Deployment and Debugging of Real-Time Applications on Multicore Architectures

by

Hany Kashif

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

In what follows is a list of publications which I have co-authored and used their content in this dissertation. For each publication, I present a list of my contributions.

The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

1. Hany Kashif, Hiren D. Patel, and Sebastian Fischmeister. Using Link-level Latency Analysis for Path Selection for Real-time Communication on NoCs. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC),* Sydney, Australia, February 2012 [67].

    - Defined the path selection algorithm
    - Defined the heuristics used by the algorithm
    - Implemented the algorithm
    - Designed and executed the experiments
    - Analyzed the experimental results
    - Wrote a significant portion of the article

2. Hany Kashif and Sebastian Fischmeister. Program Transformation for Time-aware Instrumentation. In *Proceedings of the 17th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, Krakow, Poland, September 2012 [63].

    - Defined the program transformation techniques
    - Defined the instrumentable edges detection algorithm
    - Defined the solution to the overhead optimization problem
    - Implemented the instrumentation tool
    - Designed and executed the experiments
    - Analyzed the experimental results
    - Wrote a significant portion of the article

3. Hany Kashif, Sina Gholamian, Rodolfo Pellizzoni, Hiren D. Patel, and Sebastian Fischmeister. ORTAP:An Offset-based Response Time Analysis for a Pipelined Communication Resource Model. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, USA, April 2013 [65].

    - Involved in deriving the offset-based stage-level analysis
    - Formalized the the offset-based stage-level analysis
    - Derived and formalized the offset-based flow-level analysis
    - Implemented the offset-based stage-level analysis

- Involved in writing the article

4. Hany Kashif, Pansy Arafa, and Sebastian Fischmeister. INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Taipei, Taiwan, August 2013 [62].

   - Defined the instrumentation flow
   - Implemented the instrumentation framework
   - Designed the experiments
   - Wrote a significant portion of the article

5. Hany Kashif and Hiren Patel. Bounding Buffer Space Requirements for Real-Time Priority-Aware Networks. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, SunTec, Singapore, January 2014 [66].

   - Defined the problem and the solution
   - Defined and formalized the buffer space analysis techniques
   - Implemented the buffer space analysis techniques
   - Designed and executed the experiments
   - Analyzed the experimental results
   - Wrote a significant portion of the article

6. Hany Kashif, Sina Gholamian, and Hiren Patel. SLA: A Stage-level Latency Analysis for Real-time Communication in a Pipelined Resource Model. In *IEEE Transactions on Computers*, 64(4), April 2014 [64].

   - Defined a solution to analyze communication at the stage-level
   - Formalized the stage-level analysis
   - Implemented the stage-level analysis
   - Designed and executed the experiments
   - Analyzed the experimental results
   - Wrote a significant portion of the article

7. Hany Kashif, Johnson Thomas, Hiren Patel, and Sebastian Fischmeister. Static Slack-Based Instrumentation of Programs. In *Proceedings of the 20th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, Luxembourg, September 2015 [68].

   - Involved in designing the algorithm for minimizing instrumentation points
   - Defined a solution to the overhead optimization problem

- Implemented the software and hardware solutions
- Designed and executed the experiments
- Analyzed the experimental results
- Involved in writing the article

Abstract

It is essential to enable information extraction from software. Program tracing techniques are an example of information extraction. Program tracing extracts information from the program during execution. Tracing helps with the testing and validation of software to ensure that the software under test is correct. Information extraction is done by instrumenting the program. Logged information can be stored in dedicated logging memories or can be buffered and streamed off-chip to an external monitor. The designer inspects the trace after execution to identify potentially erroneous state information. In addition, the trace can provide the state information that serves as input to generate the erroneous output for reproducibility.

Information extraction can be difficult and expensive due to the increase in size and complexity of modern software systems. For the sub-class of software systems known as real-time systems, these issues are further aggravated. This is because real-time systems demand timing guarantees in addition to functional correctness. Consequently, any instrumentation to the original program code for the purpose of information extraction may affect the temporal behaviors of the program. This perturbation of temporal behaviors can lead to the violation of timing constraints, which may bias the program execution and/or cause the program to miss its deadline. As a result, there is considerable interest in devising techniques to allow for information extraction without missing a program's deadline that is known as time-aware instrumentation. This thesis investigates time-aware instrumentation mechanisms to instrument programs while respecting their timing constraints and functional behavior. Knowledge of the underlying hardware on which the software runs, enables the extraction of more information via the instrumentation process.

Chip-multiprocessors offer a solution to the performance bottleneck on uni-processors. Providing timing guarantees for hard real-time systems, however, on chip-multiprocessors is difficult. This is because conventional communication interconnects are designed to optimize the average-case performance. Therefore, researchers propose interconnects such as the priority-aware networks to satisfy the requirements of hard real-time systems. The priority-aware interconnects, however, lack the proper analysis techniques to facilitate the deployment of real-time systems. This thesis also investigates latency and buffer space analysis techniques for pipelined communication resource models, as well as algorithms for the proper deployment of real-time applications to these platforms.

The analysis techniques proposed in this thesis provide guarantees on the schedulability of real-time systems on chip-multiprocessors. These guarantees are based on reducing contention in the interconnect while simultaneously accurately computing the worst-case communication latencies. While these worst-case latencies provide bounds for computing the overall worst-case execution time of applications on chip-multiprocessors, they also provide means to assigning instrumentation budgets required by time-aware instrumentation. Leveraging these platform-specific analysis techniques for the assignment of instrumentation budgets, allows for extracting more information from the instrumentation process.

Acknowledgements

First and foremost, I am grateful to God for bestowing upon me the strength, knowledge, and good health to complete the research for this thesis.

I would like to seize this opportunity to thank my supervisor, Professor Sebastian Fischmeister, for his continuous encouragement, help, and support. His advice, tolerance, and consideration were invaluable. I also thank my co-supervisor, Professor Hiren Patel, for his constant guidance and thoughtfulness.

