd}}(m)$, as the maximal value of $\mathbf{rd}(m)$ over all possible memory access sequences leading to a particular access. We can evaluate analysis techniques by the maximal reuse distances for which they can predict cache hits.

Reineke et al. explored the *minimal life-span* [105] for different replacement policies, which is the minimal length of a sequence of pairwise different memory accesses necessary to evict a block that has just been accessed from the cache. The minimal life-span values are given in the $mls(k)$ column of Table 4. A slight variation of $mls(k)$ is $mls'(k)$, which considers the minimal number of pairwise different memory blocks required to evict a block that has just been accessed from the cache. Notice, the slight difference between the two notions: $mls(k)$ considers only sequences consisting of pairwise different accesses, whereas $mls'(k)$ allows multiple accesses to the

[7] In the literature, this is also referred to as the *(LRU) stack distance* [86]. Also, note that the term reuse distance is used ambiguously in the literature, sometimes referring to the number of accesses since the previous access to m, and sometimes referring to the number of distinct blocks accessed since the previous access to m. We follow the latter notion.

Figure 20 Levels to explore cache hits.



(a) Program 1 (b) Program 2

Figure 21 Example programs.

same block. For all the considered policies, $mls(k)$ is equal to $mls'(k)$. The metric $evict'$, also listed in Table 4, will be discussed in Sec. 4.3. The $mls'(k)$ metric tells us how many of the most recently accessed blocks are guaranteed to be in the cache. By this result, a Must analysis can be constructed as follows: for any memory access to m, one can check if $\widehat{\mathbf{rd}}(m) \leq mls'(k)$ holds for the given replacement policy. If yes, the memory access to m is guaranteed to be a hit. We say that such an analysis explores *Level I*, which is illustrated in Figure 20.

Notice that to predict cache hits, the Must analysis for LRU presented in Sec. 3.1 computes upper bounds on the maximal reuse distances of memory blocks. Similarly, the May analysis computes lower bounds on the minimal reuse distance of memory blocks to predict cache misses. As the notion of reuse distances is replacement policy-independent, these LRU Must analysis can thus safely be reused to predict hits for other policies, by relying on the policies value of $mls'(k)$.

However, the $mls'(k)$ values for non-LRU replacements are commonly small compared to cache associativity k, because they consider worst-case scenarios. In practice, it is unlikely that the worst case occurs at every program point. Thus, analyses tailored to a particular replacement policy can often go beyond *Level 1* in predicting hits.

For a program that can fit into the cache of size k, there is a strong intuition that each block of the program eventually persists in the cache, i.e., after some misses, the remaining accesses to each block are definitely cache hits. For such programs, the maximal reuse distance of any access is no larger than k (*Level II* in Figure 20). This property is attractive as it enables an efficient Persistence analysis that simply collects the set of different blocks accessed by a program. However, such a Persistence analysis is not correct for every replacement policy: it works for LRU, MRU and FIFO, but not for PLRU.

Figure 22 illustrates the cache state transitions when the loop in Figure 21(a) is executed, alternating between the three branches in the loop body on a 4-way PLRU cache. Each time a is accessed, the root bit points to the right subtree, so b, c, and d have to compete for the two cache lines on the right. Even though the loop can fit into the 4-way cache set, only block a is persistent. This example unveils a negative property of PLRU: it does not always make use of all its capacity [14].

It is crucial to investigate what process a block may go through before it finally persists in the cache. This is important to safely bound the number of misses that may occur.

For MRU, Guan et al.'s results [45] show that if for a block m, $\widehat{\mathbf{rd}}(m) \leq k$ holds, m will eventually persist in the cache. However, m may suffer more than one miss before reaching the

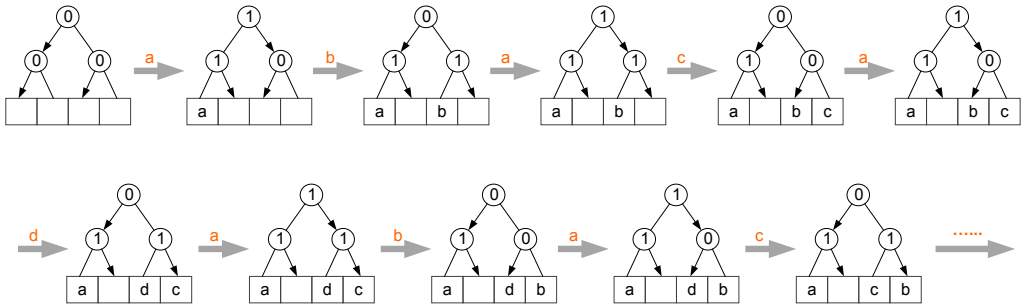


Figure 22 An example that demonstrates that PLRU does not always use the cache's entire capacity.

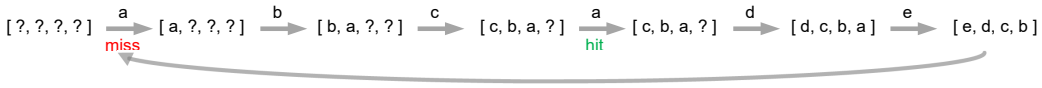


Figure 23 Alternative hit and miss behavior on FIFO.

stable state. The result is strong in that bounds on the number of misses are determined for all reuse distances in *Level II* for MRU. Consider the program in Figure 21(b): even if the program cannot fit into a 4-way MRU cache, block a's number of misses can be bounded by a constant since $\widehat{\mathbf{rd}}(a) \leq 4$.

For FIFO replacement, Grund and Reineke [42] show that if a loop entirely fits into the cache, each block suffers at most one miss and then persists in the cache[8]; otherwise, no guarantee is given. Guan et al. further explored this problem, and found that if a block m satisfies $\widehat{\mathbf{rd}}(m) \leq k$, then even though m is not eventually persistent, it is still guaranteed to enjoy cache hits, which can be expressed by a bound on the number of misses that accesses to m may suffer [47]. For example in Figure 21(b), $\widehat{\mathbf{rd}}(a) \leq 4$ holds for a 4-way FIFO cache. In the worst case, the loop alternatively takes the two branches, and a may suffer cache misses repeatedly. To evict a from the cache, both branches have to be taken, which causes a to enjoy a cache hit in the execution of one of the branches (shown in Figure 23). It can be shown that a suffers at most $\lfloor \frac{1}{2} \cdot x \rfloor + y$ misses, where x is the execution count of a, and y is the total number of times the loop is entered.

The only analysis to predict hits for PLRU for maximal reuse distances in *Level II* is a Must analysis proposed in [43]. The analysis presented is based on the following observation: to evict a block with the fewest possible accesses to distinct memory blocks, the three bits (assuming an associativity of 8) that are on the path from a cache line to the root of the tree need to be flipped in a particular order, namely from the bottom to the top. This is illustrated in Figure 24 for block m. Notice that flipping bits near the root before flipping all bits closer to the leaves does not contribute to evicting m, as these bits will eventually be flipped back before evicting m. The basic idea behind the analysis in [43] is to track two properties: a) the number of bits that already point towards a block (counting from the leaf of the tree), and b) the so-called "sub-tree distance" between pairs of blocks. The sub-tree distance between a and b captures which bits on the path from a to the root an access to b may flip. By analyzing these key properties, it is sometimes possible to predict that a block stays in the cache even if more than $mls'_{PLRU}(k) = \log_2 k + 1$ other blocks have been accessed.

[8] Note that blocks do not necessarily encounter their misses in the first loop iteration. It may take several iterations for all the blocks to stabilize in the cache.

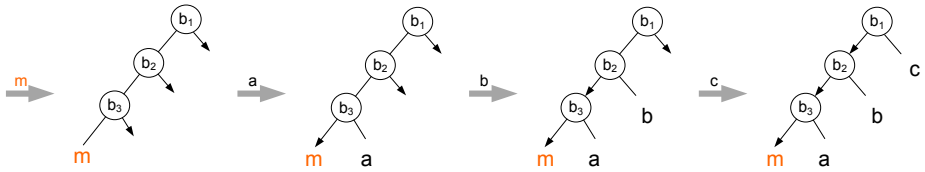


Figure 24 A scenario in which block m is evicted with the minimal $mls'(k) = \log_2 k + 1$ accesses.

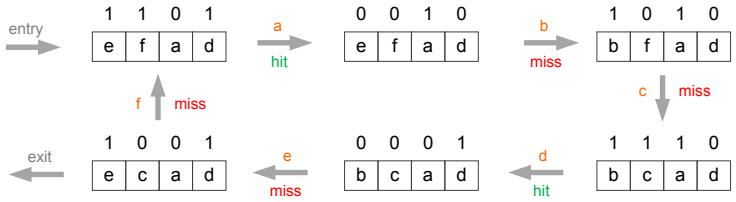


Figure 25 Cache behavior under MRU in *Level III*.

To go beyond *Level II* means to explore whether blocks with maximal reuse distance larger than k still have cache hits. One needs to first show that the above fact does occur for some replacement policy, and second propose an analysis to discover the cache hits. Take MRU for example, Figure 25 shows that even if we have $\widehat{\mathbf{rd}}(a) = \widehat{\mathbf{rd}}(d) > 4$ for a 4-way cache, accessing a and d is always hit. But this phenomenon relies on the initial state at the entry of the loop. To explore such behavior, the abstract domain must be able to preserve very detailed information on cache states. This requirement makes it very hard to explore cache hits in *Level III* by abstract analysis methods. The abstractions introduced by Grund and Reineke for FIFO [41] are in principle able to predict *Level III* hits, however, they usually require a highly context-sensitive analysis to do so. *Level III* ends at $evict'(k)$, after which no more hits are possible.

4.3 Predicting Cache Misses

Predicting more misses tightens the estimated BCET, but it can also indirectly help with the analyses for some non-LRU replacement policies to predict more hits [41]. A concrete example is the case of FIFO explained in the discussion of the influence of initial states in Sec. 4.1. Furthermore, for multi-level cache analysis, predicting more misses for level L reduces the uncertainty of cache accesses on level $L+1$, which leads to more precise overall estimations. Lastly, in micro-architectures with timing anomalies, if a memory access cannot be classified as a cache hit, both the cache hit and the cache miss cases need to be explored. Predicting cache misses may in such cases drastically reduce analysis times, as it allows to explore only the cache miss case.

To predict cache misses requires to show that memory blocks are not in the cache right before they are being accessed. Thus a May analysis, i.e., an analysis that overapproximates cache contents, is required to safely predict cache misses. May analyses can be constructed based on a variation of the *evict* metric [105], which we denote by $evict'$. The difference between *evict* and $evict'$ is the same as the difference between *mls* and mls': any sequence s containing $evict'(k)$ distinct memory blocks is guaranteed to evict any prior cache contents not contained in the sequence s. In contrast, *evict* refers only to sequences that never access the same memory block twice. Values of $evict'$ for common policies are listed in Table 4. For example, for FIFO, $evict'(k) = 2k-1$. This means that after accesses to $2k-1$ pairwise different blocks, the cache only contains elements from the accessed sequence. Then, the May analysis only needs to observe a sequence with $2k-1$ pairwise different blocks other than m precluding the current access to m. On the other hand, for PLRU, $evict'(k) = \infty$. In other words, there are sequences of memory

accesses containing an arbitrary number of distinct memory blocks that do not evict all prior cache contents. We have seen an example of such a sequence in the previous section, which is illustrated in Figure 22.

May analyses based on $evict'$ can be constructed by determining *lower bounds* on the reuse distances of memory blocks. The LRU May analysis presented in Sec. 3.1 does exactly that.

The *evict* metric suggests that less than $evict(k)$ accesses to pairwise different memory blocks do not allow to predict any misses, thereby constituting a limit on how much information a May analysis can obtain. However, this conclusion is built on the assumptions that the initial state is *completely unknown*, and that the access sequence consists of *pairwise different* memory accesses. There is thus hope that by tailoring an abstract domain to a specific replacement policy, more precise May analyses can be achieved. So far, such abstractions have only been built for the FIFO replacement policy [40, 41]. Due to limited space, we only explain the main intuitions and the key constituents of the two existing FIFO abstract domains.

Consider a 4-way FIFO cache and the access sequence $x \circ s \circ x$, where the first access to x is a miss and installs x into the "first-in" cache way, and then a sequence s that does not contain x is accessed followed by another access to x. To predict a miss for the second access to x, it suffices to check whether either of the following two properties holds:

- *Property 1*: The accesses in s result in at least 4 misses;
- *Property 2*: Before the second access to x, every cache way is occupied by a memory block from s.

The abstract domain proposed in [41], which we call $FIFO^{\alpha}$, checks *Property 1*. The key information to be maintained in the abstract domain is the number of *definite misses* after the first access to x, denoted by $dm(x)$. When $dm(x)$ reaches 4 during the analysis, one can predict misses for a future access to x. To better maintain definite misses, the number of *cache ways covered* by blocks accessed after the last access to x is maintained as auxiliary information.

A disadvantage of $FIFO^{\alpha}$ is that it only starts to predict misses after $2k-1$ pairwise different memory blocks have been accessed, which is in line with the *evict* metric. Therefore, Grund and Reineke proposed another FIFO domain, which we denote by $FIFO^{\beta}$, so that cache misses can be predicted even if fewer than $2k-1$ pairwise different blocks are accessed between two different accesses to the same block [40, 42]. The domain $FIFO^{\beta}$ checks *Property 2* to predict cache misses. For the above example, it explores if memory blocks accessed in sequence s eventually cover all the 4 cache ways. The exploration is based on a more powerful result (Lemma 4 in [42]):

If a sequence s contains l distinct blocks, then $l-k+1$ cache ways must be occupied by the contents of s, regardless of the initial cache state.

Importantly, the effect of consecutive sequences adds up. For example, let $s = s_1 \circ s_2 = \langle a,b,c,d,e\rangle \circ \langle a,b,c,d,e\rangle$. The accesses to s_1 cover the $5-4+1=2$ most-recently-used ways in the cache set. Similarly, the accesses to s_2 contribute another $5-4+1=2$ to the covered positions. Then we can guarantee that access to s finally covered all the 4 cache ways[9]. This means the execution of s actually evicts x out of the cache and a miss on the second access to x can safely be predicted.

So far, no May analysis is known for PLRU. For MRU the best known May analysis is based on $evict'$. Precisely predicting misses for these two policies is still a challenge.

[9] The effectiveness of this analysis depends on how a long sequence is partitioned. Grund has a systematic method to explore different partitionings for optimization [40].

4.4 The Relative Competitiveness Framework

Besides the above research, Reineke and Grund proposed the Relative Competitiveness framework [103] that allows to translate analysis results for one replacement policy to another policy. The promise is then to apply known LRU analyses to non-LRU caches.

A policy P is (k, c)-*hit-competitive relative* to policy Q if the number of cache hits $h_P(s)$ of P on sequence s is bounded from *below* by the number of cache hits $h_Q(s)$ of Q as follows: $h_P(s) \geq k \cdot h_Q(s) - c$. Similarly, a policy P is (k, c)-*miss-competitive relative* relative to policy Q if the number of cache misses $m_P(s)$ of P on sequence s is bounded from *above* by the number of cache misses $m_Q(s)$ of Q as follows: $m_P(s) \leq k \cdot m_Q(s) + c$.

By monotonicity of the two inequalities, they can also be applied to lower bounds on the number of hits and upper bounds on the number of misses: For example, given a lower bound on the number of hits of Q using hit-competitiveness a lower bound on the number of hits of P can be derived.

For $(k, c) = (1, 0)$ the notions of (k, c)-hit- and miss-competitiveness coincide. In this case, P "dominates" Q. In other words, P never incurs more misses than Q. In such a case we simply say that P is $(1, 0)$-competitive relative to Q. Then, a Must analysis for Q is a valid Must analysis for P; conversely, a May Analysis for Q is a valid May Analysis for P.

In [103] it is shown how to automatically compute the best values for (k, c) such that policy P is (k, c)-hit/miss-competitive relative to policy Q, for fixed associativities of the two policies. Depending on the similarity of P and Q this computation scales to associativities between 8 and 256.