I would like to thank Professor Rodolfo Pellizzoni for the valuable research discussions and sharp insights. I would like to thank Professor Mahesh Tripunitara for teaching me a mind-set and an approach to solving certain problems. I would like to thank Professor Mark Aagaard for giving me the opportunity to be, on various occasions, the teaching assistant of someone as respectful and decent as he is. I would like to thank Professor Siddharth Garg and Professor Borzoo Bonakdarpour for the numerous interesting research discussions. I would also like to thank my committee members: Professor Bill Cowan, Professor Catherine Gebotys, Professor Lin Tan, and Professor Kees Goossens for taking the time and effort to participate in my examination committee and provide me with valuable feedback.

I would like to thank my friends in the Real-Time Embedded Software Group at the University of Waterloo for all the beautiful moments we shared together. We have been together through thick and thin: Ramy Medhat, Mahmoud Salem, Pansy Arafa, Samaneh Navabpour, Johnson Thomas, Wallace Wu, Aymen Ketata, Augusto Born de Oliveira, Jean-Christophe Petkovic, Akramul Azim, and Sina Gholamian.

I cannot find the words to thank my parents, Ismail and Nahla, for their continuous and endless support throughout my whole life. I thank them for believing in me and for helping me be who I am today. Thanks to my brothers, Mostafa and Ahmed, for the help and support they offer me. Thanks to my young and beautiful sister, Mariam, for representing all the beauty and innocence in this world.

I would like to express my deep gratitude for my wife, Pansy. Thanks for all the sacrifices, support, and endless patience. I also thank my son, Adham, for shining my days with his innocent smiles.

Dedication

To my lovely wife, Pansy.
To my son, Adham.
To my dear parents, Ismail and Nahla.
To my siblings, Ahmed, Mostafa, and Mariam.
To all those who touched my life.

Table of Contents

List of Tables

List of Figures

Abbreviations

ACK	Acknowledgment.
ALU	Arithmetic logic unit.
BIP	Binary integer programming.
CFG	Control-flow graph.
CI	Confidence interval.
CMP	Chip-multiprocessor.
CP	Conditional instrumentation point.
DAG	Directed acyclic graph.
DSP	Digital signal processing.
DUA	Definitive-use association.
ETP	Execution time profile.
ETPsem	Execution time profile shift-effectiveness metric.
FIFO	First in first out.
FLA	Flow-level analysis.
FLBA	Flow-level buffer-space analysis.
GALS	Globally asynchronous, locally synchronous.
GCC	GNU compiler collection.
II	Instrumentation intent.
ILP	Integer linear programming.
IP	Instrumentation point.
ISA	Instruction-set architecture.
LHS	Left hand side.
MCU	Micro-controller unit.
MIRA	Minimum interference routing algorithm.
MUX	Multiplexer.

NACK	Negative acknowledgment.
NLP	Non-linear programming.
NoC	Network-on-chip.
NP	Non-deterministic polynomial time.
OFLA	Offset-based flow-level analysis.
OLPC	One laptop per child.
OSCCFG	One-state change CFG.
OSLA	Offset-based stage-level analysis.
PAL	Palencia and Gonzalez's analyses.
PSA	Path selection algorithm.
RHS	Right hand side.
SEM	Standard error of mean.
SLA	Stage-level analysis.
SLBA	Stage-level buffer-space analysis.
SoC	System-on-chip.
SWP	Shortest widest path.
TDM	Time division multiplexing.
VC	Virtual channel.
WCET	Worst-case execution time.
WCL	Worst-case latency.
WCP	Worst-case path.
WCRT	Worst-case response time.
WSP	Widest shortest path.

List of Symbols

Time-Aware Instrumentation

α	The difference between the program's worst-case execution time and deadline.
β	The program's debugging time budget.
O	The overhead on the program's worst-case execution time due to instrumentation.
$\Upsilon(t)$	The execution time profile shift-effectiveness metric.
ϱ	The instrumentation coverage.

Stage-Level Analysis

Γ	A set of communication tasks deployed on the pipelined resource model.
τ_i	A communication task i.
P_i	Priority of communication task τ_i.
T_i	Period or minimum interarrival time between jobs of communication task τ_i.
D_i	Deadline of communication task τ_i.
J_i^R	Release jitter of communication task τ_i.
L_i	Worst-case latency of communication task τ_i on a single communication resource.
C_i	Worst-case latency of communication task τ_i along its path of communication resources from source to destination.
δ_i	A path formed of a series of contiguous communication resources of communication task τ_i from a source stage s_1 to a destination stage $s_{\lvert\delta_i\rvert}$.
$\sigma_i(s_l)$	A subsequence of the path δ_i of communication task τ_i starting with the same source stage s_1 and ending with a stage s_l..

$s_{i,l}$	A stage on the path δ_i of a communication task τ_i.
s'	A stage preceding stage s on the path of a communication task.
s^+	A stage succeeding stage s on the path of a communication task.
(v_i, v_j)	A communication resource (stage) connecting the two computation resources V_i and V_j.
$\underset{s}{\mathbb{S}_i^D}$	Set of directly interfering tasks with task τ_i on stage s of its path δ_i.
$\underset{s}{\mathbb{S}_i^I}$	Set of indirectly interfering tasks with task τ_i on stage s of its path δ_i.
$\underset{s}{J_i^I}$	Indirect interference jitter for a task under analysis on stage s through a directly interfering task τ_i.
R_i	Worst-case latency of communication task τ_i along its path δ_i.
$\underset{s}{R_i}$	Worst-case latency of communication task τ_i on stage s of its path δ_i.
$\underset{s}{I_i}$	The worst-case interference suffered by communication task τ_i from higher priority tasks on stage s of its path δ_i.
$\underset{s}{B_i}$	Busy period on stage s on the path of communication task τ_i.
$\underset{s}{w_i}(p)$	The worst-case completion time of each job p of communication task τ_i on stage s from the start of the busy period $\underset{s}{B_i}$.
$\underset{s}{I_i}(p)$	The worst-case interference suffered by job p of communication task τ_i from higher priority tasks on stage s of its path δ_i.
$\underset{s}{p_{B,i}}$	Maximum value of p for task τ_i on stage s.
F_i	The number of flits in one packet of communication task τ_i.
$\underset{s}{VC_i}$	The buffer space in the virtual channel of task τ_i in the priority-aware router sending flits on stage s of the task's path δ_i.
$R_{s_{i,l}}$	The delay that a data unit of communication task τ_i experiences as it moves from one stage $s_{i,l-1}$ to another stage $s_{i,l}$.
$\underset{s}{IB_i}$	The blockage suffered by task τ_i on stage s of its route δ_i due to limited buffer space in the downstream router of stage s.
CF	The credit feedback delay of the credit-based task control mechanism in the priority-aware network.
$\underset{s}{DR}$	The downstream router of stage s.