The most interesting cases are those in which either P or Q is LRU, as precise analyses for LRU are known. Examples for hit-competitiveness results derived in this way are [101, 103]:

- An 8-way FIFO cache is $(\frac{1}{2}, \frac{7}{2})$-hit-competitive relative to an 8-way LRU cache.
- An 8-way FIFO cache is $(\frac{2}{3}, 2)$-hit-competitive relative to an 4-way LRU cache.
- An 8-way PLRU cache is $(\frac{1}{2}, \frac{3}{2})$-hit-competitive relative to an 6-way LRU cache.
- An 8-way MRU cache is $(\frac{2}{3}, \frac{4}{3})$-hit-competitive relative to a 4-way LRU cache.

To use this relation, assume 100 hits are predicted for a program on an 4-way LRU cache, then $\lceil \frac{2}{3} \times 100 - 2 \rceil = 65$ hits are guaranteed on an 8-way FIFO cache.

The metrics $mls'(k)$ and $evict'(k)$ are strongly related to $(1, 0)$-competitiveness relative to LRU. In particular, let $mls'_P(k)$ and $evict'_P(k)$ denote the values of the two metrics under policy P. Then, P is $(1, 0)$-competitive relative to $LRU(mls'_P(k))$ and $LRU(evict'_P(k))$ is $(1, 0)$-competitive relative to P. For example:

- $LRU(2k - 1)$ is $(1, 0)$-competitive relative to $FIFO(k)$.
- $LRU(2k - 2)$ is $(1, 0)$-competitive relative to $MRU(k)$.
- $PLRU(k)$ is $(1, 0)$-competitive relative to $LRU(\log_2 k + 1)$.

Cache analyses based on relative competitiveness can be pessimistic, because the relation holds for *any* possible workload. Moreover, the framework provides bounds on hits (or misses) for the whole program or alternatively program fragments rather than classifying independent memory access, except for the case of $(1, 0)$-competitiveness. This makes it difficult to apply the approach in multi-level cache analysis, or in integrated analyses considering both caches and pipelines.

5 Execution Environments

Discussions so far have focussed on analyzing an independent program. Cache analysis is severely challenged in the presence of complex execution environment, such as multi-tasking systems or shared-cache multi-cores, where extra time delay due to interference on caches from other co-scheduled/running programs must be taken into account.

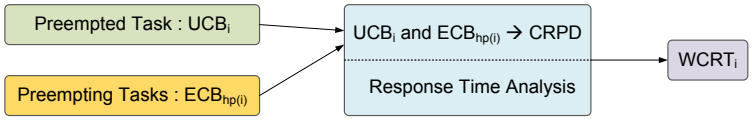


Figure 26 The separate CRPD analysis framework.

5.1 Cache-Related Preemption Delay

An essential feature of real-time systems is preemption, which allows a higher priority task to preempt a lower priority task so that the higher priority one meets its deadline. However, preemptions may lead to extra cache misses: the execution of the preempting task may alter the cache state, so that once resumed, the preempted task needs to bring data back into the cache that was evicted as a consequence of the preemption. The extra delay due to cache reloading is commonly referred to as the *Cache-Related Preemption Delay* (CRPD). Empirical results [74] show that CRPD contributes significantly to the execution time, so it must be precisely estimated to obtain tight estimations of response times. Furthermore, it has also been shown that with CRPD, the synchronous release of all higher priority tasks does not represent the critical instance of single-core preemptive scheduling [134]. Clearly, preemptions introduce a new dimension of complexity into timing analysis.

The most intensively studied framework is *separate* CRPD analysis, in which the CRPD is treated as a separate overhead rather than as a part of the WCET of the preempted task. To bound the CRPD under LRU replacement, two approaches have been proposed, which are illustrated in Figure 26:

1. By analyzing the preempted task [2, 66, 91, 119, 113]: Additional misses can only occur for *useful cache blocks* (UCBs), i.e., blocks that may be cached and that may be reused later, resulting in cache hits. For LRU, the number of such UCBs is a bound on the number of additional misses due to preemptions. Static analyses have been proposed to safely approximate the set of UCBs.
2. By analyzing the preempting task [91, 113, 119, 121]: The preempting task may only cause additional cache misses in those cache sets that it modifies. Thus, analyses to compute bounds on the number of *evicting cache blocks* (ECBs) have been developed. A memory block is an ECB if it may be accessed during the preempting task's execution. However, for set-associative caches, approaches based purely on ECBs have so far been either imprecise [17] or unsound [119], as shown in [17].

The CRPD is computed as the total time delay of all preemption-related cache misses. The final step is to take into account the computed CRPD bounds in a schedulability analysis framework, so that the Worst-Case Response Time (WCRT) of the preempted task can be obtained. All such approaches assume *timing compositionality* [52] so that the cost of additional cache misses can be accounted for separately.

5.1.1 Computing Useful and Evicting Cache Blocks

Since a preemption must happen before some instruction, here we first consider what happens at a particular program point. Most existing work adopts the UCB definition in [66]. A cache block m is useful at a given program point p, if:

1. m may be cached at p;
2. m may be reused at some program point reachable from p without being evicted along the corresponding path.

To determine memory blocks that satisfy Condition (1), one needs to collect the set of blocks that may be cached by any possible program path from the starting point of the CFG to p, referred to as Reaching Cache Blocks (RCBs) and denoted by RCB_p. This corresponds to a May analysis as discussed in Sec. 3.1. To determine memory blocks that satisfy Condition (2), a set of Live Cache Blocks (LCBs), denoted by LCB_p, is computed similarly to RCB_p, however, by a *backward analysis*. Then, an overapproximation of the set of useful cache blocks at point p, UCB_p is obtained by the intersection of RCB_p and LCB_p. A bound on the CRPD is then obtained by taking the maximum size of UCB_p over all program points.

For direct-mapped caches, two major techniques exist: set-based analysis [66] and state-based analysis [91]. Both techniques rely on dataflow analyses to collect the RCBs and LCBs at each program point. State-based analysis maintains all possible concrete cache states at a program point. The analysis is precise, but does not scale to large programs. In contrast, set-based analysis maintains one abstract state at each program point, which collects the set of all possible cached blocks for each cache line. Staschulat and Ernst [113] proposed a scalable precision analysis that presents a trade-off between the above two analyses. The main idea is to pose a bound on the number of cache states maintained at each program point. Whenever the number of states goes beyond the limit, cache states are merged.

Regarding the analysis of the preempting task, note that (a) what matters is the size of the set of evicting cache blocks and not its actual contents, and (b) sizes that exceed the associativity of the cache do not have to be distinguished, as they will evict all prior cache contents anyway. For those reasons bounds on the number of ECBs can be obtained from bounds on the number of reaching cache blocks at the end of program execution, i.e., RCB_{end}.

5.1.2 CRPD Computation for Direct-Mapped Caches

For direct-mapped caches, the CRPD can be estimated by only considering the preempted task, which pessimistically assumes that each UCB of the preempted task could be evicted by the preempting task [66]. These techniques are classified as the UCB-Only approach by [4]. The CRPD can also be computed by only considering the preempting tasks [20, 121], which assumes any ECB of a preempting task may cause a preemption related cache miss (ECB-Only by [4]). Clearly, more precise CRPD can be computed by evaluating both the preempting and the preempted tasks. Specifically, the ECB-Only approaches have been improved by considering the preempted tasks, resulting in the UCB-Union class [118]; similarly, the UCB-Only approaches have been extended into the ECB-Union class [4].

Schedulability analysis needs to take the CRPD into account. Consider a widely adopted schedulability analysis [7] shown in Equation (5), where R_i is the response time, C_i is the WCET of a task, and T_j is the activation period. Equation (5) can be interpreted in the following way: the preemption cost of task τ_i preempted by τ_j, denoted by $\gamma_{i,j}$, is seen as an *extra* part of the execution time of the *preempting task* τ_j.