$\underset{s}{UR}$	The upstream router of stage s.

Offset-based WCRT Analysis

$\mathcal{A}$	A set of n applications deployed on the pipelined resource model.
A_i	An application i.
T_i	Period or minimum interarrival time for the execution of application A_i.
D_i	Deadline of application A_i.
J_i^R	Release jitter of application A_i.
G_{A_i}	Task graph of application A_i.
Γ_i^C	Set of computation tasks in application A_i.
Γ_i^M	Set of communication tasks in application A_i.
τ_{ik}	Task k of application A_i.
τ_{ikc}	The c-th job of task τ_{ik}.
P_{ik}	Priority of task τ_{ik}.
δ_{ik}	A path of communication task τ_{ik} formed of a series of contiguous communication resources from a source computation resource $v_{s_{ik}}$ to a destination $v_{d_{ik}}$.
L_{ik}	Worst-case latency of communication task τ_{ik} on a single communication resource.
C_{ik}	Worst-case execution time of computation task τ_{ik} on a processing resource $v_{c_{ik}}$ or worst-case latency of communication task τ_{ik} along its path of communication resources from source to destination.
J_{ik}^R	Release jitter of task τ_{ik}.
$\mathbb{S}_i^D(\tau_{ab})$	The set of tasks of application A_i that directly interfere with task τ_{ab} along its path δ_{ab}.
$\underset{s}{\mathbb{S}_i^D}(\tau_{ab})$	The set of tasks of application A_i that directly interfere with task τ_{ab} on stage s of its path δ_{ab}.
$\mathbb{S}_{ij}^I(\tau_{ab})$	The set of tasks indirectly interfering with task τ_{ab} through the intermediate task τ_{ij}.
$J_{ij}^I(\tau_{ab})$	Indirect interference jitter of task τ_{ij} due to the tasks in the indirect interference set of task τ_{ab}.

Π_{ik} The set of all paths in the task graph of application A_i, G_{A_i}, starting from τ_{i1} to τ_{ik}.

ρ_{ik} The set of tasks forming an individual path of Π_{ik}.

ϕ_{ik} The offset of task τ_{ik} relative to the activation of the root task of application A_i.

$\bar{\Theta}(t,s)$ A stage-normalized schedule where the transmission schedule $\Theta(t,s)$ on successive stages is moved earlier in time to account for stage delay so that release times coincide across all stages.

$\Theta(t,s)$ A function schedule that denotes the assignment of a data unit of a job τ_{abc} of task τ_{ab} to a particular slot t on stage s.

R_{ab} The worst-case response time of task τ_{ab} along its route δ_{ab} measured from the activation of the root task of application A_a.

$\hat{R}_{ab}$ The worst-case response time of task τ_{ab} along its route δ_{ab} measured from the activation of task τ_{ab}.

$\bar{R}_{ab}$ The worst-case response time of task τ_{ab} measured from the activation of task τ_{ab} in a stage-normalized schedule.

φ_{ijk} The phase between any task τ_{ij} and the beginning of the busy chain of a critical activation pattern created by task τ_{ik}.

$p^v_{0,ab}$ The first job instance of task τ_{ab} that has enough jitter to be part of the busy chain.

$W_i''^*(\tau_{ab},s,t)$ An upper bound to the worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage $s-1$ solely due to tasks that are on stage $s-1$ but not s, i.e., in the set $\underset{s-1}{\mathbb{S}}{}^{D}_{i}(\tau_{ab}) \backslash \underset{s}{\mathbb{S}}{}^{D}_{i}(\tau_{ab})$, and by considering each task in A_i to coincide with the critical instant.

$\underset{s}{p}{}^{v}_{B,ab}$ The largest-numbered job instance of task τ_{ab} which exists in the busy chain of τ_{ab} up to stage s.

$W_{ik}(\tau_{ab},s,t)$ The worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage s when the activation of task τ_{ik} coincides with the critical instant.

$W'_{ik}(\tau_{ab},s,t)$ The worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage $s-1$, when the activation of task τ_{ik} coincides with the critical instant, solely due to tasks that are common on stages $s-1$ and s, i.e., in the set $\underset{s-1}{\mathbb{S}}{}^{D}_{i}(\tau_{ab}) \cap \underset{s}{\mathbb{S}}{}^{D}_{i}(\tau_{ab})$.

$W_i^*(\tau_{ab},s,t)$ An upper bound to the worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage s by considering each task in A_i to coincide with the critical instant.

$W''_{ik}(\tau_{ab}, s, t)$ The worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage $s-1$, when the activation of task τ_{ik} coincides with the critical instant, solely due to tasks that are on stage $s-1$ but not s, i.e., in the set $\underset{s-1}{\mathbb{S}}{}_i^D(\tau_{ab}) \backslash \underset{s}{\mathbb{S}}{}_i^D(\tau_{ab})$.

$\underset{s}{w}{}_{ab}^{v}(p)$ The worst-case length of the busy chain for job p of task τ_{ab} up to stage s for an activation pattern v.

$\underset{s}{w}_{abc}(p)$ The worst-case length of the busy chain for job p of task τ_{ab} up to stage s for a critical instant created by tasl τ_{ac}.

$\underset{s}{R}{}_{ab}^{v}(p)$ The worst-case response time of job p of task τ_{ab} on a stage s for an activation pattern v measured from the activation of application A_a.

$\underset{s}{R}_{abc}(p)$ The worst-case response time of job p of task τ_{ab} on a stage s for a critical instant created by task τ_{ac} measured from the activation of application A_a.

Chapter 1

Introduction

In this thesis, we address two problems that exist in the field of real-time embedded systems. The first problem is the instrumentation of embedded software. It is essential to take timing constraints into account when instrumenting real-time software. Hence, it is crucial to find new instrumentation techniques that are more suited for real-time systems. The second problem is scalability. Modern chip-multiprocessors (CMPs) connect a large number of embedded processing elements using a pipelined communication interconnect. While the use of CMPs is becoming widespread in general purpose computing, its adoption for hard real-time systems has been cautious at best. This is because of the need to provide provable guarantees that the hard real-time software always meets its timing requirements. Providing tight bounds on the communication latencies between tasks deployed on CMPs can increase the instrumentation budget available for time-aware instrumentation. My main research focus is, thus, (1) providing solutions to instrumentation of real-time embedded software and (2) providing analysis techniques to enable using CMPs as a platform for real-time systems.

1.1 Tracing

Program tracing is the extraction of information from a program at runtime during its execution. The tracing technique was developed as early as programming itself [8, 10]. There are various ways for tracing program execution; instrumentation, sampling, and emulation. Instrumentation is the insertion of additional code to the original program to support tracing. The instrumentation process can either be applied to the source code or the binary executable of the program. It can also be a static or a dynamic process. Sampling can also be used to extract information about a running program. The program is unmodified and, at certain intervals, an external monitor interrupts the program to capture some parameters. Emulation as well is recognized as a method for tracing. The reason is that the emulated system can be totally controlled and every single instruction execution is visible for instrumentation.

A common example of tracing program execution is state logging where monitors extract certain critical software state information and program counter locations. Logged

information can be stored in dedicated logging memories or can be buffered and streamed off-chip to an external monitor. The designer inspects the trace after execution to identify potentially erroneous state information. In addition, the trace can provide the state information that serves as input to generate the erroneous output for reproducibility. These techniques are used during the software design process to test, validate, and debug the system, but they are also often used during deployment as well. This is to continue collecting trace information in the event of certain failures that require diagnosis.

1.1.1 Tracing of Real-Time Systems

Of all computers, 98% are embedded systems and plenty of those are real-time control systems (DARPA, 2000). This highlights the prevalence of real-time systems in our daily lives including, but not limited to: automotive vehicles, aeroplanes, cellular phones, etc. Real-time applications can be classified into hard real-time and soft real-time applications. For hard real-time applications, it is imperative to meet deadlines. Missing deadlines can lead to system failure which might result in loss of life for instance. Soft real-time applications, on the other hand, can occasionally miss deadlines without causing serious harm. Missing deadlines, however, leads to a degradation in the overall system performance.

While the various real-time applications have different non-functional requirements such as safety and memory consumption, they are all time-sensitive systems. In general, writing correct software is both difficult [71] and expensive [43]. This is further aggravated for real-time systems because in addition to functional correctness, timeliness is important. Tracing is one of the common techniques for debugging real-time systems. The instrumentation process, required to enable tracing, naturally causes perturbation to the system under analysis. Hence, any instrumentations to the original program code may affect the temporal behaviors of the system. Typically, the more tracing code the program executes during the run, the more the perturbation in temporal behaviors. The reason is that, in general, the addition of more tracing code increases the number of instructions the processor executes leading to a longer execution time (except for timing anomalies).

This perturbation of temporal behaviors raises issues during testing and validation in the software design process, and during deployment. In the former, violating the temporal constraints of the program may bias the execution of the program to certain operations, which otherwise would not occur in a real deployment. In the latter, deployment is simply not possible if the program may potentially miss its deadline. As a result, there is considerable interest in devising techniques to allow for program tracing while meeting the program deadlines that is known as time-aware instrumentation [39, 63, 62, 5].

1.2 Time-Aware Instrumentation

Time-aware instrumentation tries to preserve logical correctness as well as meeting timing constraints. While a minor influence on execution time maybe be acceptable within a specified timing constraint, naive instrumentation will usually violate such constraints.

Time-aware instrumentation attempts to honor the timing constraints and shifts the execution time profile (ETP) of the program closer to the program's deadline.

Case studies investigated in related work [39] demonstrate the promise of the general concept but the results revealed new problems. Figure 1.1 shows the ETP of a case study reported in [39] on the one laptop per child (OLPC) keyboard controller. The figure shows the success of time-aware instrumentation in shifting the ETP of the instrumented program. While the shift in the ETP is visible, it is lower than expected. The expectation was a larger shift in the time profile towards longer execution times. Further investigation revealed that, in this example, about 25 percent of the paths share basic blocks with the worst-case path (WCP). This means that large portions of the program are unavailable for instrumentation, because instrumenting them could affect the worst-case execution time (WCET) and thus violate existing timing constraints.