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + \gamma_{i,j}) \,. \tag{5}$$

In the presence of nested preemptions, as shown in Figure 27, the response time of τ_3 includes both the indirect cost of τ_1 preempting τ_2 (γ_a) and the direct cost of τ_2 preempting τ_3 (γ_b). The main problem is how to safely account for γ_a. Actually, γ_a can be considered in $\gamma_{3,1}$. Note that γ_a may be larger than γ_b, so a safe $\gamma_{3,1}$ needs to account for the maximal cost of τ_1 preempting any lower priority task, however not lower than τ_3. Note that the ECB-Only approaches do not suffer from such nested preemption problems since they do not consider the preempted task.

A disadvantage of analyses by Equation (5) is: the worst-case delay $\gamma_{i,j}$ is always assumed for each preemption (τ_j preempting τ_i). As a result, some cache evictions can be included multiple

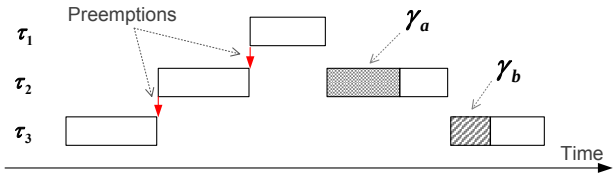


Figure 27 An example of nested preemptions.

Figure 28 An example of reordered misses.

times. To reduce this pessimism, other approaches [4, 114] adopted the schedulability test of Equation (6) instead, which evaluates the total cost of multiple preemptions of τ_j preempting τ_i as a whole. The computation of $\gamma_{i,j}^{sta}$ differentiates preemption scenarios, and thus can avoid unnecessary inclusion of cache evictions.

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j + \gamma_{i,j}^{sta} \right). \tag{6}$$

Altmeyer et al. provided a detailed classification of different approaches to bound the CRPD for direct-mapped caches and their relationship [4].

5.1.3 CRPD Computation for Set-Associative Caches

The computation of UCBs and ECBs can be solved by existing May analyses for set-associative caches. The main challenge is how to precisely and safely compute the "intersection" between the sets of UCBs and ECBs.

Let us first discuss a safety problem, given LRU replacement. Consider the case in Figure 28. Blocks a, b, c and d are all useful blocks. A preemption installs x into the cache set and thereby evicts a. The subsequent accesses of the preempted task to a, b, c and d are all cache misses even though the preempting task only evicted one cache block. This illustrates that there are two types of context-switch misses [17, 74]. The miss to a is a *replaced miss*, as a direct result of the preemption. In contrast, the misses to b, c and d are an indirect result of *reordering* of blocks by the LRU replacement policy. This example shows that even a single ECB can lead to a chain of misses to multiple UCBs, which cannot happen for direct-mapped caches. An example of an unsafe analysis is [118], which overlooked reordered misses.

One way to cope with this problem was proposed in [17]. As soon as there is a single ECB that maps to a particular cache set, all the UCBs that map to the same cache set are assumed to contribute to context-switch misses. This is obviously conservative, and it can be improved by obtaining more detailed information about the useful cache blocks. This information is captured by the notion of *resilience* introduced by [5].

The resilience $res(m)$ of a useful cache block m is the amount of "disturbance", i.e. its ECBs, by a preempting task that the block may endure before becoming useless to the preempted task. Consider a useful cache block m for an 8-way LRU cache in Figure 29, where all blocks are mapped to the same cache set. The maximal age of m before its second access is 5. If the program is preempted at any point between the two accesses to m, for example at program point p, m will not be evicted from the cache as long as at most 3 ECBs from the preempting task map to the same set. So m's resilience is 3.

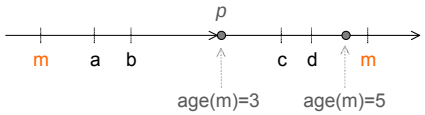


Figure 29 The notion of resilience.

By computing lower bounds on the resilience of useful cache blocks, one can exclude many cache misses compared with the conservative assumption in [17]. However, nested preemptions must be very carefully handled. The ECBs from nested preempting tasks may accumulate to age a useful block. In this case, the ECBs of all possible preempting tasks must be considered, which may adversely introduce some pessimism.

The problem of reordered misses is rooted in LRU. A new policy called Selfish-LRU [102] has been proposed to eliminate reordered misses. The idea is to first evict cache blocks that do not belong to the currently active task.

For other replacement policies, such as FIFO and PLRU, the number of additional misses can even be greater than the number of UCBs, the number of cache ways, and the number of ECBs [17]. This makes it difficult to obtain precise CRPD bounds for these policies. An approach based on relative competitiveness [103] was sketched in [17] that allows bounding the total misses (intra- and inter-task misses) of a non-LRU policy from the results of LRU. Due to the generic nature of the relative competitiveness framework, the analysis results can be imprecise.

5.1.4 Other Analyses

It has been observed [2] that some pessimism is introduced by independently computing bounds on the CRPD and on the WCET. Consider the treatment of memory accesses to blocks that have been classified as useful cache blocks during the WCET analysis. If such accesses cannot be guaranteed to result in cache hits, a sound WCET analysis will also cover the cache miss case. However, in that case, while a preemption-related cache miss may occur in reality, it has already been accounted for in the computed WCET bound. Motivated by this observation, a notion of *definitely-cached* UCBs has been proposed in [2], which excludes such blocks from the CRPD computation. Excluding such blocks from the CRPD computation had previously been proposed by Schneider [107] in what he calls the "isolated method". This approach relies on a coupling of WCET and CRPD analysis and may improve precision significantly.

Ramaprasad and Mueller [98, 100] presented an approach to response-time analysis for strictly-periodic task sets under fixed-priority scheduling, taking into account the CRPD in data caches. Due to their assumption of periodic tasks, they are able to simulate the scheduler during a hyperperiod of the system. Taking into account BCET and WCET estimates they can then accurately predict the number of preemptions, and to some extent even the preemption points within each job, which are taken into account in the CRPD analysis. In addition to the restriction to periodic task sets, this work shares the limitations of the data-cache analysis framework [97] that it builds upon, i.e., neither input-dependent memory accesses nor input-dependent control flows are supported. This approach was later extended to support a single non-preemptive region within each task [99].

CRPD analysis under dynamic priority scheduling has also been studied [63, 79]. The difference lies in the CRPD calculation for different schedulability tests of new scheduling policies. Lunniss et al. [78] compared the effectiveness of fixed priority scheduling and EDF in the presence of CRPD. Phavorin et al. [96] showed that common assumptions about optimality and sustainability of scheduling algorithms do not hold anymore, once CRPD is taken into account.

An alternative to bounding the cost of individual preemptions and taking this cost into account within schedulability analysis is to conservatively account for all possible preemptions within WCET analysis. This was first proposed by Schneider [107] in what he calls the "integrated method". The advantage of such an approach is that it can take into account overlapping of memory latencies and computations in pipelined processors. This advantage, however, is usually outweighed by overestimating the number of additional misses that may occur, as an unbounded number of preemptions needs to be taken into account.

5.1.5 Limiting Preemptions

We have focused on approaches to bound the CRPD under fully-preemptive scheduling. Various mechanisms exist to reduce the number of preemptions, and thereby also the overheads introduced by preemptions. In the extreme case, tasks are scheduled non-preemptively, eliminating preemption cost entirely. However, under non-preemptive scheduling, long-running lower priority tasks often render task sets unschedulable. Prominent approaches that represent a compromise between the extremes of fully-preemptive and non-preemptive scheduling are *preemption thresholds* [125], *cooperative scheduling* [19], and *floating non-preemptive regions* [13].

Under fixed-priority scheduling with preemption thresholds, in addition to its priority, a task is associated with a preemption threshold. When a task is running, it can only be preempted by tasks whose priority is higher than the preemption threshold of the running task. This has been shown to sometimes improve schedulability and reduce the number of preemptions [125]. Schedulability analysis taking into account CRPD under preemption thresholds [16] relies on the same basic cache analysis concepts, i.e., UCBs and ECBs, as schedulability analyses under standard fixed-priority preemptive scheduling.

Under cooperative scheduling, each task allows the scheduler to preempt it at *fixed preemption points*, i.e., program locations at which it yields to the scheduler. Between preemption points, a task is executed non-preemptively. This introduces lower-priority blocking time, which may reduce schedulability, and requires analysis of the maximum blocking time [3]. The CRPD analyses described above, based on UCBs, ECBs, and resilience can be applied to fixed preemption points in a straightforward manner. A challenge is to select preemption points in a way that maximizes a task set's schedulability. Fewer preemption points may result in lower CRPD but may increase the maximum blocking time. For programs consisting of straight-line code only, Bertogna et al. [15] provided an algorithm to optimally select preemption points. They assumed that the WCET is inflated accounting for preemptions at every preemption point by arbitrary preempting tasks, rather than accounting for the effect of preemptions within response-time analysis based on separate CRPD bounds.

In the floating non-preemptive region scheduling model [13], preemptions are limited as follows: When the highest priority task is executing, and a job of a higher priority task is released, the running job is not immediately preempted. Instead, a floating non-preemptive region of fixed length Q_i commences, where Q_i may depend on the task τ_i currently running. The currently running job is preempted after Q_i time units, unless it completes before. Again, CRPD analyses based on UCBs, ECBs, and resilience can be applied in this setting. However, as in the case of fixed preemption points, it is also possible to derive WCET bounds for all tasks that hold under the assumption that any two preemptions of a task are separated by at least Q_i time units. Given such WCET bounds, schedulability analyses do not need to further account for preemption costs. Marinho et al. [83, 84] provided analyses to bound a task's WCET including CRPD taking into account the length of floating non-preemptive regions. Their algorithm consists of two phases:

1. First, a so-called preemption delay function f is computed. For any time t, $f(t)$ is a bound on the CRPD if the task is preempted after t time units. In order to compute f, they determine

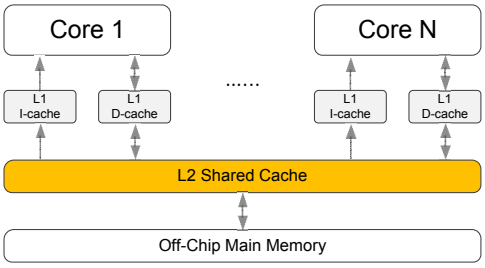


Figure 30 A common shared cache design in multi-cores.

a bound on the CRPD for each program point based on UCBs. Given lower and upper bounds on the execution time of each basic block, they determine lower and upper bounds on when a particular basic block may be running. Combining the two values yields f.

2. Then they incrementally determine based on f, one preemption at a time, the minimal progress over time in the execution of a program, when it can be preempted each Q_i time units.

If floating non-preemptive regions are short such an approach may be pessimistic as it does not take into account how often and by which tasks a task may actually be preempted. On the other hand, as in the "integrated method" of Schneider [107], it may take into account overlapping of memory latencies and computations in pipelined processors. To our knowledge, the two approaches have not been experimentally compared in the limited preemption context. This also applies to Bertogna et al. [15].

A comprehensive survey of the literature on cache-related preemption delays is given by Phavorin and Richard [95].

5.2 Shared Caches in Multi-Cores

Nowadays, multiple processing cores are deployed on a single die to fully exploit the real estate of the processor chip and to achieve high performance with low power consumption. A commonality among modern multi-core processors is the sharing of on-chip resources among multiple cores, such as the last-level cache (Figure 30), so that each core can potentially make use of the entire resource. However, tasks running in parallel on different cores compete for the shared resource, resulting in *inter-core conflicts*, also referred to as *inter-core interference*. Due to this interference, the execution time of a program now also depends on the resource-access behavior of the tasks running in parallel [133].

Inter-core interference on a shared cache is different from inter-task interference due to preemption. First, in a single-core preemptive system, a higher priority task does not suffer from interference by a lower priority task, while in multi-core systems, all tasks running in parallel on different cores interfere with each other independently of their priority level. Second, in a single-core preemptive system, a task can only suffer from interference by preempting tasks a small number of times, no more than the total number of releases of the higher priority tasks; in contrast, on multi-cores, interference on a shared cache may come between any two consecutive cache accesses of a task. Precisely analyzing all possible interleaving cache accesses on a shared cache is notoriously difficult due to the huge number of cases to consider.

One approach is to extend the AI-based analysis to take into account the interference on the shared cache [73]. The basic idea is similar to the resilience analysis in CRPD analysis. Assume that two tasks, A and B, run in parallel and share a k-way L2 cache. To estimate A's WCET, multi-level analyses for A are first conducted without considering the interference from task B. Then, a second step analyzes task B to see whether its interference could cause the blocks of A

that are guaranteed to hit when A runs in isolation, to be evicted from the cache. Consider a block m of A: its maximal age, $age(m)$, in the cache can be extracted from a Must analysis. Then task B is analyzed to determine a bound on the number of interfering memory blocks that map to the same cache set as m , denoted by $\mathcal{M}$. If $\mathcal{M} \leq k - age(m)$ holds, m willl remain in the cache even in the presence of B's interference. Otherwise, m could be evicted from the cache due to B's execution, and its classification needs to be changed accordingly.

A major drawback of the above approach is that the timing of cache conflicts is not considered, i.e., all potential cache conflicts computed from cache mapping are included. However, if by some means we know that the lifetimes of two conflicting tasks (or cache accesses) do not overlap, some cache conflicts can be safely excluded. This is a key property to tighten the estimations. Zhang and Yan proposed a technique to exclude infeasible conflicts by exploring conflicting pairs of cache accesses [135]. Liang et al. in [73] explore the overlapping of the lifetimes of co-running programs. The timing of cache conflicts can also be precisely captured by model checking. Gustavsson et al. used timed automata to model the behavior of programs on shared caches [50]. Infeasible conflicts can be precisely excluded when the UPPAAL model checker explores the system model. However, due to state space explosion, model checking based analysis can hardly scale beyond 2 cores. Another model checking based method was proposed in [132]. The SPIN model checker was adopted to exclude the infeasible cache conflicts, but the models did not explore the exact timing of the cache conflicts.

Another major analysis obstacle is that uncertainty, introduced by particular analysis technique or inherent to a hardware feature, may be amplified in the presence of shared caches. For example, in AI-based analyses, pessimistic age prediction makes the blocks of the interfered task less resilient; similarly, pessimistic age prediction for the interfering task leads to an overestimated number of conflicting blocks. Another source of uncertainty is the separation of cache behavior analysis and path analysis [120], which adversely introduces "architecturally-infeasible" paths. Pruning such infeasible paths can help to tighten WCET estimations. Banerjee et al. proposed a finer-grained abstract domain, which associates path information into the traditional Must and May abstract states to exclude non-existent cache states due to infeasible paths [12]. Chattopadhyay and Roychoudhury proposed another technique that improves the prediction for NC blocks by excluding infeasible paths using model checking [25]. Both techniques can be integrated into the analysis framework of [73] to more precisely estimate shared cache interference.

Even with the above techniques, real-time system design still faces a problem: if the shared cache is freely used, the worst-case performance of the tasks also degrades. Therefore, recent research tried to employ mechanisms that provide temporal isolation on shared caches, which both simplifies cache analysis and at the same time reduces the WCET. Cache partitioning [75, 116, 122] partitions the cache space among tasks by controlling page allocation to completely avoid cache conflicts among tasks on different cores. Cache locking [75, 116] locks the frequently used data in the cache so that hit/miss behavior is totally predictable. Another approach [54] tries to bypass the shared cache upon accesses to memory blocks with little reuse, which reduces cache interference. The idea of bypassing [54] was later extended to shared data caches [68], in which two heuristics are introduced to bypass indeterministic data references. On the system level, some further issues have to be solved. In multi-tasking systems, different tasks may try to lock the same cache segment, so scheduling of the lockings must be considered [126]. Regarding cache partitioning, the partitions assigned to the tasks may overlap in cache space. A task can only start execution if both the CPU and the cache partition are available. The schedulability tests must consider both the CPU and the cache constraints [46]. However, partitioning and locking have a side-effect of reducing the cache space available for each task. New techniques are expected for more intelligent resource allocation and arbitration, so that the WCET of the tasks can

Table 5 WCET analysis tools supporting static cache analysis.