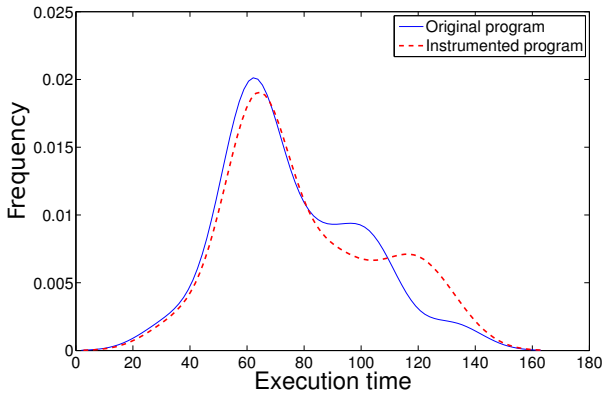


Figure 1.1: Execution time profiles of the OLPC keyboard controller [39]

1.2.1 Goals and Contributions

In this thesis, we investigate different techniques for increasing the effectiveness of time-aware instrumentation. The following is a summary of our contributions:

1. **Execution time profile shift-effectiveness metric (ETPsem) [63]:** We propose the ETPsem to measure the effectiveness of a time-aware instrumentation approach, so that different approaches can be compared against each other. One way to measure the effectiveness of time-aware instrumentation is a visual inspection as seen in Figure 1.1. This involves manual assessment and lacks accuracy. Another way is to calculate the instrumentation coverage as the ratio of extracted information to what is expected or desired. We introduce ETPsem as a more efficient metric for time-aware instrumentation.

2. **Program transformation [63]:** A central element for time-aware instrumentation is to identify regions in the program which can be instrumented. We propose an algorithm that identifies instrumentable edges in a program's control-flow. An instrumentable edge is one that lends itself for time-aware instrumentation. Different

approaches can use these edges for program instrumentation. We demonstrate the utility of this algorithm by introducing Branch Block Creation and CFG Cloning as two such approaches to increase the effectiveness of time-aware instrumentation at the expense of code size.

3. **Instrumentation on the WCP [68]:** We propose a slack-based conditional instrumentation technique for debugging hard real-time programs. This instrumentation technique preserves functional behavior, and temporal behaviors of the original program while allowing the instrumentation of variables on the WCP. Conditional instrumentation allows the instrumented code to execute only when there is sufficient slack in the program. We accomplish this using a timer to record the available slack during execution. This is a run-time approach to check for slack. Then, we address the challenge of selecting points in the program code to insert such conditional instrumentations. We use a purely software technique to implement conditional instrumentation and compare it against a technique that extends the processor with instructions to perform the conditional checks.

4. **INSTEP [62]:** Software systems are rich in extra-functional (or non-functional [132, 85]) requirements such as timing, code sizes, communication bandwidth, power consumption, and memory consumption. While our work focuses on time-aware instrumentation, other extra-functional properties might exist in the real-time system. Maintaining an extra-functional property during instrumentation is complicated and managing multiple properties simultaneously is even more so. Extra-functional properties can be competing where meeting one property might break another. We present a static instrumentation framework that gives the developer unprecedented control over what to instrument and what to preserve. It thereby presents the first fully-implemented instrumentation mechanism that considers multiple competing extra-functional properties. INSTEP uses trees to represent instrumentation intents and automata to represent cost models.

1.3 Chip-Multiprocessors for Real-Time Systems

Real-time embedded applications, and software in general, continue to evolve with increasing complexity. This increasing complexity of real-time applications translates into a need for higher computational power to satisfy their real-time requirements. Due to the limits on power consumption and heat dissipation in processors, increasing their performance is not a viable solution anymore. Multicore architectures pose themselves as a solution to the performance bottleneck. For the correct operation of the real-time system, a guarantee must be provided that the entire system is schedulable. This is done by ensuring that the temporal requirements of all tasks are met, which requires an estimate on the WCET of tasks and the worst-case communication latency between the tasks. Providing timing guarantees for hard real-time systems, however, on multicore interconnects is difficult. This is because conventional communication interconnects are designed to optimize the average-case performance. These, however, are a hindrance to accurately predicting the worst-case

communication time in the interconnects. Therefore, recent research focuses on developing custom network-on-chip (NoC) interconnects to facilitate their adoption as a platform for real-time applications.

Common NoC implementations include resource reservation and run-time arbitration (priority-aware) networks. Time division multiplexing (TDM) is an example of resource reservation networks. Resource reservation networks statically allocate resources to prevent contention between communication tasks in the network at runtime. Priority-aware networks, on the other hand, allow contention between communication tasks. Contention is resolved at runtime using priority-aware arbitration routers. TDM networks guarantee schedulability of tasks as part of their static allocation of resources and have small buffer requirements (one flit for guaranteed services and one packet for best-effort services) [44]. In this type of networks, if there is no data to be transmitted in a certain time slot, the slot will remain empty; thereby, not allowing other tasks to use it resulting in under-utilization of communication resources. For low latency communication, a larger proportion of slots are assigned; thus, causing low latency communication to be intertwined with the bandwidth when using TDM. Moreover, TDM networks do not cleanly support sporadic tasks. This is because sporadic tasks are triggered by external events and, hence, it is unknown at design time when exactly they start.

Priority-aware networks have better resource usage compared to TDM but require a worst-case latency (WCL) analysis to guarantee schedulability of the network tasks [126, 124]. A priority-aware network architecture was proposed by Shi and Burns [126, 124, 125, 127]. This architecture supports wormhole switching and priority arbitration between network messages. Shi and Burns also present a WCL analysis that we call flow-level analysis (FLA) that determines the worst-case communication latency in the priority-aware network [126, 124].

1.3.1 Goals and Contributions

The most recent WCL analysis for priority-aware networks is FLA [126, 124]. This analysis is crucial to determine the schedulability of communication tasks and accordingly the feasibility of deploying a real-time application on a priority-aware network. FLA assumes that the paths taken by communication tasks are indivisible units of communication. Hence, FLA does not incorporate the effects of pipelining and parallel transmission of data in the network. Aspects, other than the WCL analysis, need as well to be considered for the successful deployment of real-time applications on priority-aware networks. These include mapping of tasks to the network cores, path-selection for the communication between tasks, and buffer space assignment to the routers of the network. Our goal is to provide an integrated solution to deploying real-time applications on priority-aware networks while considering pipelining and parallel data transmission in the networks. The following is a summary of our contributions:

1. **Stage-level analysis (SLA) [64]:** We propose a pipelined communication resource model for analyzing the worst-case latencies for hard real-time systems. This model

supports pipelined and parallel transmission of data over communication resources with fixed priority preemption. We also present an associated stage-level analysis that uses the pipelined communication resource model to produce tight WCL estimates. The model supports communication tasks that are either periodic or sporadic. This analysis is suitable for interconnects that use run-time arbitration such as the priority-aware network proposed by Shi and Burns [126, 124].

2. **Offset-based worst-case response time (WCRT) analysis [65]:** An important challenge in distributing hard real-time systems onto modern computing platforms is in developing WCRT analysis techniques that combine communication and computation execution latencies. Such WCRT analysis techniques must consider the WCL of data transmissions across the communication medium connecting the processing resources, and its effect on any dependent computation tasks to determine accurate WCRT estimates. We present extensions to both FLA and SLA to compute end-to-end worst-case latencies of applications including both computation and communication tasks. We also present a holistic analysis algorithm for computing the dynamic offsets and jitters of tasks.

3. **Buffer space requirements [66]:** To enable the deployment of real-time applications to priority-aware networks, recent research proposes WCL analyses for such networks. Buffer space requirements in priority-aware networks, however, are seldom addressed. Hence, we bound the buffer space required for valid WCL analyses and consequently optimize router design for application specifications by computing the required buffer space in priority-aware routers. In addition to the obvious advantage of bounding buffer space while providing valid WCL bounds, buffer space reduction decreases chip area and saves energy in priority-aware networks. We present a detailed buffer space analysis using each WCL analysis: namely stage-level buffer-space analysis (SLBA) and flow-level buffer-space analysis (FLBA).

4. **Buffer space allocation in priority-aware routers:** Although computing the buffer space bounds required for valid WCL analyses is necessary, it only provides an upper bound to the required buffer space. Further reduction of the buffer space is possible, but might lead to higher worst-case latencies. These higher worst-case latencies are acceptable as long as the deployed applications are schedulable. We extend SLA to incorporate buffer space limits and we present an algorithm for buffer space allocation in the priority-aware routers.

5. **Path selection [67]:** We propose a path selection algorithm (PSA) assisted by SLA that aims to improve the number of schedulable tasks by selecting appropriate paths in the priority-aware network. We use SLA because it considers the pipelining effect of worm-hole switched NoCs, and it provides tight WCET bounds. In particular, we propose an algorithm that utilizes observations from SLA to efficiently select paths in the priority-aware network. To avoid the high complexity of an optimal algorithm, our algorithm uses heuristics to find least interference paths and to consider lower priority tasks while selecting paths for the higher priority ones.

1.4 Organization

This thesis is organized as follows. Chapter 2 presents an overview of the related work on the research topics addressed in this thesis. Chapter 3 presents ETPsem and techniques to increase the effectiveness of time-aware instrumentation (time-aware instrumentation contributions 1-3). Chapter 4 presents INSTEP, a static time-aware instrumentation framework (time-aware instrumentation contribution 4). Chapter 5 presents the stage-level WCL analysis (CMPs for real-time systems contribution 1). Chapter 6 presents the offset-based WCRT analyses (CMPs for real-time systems contribution 2). Chapter 7 discusses the buffer space requirements and buffer space allocation in priority-aware routers (CMPs for real-time systems contributions 3 and 4). Chapter 8 discusses the proposed PSA for real-time systems on priority-aware networks (CMPs for real-time systems contributions 5). Chapter 9 concludes this thesis.

Chapter 2

Overview

In this chapter, we discuss the related work on the research topics addressed in this thesis. First, we overview related work on information extraction techniques. Next, we overview related work on the different aspects of deploying real-time applications on CMPs.

2.1 Information Extraction

We mentioned in Chapter 1 that information extraction techniques include tracing, sampling, and emulation. We review the related work on these topics in this section as well as related work on multi-objective compilation.

2.1.1 Instrumentation

A program can be instrumented at the source code level either automatically or manually. In automatic instrumentation, a tool parses the program, may generate a control-flow graph (CFG), and eventually insert instrumentation points. An example of automatic instrumentation tools is the GNU compiler collection (GCC) profiling and code coverage instrumentation tool. On the other hand, in *manual instrumentation*, developers are in control, walk through the source code, and insert instrumentation statements whenever they see fit. Traditional debugging is a typical use case for manual instrumentation. Assume that the developer has received a bug report including a test case that causes the bug. Typical behavior is that the developer tries to identify the origin of the bug by manually inserting `printf()` statements at key points and rerunning the test case [69, 131]. By inserting more `printf()` statements and removing unnecessary ones, the developers test different bug hypotheses until they find the right one and can proceed with fixing the bug. Obviously, manual instrumentation, while being the most flexible, has the worst characteristics with respect to interference, because developers cannot estimate changes in the WCET [152] or memory consumption for modern computer architectures. This makes it also hard to find timing-related software defects using this method, because the instrumentation might temporarily clobber the defect.

Some instrumentation tools are capable of inserting instrumentation points to binary executables. This can happen either statically or dynamically during program execution. QPT is a program profiler and tracing system that measures the execution frequency of basic blocks and control flow [75]. EEL also provides similar functionalities for analysis and modification of executables for while abstracting details of instruction sets and executable file formats [74]. ATOM is another framework for building customized program analysis tools [134]. Binary static instrumentation tools also include Etch [119] the program performance evaluation and optimization system, and Morph [156] which re-optimizes programs to apply profile-based and other platform specific optimizations. Multiple tools exist that support dynamic binary instrumentation. Dynamic binary instrumentation tools that use code transformation during program execution include Dyninst [21], Kerninst [141], Detours [52], and Vulcan [36]. Most of these instrumentation tools, however, suffer from transparency issues, i.e., they modify native behavior of the program under analysis [20]. Examples of transparent dynamic binary instrumentation tools that have software code caches and dynamically compile binaries include Pin [88], DynamoRIO [19], and Valgrind [103]. Other examples of dynamic binary instrumentation also include DTrace, SystemTAP, Frysk and GDB. These tools overwrite code locations with trap instructions to execute instrumentation code, then after the interrupt handling and instrumentation code execution, the original instructions are restored and executed.