Tools	Instruction Cache	Data Cache	Multi-Level Cache	Non-LRU Cache	CRPD	Shared Cache
aiT	AI	AI		Pseudo-RR, PLRU, FIFO		
OTAWA	AI, ML-PER					
Chronos		Scope-Aware	Separate Framework			
Heptane	AI	AI	Separate Framework		Separate Analysis	
WCA	Model Checking			FIFO		
SWEET	AI					
METAMOC	Model Checking	Model Checking		Round Robin		
McAiT	AI		Separate Framework			Model Checking
Florida State, NC State, Furman University	SCS	SCS, CME	Separate Framework		Separate Analysis	
Chalmers University	Symbolic Execution	Symbolic Execution				

be further reduced (Unlike the general-purpose computing domain [136], cache management in real-time systems [82] optimizes the worst-case rather than the average-case performance.), and the schedulability of the overall system is improved.

6 Static Analysis Tools

In the past decades, a number of WCET analysis tools have been developed in both industry and academia. Table 5 lists the tools that support static cache analysis.

aiT [57] is the only static WCET analysis tool in routine use in industry. It has been *qualified*, i.e., admitted to certification of time-critical avionics subsystems of several Airbus planes by the European European Aviation Safety Agency (EASA) and has been used in their certification. It is also used in other air and space companies in Europe, the United States, and China and in German automotive OEMs and their suppliers. It uses the AI-based analyses [31, 38] for both instruction and data caches. Besides LRU, the aiT tool can analyze three non-LRU replacement policies: Pseudo-Round-Robin [58] as well as, PLRU and FIFO based on the analyses described in [103] and [44].

The OTAWA tool [11] developed by the University of Toulouse, France, is an open framework for WCET analysis. OTAWA provides instruction cache analysis based on abstract interpretation [38] with the improvement of multi-level Persistence analysis [10].

Chronos [70] is a static WCET analysis tool from the National University of Singapore. It was originally designed with a highlight on pipeline analysis using the SimpleScalar simulator. The

latest version, Chronos 4.2, now supports the recent contributions of the group: scope-aware data cache analysis [62] and unified cache analysis [24].

Heptane [60] is a static WCET analysis tool developed by IRISA, France. The highlight of the tool is the separate analysis of multi-level caches [55]. It also support shared cache analysis by the technique extended from AI-based analysis [54].

WCA [109] from Vienna University of Technology and DTU is a WCET analysis tool for a Java processor, JOP [108], which uses *method cache* to store the instructions of a whole Java method. A method is fully loaded into the cache upon invocation and enjoys cache hits during its execution. On exit, the content of the caller function is reloaded into the method cache. The method cache is organized like a fully-associative FIFO cache with N blocks. The tool uses model checking to analyze the method cache, and it also provides a simple persistence analysis given that a code region can fit into the cache.

SWEET [117] is a WCET analysis tool currently maintained by Mälardalen University of Sweden. Although mainly focusing on flow analysis, it supports AI-based analysis for instruction caches [38].

The METAMOC tool [33] from Aalborg University of Denmark employs model checking for both instruction and data caches. It can analyze the round-robin replacement policy used by the ARM920T processor.

McAiT [80] is a WCET analysis tool jointly developed by Uppsala University of Sweden and Northeastern University of China. The tool supports L1 instruction cache analysis by the AI-based approaches [31, 38], and shared L2 cache analysis by model checking.

Other research prototypes include a tool from Florida State, North Carolina State, and Furman Universities, which adopts Static Cache Simulation [88] for both instruction and data cache analysis, and also supports data cache analysis using cache miss equations [97]. Another prototype from Chalmers University of Technology uses symbolic execution [77] for cache analysis.

The data provided in Table 5 might be imprecise, because the information can only be inferred from the publications instead of the tools in some cases. More comprehensive knowledge on existing WCET analysis tools can be found in [130] and the reports for the WCET Tool Challenge in 2011 [53], 2008 [61] and 2006 [48].

7 Future Research Directions

WCET estimation is a key task in timing analysis of real-time systems. Since caches may significantly affect execution time, the quality of cache analysis determines the precision of the estimated WCET. This article surveys the main challenges and analysis techniques for vast cache architectures. For decades, the LRU replacement policy has been well studied. The most valuable asset is that a comprehensive understanding of cache behavior and cache analysis were established by the ingenious researchers in related communities. Several future directions can be explored to bridge the gap.

Evaluation and Comparison of Different Approaches. The reader of this survey may be disappointed not to find evaluations and comparisons of the different methods for cache analysis. These are indeed hard to find in literature and are therefore subject of future research.

One of the obstacles to fair evaluation is the difficulty to obtain industrial software for experimentation; most industrial embedded software is not openly available. Another reason for the non-existence of good experimental evaluations is the dominance of the *Mälardalen Benchmark Suite*. The programs in this benchmark suite have a very special characteristic: They are small and contain tiny loops. They start with long straight-line code sequences for the initialization of the program variables. This alone makes them already problematic; the execution is independent

of the values of the input variables. Measuring this one execution would suffice if execution times were independent of the initial architectural state—which they often are not [104]. Analyzing the cache performance using the programs from this benchmark should stress the cache; access sequences without evictions do not provide any insights.

Analytical comparisons of the different approaches should, in principle, be possible. However, they have not been performed. For instance, the abstract cache state in Mueller's static cache simulation [88, 90] is much like the abstract May cache in the abstract-interpretation-based approach [37, 38]. Intuition says that the precision of May information as obtained by abstract interpretation should therefore be the same as that obtained by static cache simulation. We would, however, assume that Must information as obtained by abstract interpretation is more precise than that obtained by static cache simulation since the computation of Must information from an abstract May cache employs rather strong conditions to eliminate contents from the May cache that cannot be guaranteed to be in all concrete caches.

Non-LRU Cache Analysis. Although LRU is highly predictable, it is practically more important to analyze non-LRU replacement policies since they are actually adopted in real-life processors. Must, May and Persistence analyses need to be established to fully characterize the cache behavior. Currently, the missing pieces are Persistence analysis for FIFO, Must and May analyses for MRU, Persistence and May analyses for PLRU. Furthermore, there are no techniques to analyze non-LRU data caches and multi-level caches, which are actually required to cover the whole cache hierarchy. For policies other than the above-mentioned ones, similar analysis targets should be fulfilled. However, we still lack a systematic way to construct abstract analyses for new replacement policies.

Application of Cache Analysis in Other Domains. So far the use of cache analysis has mostly been confined to WCET analysis. However, there is at least one more domain in which cache analysis can deliver valuable insights, namely security. *Side-channel attacks* recover secret inputs to programs from physical characteristics of the computation. Typical goals of such attacks are the recovery of cryptographic keys and private information about users. Characteristics that have been exploited for that purpose include execution time, cache behavior, memory and power consumption, and electromagnetic radiation. Doychev et al. have demonstrated that static cache analyses based on abstract interpretation can be used to derive guarantees on the amount of information leaked to an attacker [35].

Design and Analysis of Timing-Predictable Embedded Systems. Preemption delay analysis and multi-core shared cache analysis have to consider the interactions among tasks running in parallel. It is commonly acknowledged that inter-core interference not only harms cache analysis, but also degrades overall system performance. Academia has gradually come to a consensus [8, 29, 122]: the solution to this problem should be to regulate both the hardware [92, 131] and the software [36, 81, 94] so that the system behaves in a timely predictable manner. The grand challenge is to obtain predictability without sacrificing the performance provided by future powerful processors. Cache analysis will provide valuable insights to characterize tasks so that good design decisions can be made in resource allocation and arbitration, such as cache partitioning and cache-aware scheduling.

One important step in this direction would be to understand how *timing compositionality* [52] can be achieved. Due to complex interactions between caches and other microarchitectural components, such as branch predictors or out-of-order pipelines, provably sound WCET analyses can currently only be achieved by analyzing all of these components together in an *integrated* fashion. However, such an integrated approach is very unlikely to scale to multi-tasking systems

or even to the parallel execution of multiple tasks on a multi-core processor. In these scenarios, to limit analysis complexity, interference costs are better analyzed separately and then taken into account during schedulability analysis. Timing compositionality has, however, not been formally proven for models of *any* modern microarchitecture, leaving much of the recent work unapplicable to real systems.