Some instrumentation tools are more oriented towards the analysis of parallel programs. Mellor-Crummey et al. propose a software instruction counter [92] to help debug parallel programs using the integrated approach to parallel program debugging and performance analysis on large-scale shared-memory multiprocessors introduced by Fowler et al. [41]. Thane [143] and Dodd et al. [32] present integrated approaches for monitoring and debugging of real-time systems. Kim et al. uses formal requirements for the monitoring and checking of Java programs at run time [70]. Moore et al. [96] and Omre [107] introduce hardware trace debuggers. Some other examples of hardware trace debuggers include the JTAG and NEXUS trace debugging interfaces. Cargill et al. use dedicated hardware counters to support the debugging and profiling of compiled programs [23]. There also exists work on replay mechanisms in hardware tracing. Replay mechanisms record sufficient information during program execution that later on helps the developer deterministically recreate an equivalent execution. A lot of research has been conducted on hardware supported replay mechanisms for non-deterministic applications [121, 120, 138, 148, 7, 50, 104, 95]. Some work also exists on tracing interrupts during program execution [137, 45]. Although such approaches naturally provide low interference, they can still have a significant impact on performance [97] and behavior and thus warrant research on characterizing interference levels and possible bounds.

All these instrumentation methods are known to affect the behavior of the program including its temporal behavior which is sometimes not acceptable in real-time embedded systems. Partial instrumentation, as a means of ensuring timeliness, can be used to build inductive debugging mechanisms for deployed resource and space constrained systems. Since it is hard to reproduce bugs from user bug reports [15], even having a partial trace can help extract information. Partial traces help extract information and can be input to additional debugging tools [123, 114].

Fischmeister and Lam [39] introduce time-aware instrumentation which honors the programs timing constraints, especially the worst-case behavior. The idea was mainly to instrument programs at code locations that leaves the WCET of the program unmodified. The proposed tool instruments programs, optimizing for code space and instrumentation coverage, while meeting time constraints.

2.1.2 Sampling

The goal of execution monitoring is to record an execution trace of the program under test. The external monitor observes the execution of the program and needs to log the program's execution path. In sampling-based tracing, the external monitor periodically examines the state of the program and stores it. It is clear that we can obtain more accurate traces through increasing the sampling frequency at the price of sampling overhead.

Liblit et al. present a sampling infrastructure with low overhead that collects samples from numerous runtime executions [84]. These samples are used for bug isolation in subsequent program executions. Liblit et al. also present a random sampling technique for statistical multiple bug isolation in complex applications [83]. Zheng et al. propose an interactive collective voting scheme for program runs to statistically identify multiple bugs in software [157]. Fischmeister et al. introduce an approach to bound the cost of monitoring program execution through fixed rate sampling [40]. This approach discusses optimal tradeoffs between accuracy and overhead. It also provides a framework for the reconstruction of system state and execution paths. Thomas et al. extends the work in [40] through investigating the expressiveness and efficiency of different marker schemes [145]. This work also proposes and discusses the applicability of two new marker schemes. Gprof, which is part of the GNU binary utilities, uses an operating system virtual timer to sample the program at regular intervals [46]. It can also produce a call graph through the instrumentation of each function entry in the program code. However, the effect of this kind of instrumentation on program execution cannot be estimated.

Another type of sampling involves the usage of performance counters. These counters are used to count instructions, cache misses, branch delays, and other similar metrics. OProfile is an example of a Linux-based performance analysis tool that uses performance counters sampling [81]. Cheung et al. propose Endoscope, a declarative aquisitional software monitoring framework, to monitor the state and performance of program execution [26]. Jiang et al. introduce metric-correlation models for bug detection in software systems and fault localization [58]. Ball et al. present algorithms that optimize the placement of profiling code according to the estimated and measured frequency of each basic block execution in a CFG [9]. Hutchins et al. [53] and Frankl et al. [42] propose branch-based monitoring techniques using definitive-use associations (DUAs). DUA is the association between variable definition (value stored in memory) and variable use (value fetched from memory) which are used for control-flow and data-flow coverage. Santelices et al. present a method to efficiently monitor DUAs based on branch monitoring, and to accurately predict some DUAs from branch coverage information [122].

2.1.3 Emulation

An emulated system can be totally controlled and analyzed during its execution. This increase in flexibility comes at the expense of performance (slower real-time execution). Examples of emulation systems are Unisim [6] and Qemu [11] which emulate different processor architectures. Another example is the Valgrind emulation system [103] which is used for program instrumentation, validation, and performance analysis. Valgrind is a framework for creating dynamic analysis tools for memory debugging, memory leak detection, and cache profiling. Valgrind has tools for the detection of memory and thread bugs, profiling of cache and branch prediction, heap profiling, and overrun detection in heaps, stacks, or arrays. Phillips also presents work on using an emulator with traces [113].

2.1.4 Multi-Objective Compilation

Naik and Palsberg [98] present a framework for code-size-aware compilation. They formulate register allocation as an Integer linear programming (ILP) problem. The objective is to minimize target code size under a set of linear constraints. Lee et al. [78] introduce a framework to balance the tradeoffs between code size, execution time, and energy consumption in a real-time embedded system. It minimizes a system's cost function while satisfying the design constraints. They enable a tradeoff between code size and execution time based on a dual instruction set processor. The framework satisfies the design constraints and assigns code/WCET pairs to the tasks to minimize the system cost function.

2.2 Chip-Multiprocessors

Buses are the most popular communication architecture between system-on-chip (SoC) components. Buses are widely used due to their flexibility and easy adoption. On the other hand, they are power inefficient and lack scalability [76]. Point-to-point connections offer dedicated communication channels between the SoC components. However, the required number of wires for the point-to-point scheme makes it unscalable [30]. Another architecture alternative is SoC interconnect architecture that model multiprocessor systems. NoC interconnects [30, 13] offer a reusable solution to the limitations of buses and point-to-point connections. In this section, we overview some aspects of the NoC interconnects.