Acknowledgement. This article was partially supported by National Natural Science Foundation of China (61370076 and 61532007), Collaborative Innovation Center of Major Machine Manufacturing in Liaoning, and State Key Laboratory of Synthetical Automation for Process Industries (PAL-N201503).

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996. doi:10.1007/3-540-61739-6_33.

2 Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, pages 109–118. IEEE Computer Society, 2009. doi:10.1109/ECRTS.2009.21.

3 Sebastian Altmeyer, Claire Burguiere, and Reinhard Wilhelm. Computing the maximum blocking time for scheduling with deferred preemption. In *Future Dependable Distributed Systems, 2009 Software Technologies for*, pages 200–204, March 2009. doi:10.1109/STFSSD.2009.12.

4 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority preemptive systems. *Real-Time Systems*, 48(5):499–526, 2012. doi:10.1007/s11241-012-9152-2.

5 Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In Jaejin Lee and Bruce R. Childers, editors, *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010*, pages 153–162. ACM, 2010. doi:10.1145/1755888.1755911.

6 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

7 Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=238595.

8 Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014. doi:10.1145/2560033.

9 Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 – April 2, 2004, Proceedings*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004. doi:10.1007/978-3-540-24723-4_2.

10 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.

11 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems – 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

12 Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Precise micro-architectural modeling for WCET analysis via AI+SAT. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 87–96. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531082.

13 Sanjoy K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, pages 137–144. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.32.

14 Christoph Berg. PLRU cache domino effects. In Frank Mueller and Frank Mueller, editors, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2006.672.

15 Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio C. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In Karl-Erik Årzén, editor, *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*, pages 217–227. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.28.

16 Reinder J. Bril, Sebastian Altmeyer, Martijn M. H. P. van den Heuvel, Robert I. Davis, and Moris Behnam. Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 161–172. IEEE Computer Society, 2014. doi:10.1109/RTSS.2014.25.

17 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches – pitfalls and solutions. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/OASIcs.WCET.2009.2285.

18 Claire Burguière and Christine Rochange. A contribution to branch prediction modeling in WCET analysi. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 612–617. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.7.

19 Alan Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In Sang H. Son, editor, *Advances in Real-time Systems*, pages 225–248. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. URL: http://dl.acm.org/citation.cfm?id=207721.207731.

20 José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *2nd IEEE Real-Time Technology and Applications Symposium, RTAS'96, Boston, MA, USA, June 10-12, 1996*, pages 204–212. IEEE Computer Society, 1996. doi:10.1109/RTTAS.1996.509537.

21 Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

22 Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a timing anomaly? In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/OASIcs.WCET.2012.1.

23 Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 286–297. ACM, 2001. doi:10.1145/378795.378859.

24 Sudipta Chattopadhyay and Abhik Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 47–56. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.20.

25 Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 – December 2, 2011*, pages 193–203. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.25.

26 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

27 Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In Pen-Chung Yew, editor, *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*, pages 278–285. ACM, 1996. doi:10.1145/237578.237617.

28 Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000. doi:10.1023/A:1008149332687.

29 PREDATOR Consortium. The predator project page, 2011. URL: http://www.predator-project.eu/consortium.htm.

30 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.

31 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embedded Comput. Syst.*, 12(1s):40, 2013. doi:10.1145/2435227.2435236.

32 Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: modular execution time analysis using model checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123. Schloss Dagstuhl

– Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.113.

33 Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: modular execution time analysis using model checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.113.

34 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

35 Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446. USENIX Association, 2013. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

36 Heiko Falk and Helena Kotthaus. Wcet-driven cache-aware code positioning. In Rajesh K. Gupta and Vincent John Mooney, editors, *Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 145–154. ACM, 2011. doi:10.1145/2038698.2038722.

37 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, Saarbruecken, Germany, 1997. ISBN: 3-9307140-31-0.

38 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.

39 Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999. doi:10.1145/325478.325479.

40 Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, Saarbruecken, Germany, 2011. ISBN: 978-3-8442-1699-8.

41 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In 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*, pages 120–136. Springer, 2009. doi:10.1007/978-3-642-03237-0_10.

42 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.

43 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 *OpenAccess Series in Informatics (OASIcs)*, pages 23–35. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.23.

44 Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture – Embedded Systems Design*, 57(6):625–637, 2011. doi:10.1016/j.sysarc.2010.05.013.

45 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In Marco Di Natale, editor, *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China, April 16-19, 2012*, pages 55–64. IEEE Computer Society, 2012. doi:10.1109/RTAS.2012.31.

46 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 245–254. ACM, 2009. doi:10.1145/1629335.1629369.

47 Nan Guan, Xinping 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.

48 Jan Gustafsson. WCET challenge 2006 – technical report. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-206/2007-1-SE, Mälardalen University Sweden, January 2007. URL: http://www.es.mdh.se/publications/1020-.

49 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 57–66. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.12.

50 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 101–112. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.101.

51 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 102–111. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.14.

52 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time

analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

53 Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET tool challenge 2011: Report. In *the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*, July 2011.

54 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 68–77. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.34.

55 Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November – 3 December 2008*, pages 456–466. IEEE Computer Society, 2008. doi:10.1109/RTSS.2008.10.

56 Damien Hardy and Isabelle Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture – Embedded Systems Design*, 57(7):677–694, 2011. doi:10.1016/j.sysarc.2010.08.007.

57 Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. *The whitepaper of aiT*, 2014. URL: https://www.absint.com/aiT_WCET.pdf.

58 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. doi:10.1109/JPROC.2003.814618.

59 John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

60 Heptane. The Heptane tool page, 2013. URL: http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29&Itemid=0&lang=en.

61 Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne De Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET 2008 – report from the tool challenge 2008 – 8th intl. workshop on worst-case execution time (WCET) analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1663, doi:10.4230/OASIcs.WCET.2008.1663.

62 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.

63 Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Design, Automation Test in Europe Conference Exhibition, 2007*, pages 1–6, April 2007. doi:10.1109/DATE.2007.364534.

64 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.

65 Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *2nd IEEE Real-Time Technology and Applications Symposium, RTAS'96, Boston, MA, USA, June 10-12, 1996*, pages 230–240. IEEE Computer Society, 1996. doi:10.1109/RTTAS.1996.509540.

66 Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong-Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Trans. Computers*, 47(6):700–713, 1998. doi:10.1109/12.689649.

67 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/OASIcs.WCET.2009.2283.

68 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data cache conflicts reduction for wcet computation in multi-core architectures. In *Proc. of the 18th Real-Time and Network Systems, Toulouse, France*, 2010.

69 Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006. doi:10.1007/s11241-006-9205-5.

70 Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007. doi:10.1016/j.scico.2007.01.014.

71 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995. doi:10.1145/217474.217570.

72 Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96), December 4-6, 1996, Washington, DC, USA*, pages 254–263. IEEE Computer Society, 1996. doi:10.1109/REAL.1996.563722.

73 Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012. doi:10.1007/s11241-012-9160-2.

74 Fang Liu and Yan Solihin. Understanding the behavior and implications of context switch misses. *TACO*, 7(4):21, 2010. doi:10.1145/1880043.1880048.

75 Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. Task assignment with cache partitioning and locking for WCET minimization on mpsoc. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 573–582. IEEE Computer Society, 2010. doi:10.1109/ICPP.2010.65.

76 Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA'99), 13-16 December 1999, Hong Kong, China*, pages 255–262. IEEE Computer Society, 1999. doi:10.1109/RTCSA.1999.811244.

77 Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA'99), 13-16 December 1999, Hong Kong, China*, pages 255–262. IEEE Computer Society, 1999. doi:10.1109/RTCSA.1999.811244.

78 Will Lunniss, Sebastian Altmeyer, and Robert I. Davis. A comparison between fixed priority and EDF scheduling accounting for cache related preemption delays. *LITES*, 1(1):01:1–01:24, 2014. doi:10.4230/LITES-v001-i001-a001.

79 Will Lunniss, Sebastian Altmeyer, Claire Maiza, and Robert I. Davis. Integrating cache related pre-emption delay analysis into EDF scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 75–84. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531081.

80 Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. Mcait – A timing analyzer for multicore real-time software. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 414–417. Springer, 2011. doi:10.1007/978-3-642-24372-1_29.

81 Mohamed Abdel Maksoud and Jan Reineke. A compiler optimization to increase the efficiency of WCET analysis. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS'14, Versaille, France, October 8-10, 2014*, page 87. ACM, 2014. doi:10.1145/2659787.2659825.

82 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 45–54. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531078.

83 José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. An improved preemption delay upper bound for floating non-preemptive region. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012, Karlsruhe, Germany, June 20-22, 2012*, pages 57–66. IEEE, 2012. doi:10.1109/SIES.2012.6356570.

84 José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 497–502. IEEE, 2012. doi:10.1109/DATE.2012.6176520.

85 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998. doi:10.1007/BFb0026424.

86 Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. doi:10.1147/sj.92.0078.

87 Frank Mueller. Timing predictions for multi-level caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.

88 Frank Mueller and David B. Whalley. Fast instruction cache analysis via static cache simulation. In *Proceedings 28st Annual Simulation Symposium (SS'95), April 25-28, 1995, Santa Barbara, California, USA*, pages 105–114. IEEE Computer Society, 1995. doi:10.1109/SIMSYM.1995.393589.

89 Frank Müller. Generalizing timing predictions to set-associative caches. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, RTS 1997, 11-13 June, 1997, Toledo, Spain*, pages 64–71. IEEE Computer Society, 1997. doi:10.1109/EMWRTS.1997.613765.

90 Frank Müller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, 2000. doi:10.1023/A:1008145215849.

91 Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In Rajesh Gupta, Yukihiro Nakamura, Alex Orailoglu, and Pai H. Chou, editors, *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS*

2003, Newport Beach, CA, USA, October 1-3, 2003, pages 201–206. ACM, 2003. doi:10.1145/944645.944698.

92 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 57–68. ACM, 2009. doi:10.1145/1555754.1555764.

93 Kaustubh Patil, Kiran Seth, and Frank Mueller. Compositional static instruction cache simulation. In David B. Whalley and Ron Cytron, editors, *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, pages 136–145. ACM, 2004. doi:10.1145/997163.997183.

94 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 269–279. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.33.

95 Guillaume Phavorin and Pascal Richard. Cache-related preemption delays and real-time scheduling: A survey for uniprocessor systems. Technical report, Laboratoire d'Informatique et d'Automatique pour les Systèmes, 2015. URL: http://www.lias-lab.fr/publications/19296/survey.pdf.

96 Guillaume Phavorin, Pascal Richard, Joël Goossens, Thomas Chapeaux, and Claire Maiza. Scheduling with preemption delays: anomalies and issues. In Julien Forget, editor, *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 109–118. ACM, 2015. doi:10.1145/2834848.2834853.

97 Harini Ramaprasad and Frank Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005), 7-10 March 2005, San Francisco, CA, USA*, pages 148–157. IEEE Computer Society, 2005. doi:10.1109/RTAS.2005.12.

98 Harini Ramaprasad and Frank Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 71–80. IEEE Computer Society, 2006. doi:10.1109/RTAS.2006.14.

99 Harini Ramaprasad and Frank Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*, pages 58–67. IEEE Computer Society, 2008. doi:10.1109/RTAS.2008.18.

100 Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Trans. Embedded Comput. Syst.*, 10(2):27, 2010. doi:10.1145/1880050.1880063.

101 Jan Reineke. *Caches in WCET Analysis: Predictability – Competitiveness – Sensitivity*. PhD thesis, Saarland University, 2009. URL: http://www.epubli.de/shop/buch/Caches-in-WCET-Analysis-Jan-Reineke-9783941071698/12835.

102 Jan Reineke, Sebastian Altmeyer, Daniel Grund, Sebastian Hahn, and Claire Maiza. Selfish-lru: Preemption-aware caching for predictability and performance. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 135–144. IEEE Computer Society, 2014. doi:10.1109/RTAS.2014.6925997.

103 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.

104 Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. *ACM Trans. Embedded Comput. Syst.*, 12(1s):42, 2013. doi:10.1145/2435227.2435238.

105 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

106 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006. doi:10.4230/OASIcs.WCET.2006.671.

107 Jörn Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*, pages 195–204. IEEE Computer Society, 2000. doi:10.1109/REAL.2000.896009.

108 Martin Schoeberl. A java processor architecture for embedded real-time systems. *Journal of Systems Architecture – Embedded Systems Design*, 54(1-2):265–286, 2008. doi:10.1016/j.sysarc.2007.06.001.

109 Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a java processor. *Softw., Pract. Exper.*, 40(6):507–542, 2010. doi:10.1002/spe.968.

110 Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EM-*

SOFT 2007, September 30 – October 3, 2007, Salzburg, Austria, pages 203–212. ACM, 2007. doi:10.1145/1289927.1289960.

111 Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 – December 3, 2010*, pages 395–404. IEEE Computer Society, 2010. doi:10.1109/RTSS.2010.8.

112 Jan Staschulat and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. In *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*, pages 227–236. IEEE Computer Society, 2006. doi:10.1109/ECRTS.2006.33.

113 Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for real-time software. *ACM Trans. Embedded Comput. Syst.*, 6(4), 2007. doi:10.1145/1274858.1274863.

114 Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, pages 41–48. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.26.

115 Martin Stigge and Wang Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 51(5):602–636, 2015. doi:10.1007/s11241-015-9234-z.

116 Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 300–303. ACM, 2008. doi:10.1145/1391469.1391545.

117 SWEET. The SWEET tool page, 2012. URL: http://www.mrtc.mdh.se/projects/wcet/sweet/online/content/index.php.

118 Yudong Tan and Vincent John Mooney III. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embedded Comput. Syst.*, 6(1), 2007. doi:10.1145/1210268.1210275.

119 Yudong Tan and Vincent John Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In Henk Schepers, editor, *Software and Compilers for Embedded Systems, 8th International Workshop, SCOPES 2004, Amsterdam, The Netherlands, September 2-3, 2004, Proceedings*, volume 3199 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2004. doi:10.1007/978-3-540-30113-4_14.

120 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000. doi:10.1023/A:1008141130870.

121 Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In Frank Vahid and Jan Madsen, editors, *Proceedings of the Eighth International Workshop on Hardware/Software Codesign, CODES 2000, San Diego, California, USA, 2000*, pages 67–71. ACM, 2000. doi:10.1145/334012.334025.

122 Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

123 Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 154–165. IEEE Computer Society, 2003. doi:10.1109/REAL.2003.1253263.

124 Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, Massachusettes, USA, February 2-6, 2002*, pages 175–186. IEEE Computer Society, 2002. doi:10.1109/HPCA.2002.995708.

125 Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA'99), 13-16 December 1999, Hong Kong, China*, page 328. IEEE Computer Society, 1999. doi:10.1109/RTCSA.1999.811269.

126 Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 157–167. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.26.

127 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*, pages 430–444. Springer, 2008. doi:10.1007/978-3-540-88479-8_30.

128 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.

129 Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, 1999. doi:10.1023/A:1008190423977.

130 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller,

Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.

131 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

132 Lan Wu and Wei Zhang. A model checking based approach to bounding worst-case execution time for multicore processors. *ACM Trans. Embedded Comput. Syst.*, 11(S2):56, 2012. doi:10.1145/2331147.2331166.

133 Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*, pages 80–89. IEEE Computer Society, 2008. doi:10.1109/RTAS.2008.6.

134 Patrick Meumeu Yomsi and Yves Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *19th Euromicro Conference on Real-Time Systems, ECRTS'07, 4-6 July 2007, Pisa, Italy, Proceedings*, pages 280–290. IEEE Computer Society, 2007. doi:10.1109/ECRTS.2007.15.

135 Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*, pages 455–463. IEEE Computer Society, 2009. doi:10.1109/RTCSA.2009.55.

136 Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 89–102. ACM, 2009. doi:10.1145/1519065.1519076.