2.2.1 Architecture and Operation

The NoC topology determines the layout of the network nodes as well as the connections between the nodes. Mesh [73] and torus [30] interconnects are amongst the most popular NoC topologies.

The way packets traverse the NoC routers is determined by the switching protocol. Circuit switching protocols establish dedicated connections between source and destination

pairs for packet traversal. Packet switching, on the other hand, does not reserve links. Examples of packet switching protocols include store and forward, virtual cut through, and wormhole switching. In store and forward [34], a router must buffer a complete packet before forwarding it to a neighboring router. This increases the transmission latency as well as buffer requirements. In virtual cut through [33], a packet does not have to be buffered and can be directly transmitted to another router. However, if the receiving router is busy, then the complete packet must be buffered. Wormhole switching [105] operates similarly but reduces the buffering requirement to the flit level of the packet. Virtual channels (VCs) [31] can be used to provide multiple buffer queues for the same physical channel. Wormhole switching along with using VCs increases the network utilization by reducing blockage in the NoC.

Routing algorithms define the paths that packets take in the NoC [34]. In deterministic routing, the path selection process is deterministic for any source and destination pair. In adaptive routing, on the other hand, the routes can change dynamically to, for instance, avoid congestion in the network. While adaptive routing increases the network efficiency, it adds a communication overhead and has more complicated logic compared to deterministic routing.

Flow control mechanisms are responsible for controlling the traversal of packets between the network routers and for the prevention of overflow in buffers. The handshake flow control involves the exchange of valid and acknowledgement signals between the sender and the receiver, respectively [155]. While handshaking is simple, it has a high overhead. In the ACK/NACK flow control scheme, the receiving router will respond with an acknowledgment (ACK) signal if it has enough buffer space to accept the incoming flit, otherwise it will reply with a negative acknowledgment (NACK) signal [115]. This scheme is combined with a GO-BACK-N policy in which the sender sends flits continuously without waiting for ACK signals until it receives a NACK. Upon the occasion of receiving a NACK, the sender resends the flit for which the NACK was received as well as all the flits that were sent after it. The ACK/NACK scheme is more efficient than handshaking, but requires expensive buffering for resending purposes. Credit-based flow control uses credits to avoid overflowing full buffers [116]. An upstream router has credits equivalent to the available buffer space at a downstream router. These credits are decremented as the upstream router sends flits downstream and are incremented as the downstream router frees buffer space.

2.2.2 Real-Time Communication

There are several research efforts that enable real-time communication over CMPs [44, 87, 128, 17]. Bjerregaard and Sparso [17] present a clock-less NoC called message passing asynchronous NoC (MANGO) for guaranteed services. They use an asynchronous latency guarantee arbiter, which consists of a set of VCs, and priority selection and arbitration modules to support real-time communication. MANGO combines wormhole switching with virtual circuits and provides guaranteed service in terms of bandwidth and latency [17]. The arbiter used by MANGO ensures that a flit in a virtual channel can block a lower priority flit only once. This is different from the fixed priority scheme used in this work. Wiklund

and Liu [151], and Wolkotte et al. [153] use circuit switching that requires establishing a connection between source and destination before sending data packets.

Millberg et al. [93] and Lu et al.[87] use the TDM approach for communication. TDM divides the link access into equal time slots such that a communication task can use the slot time to transfer its own packets. Æthereal [44] uses TDM to guarantee WCL bounds on real-time communication tasks. In each time slot, the router forwards the data from input to output ports. According to a pre-determined slot table, network adapters inject packets into the routers. Hence, TDM avoids contention between packets in the network, and has no need for arbitration and buffering of packets [133]. The WCL depends on the slot allocation [87], which computes the time for the last flit of a task to reach its destination. Æthereal, like MANGO, combines guaranteed service with best-effort to increase its resource utilization [118]. Hence, the Æthereal router design [44] supports both guaranteed service and best-effort traffic. Each router contains a slot table to control the switching of the guaranteed service traffic.

A light version of Æthereal, aelite [48], is available. It carries the routing information inside packets headers, and only supports guaranteed service traffic. This simplifies the routers by avoiding the use of slot tables. Another version of aelite is called dAElite [136]. It supports multi-cast routing by using slot tables.

In TDM NoCs, a global notion of time is required for the usage of the slot tables in the routers. This can be achieved using a fully-synchronous NoC implementation [133]. This, however, might be practically infeasible due to different operating frequencies on cores as well as difficulties with clock distribution. Alternative implementations of the NoC are, hence, needed. In [48], a mesochronous implementation of aelite is presented. This can be achieved by adding first in first out (FIFO) buffers on the links to compensate for phase differences between clocks.

Shi and Burns [126, 128] propose the use of priority-based routers with wormhole switching to support real-time communication. This approach supports multiple priorities, and their routers ensure that higher priority tasks can preempt lower priority ones.

2.2.3 Timing Analysis

The problem of the WCL computation for inter-process communication in real-time systems is addressed in [60, 111, 110]. Authors develop an upper bound on the delivery time of messages. The downsides of these methods are the overhead of the establishment and tear down of channels between source and destination pairs, as well as under utilization of the system's resources. They also store packets at intermediate nodes which leads to expensive buffer capacity for storing early arriving packets and queuing packets in order of arrival [60].

The two common approaches used in customizing interconnects for real-time applications use resource reservation or run-time arbitration. An example of a resource reservation approach is TDM [93, 44, 87]. This approach statically allocates slots to communication tasks such that no other task can use that slot other than the assigned one. In the event

that there is no data to be transmitted in that slot, the slot remains empty; thereby, not allowing other tasks to use it resulting in under-utilization of communication resources. For low latency communication, a larger proportion of slots are assigned; thus, causing low latency communication to be inter-twined with the bandwidth when using TDM. Note also that TDM does not cleanly support sporadic tasks. This is because sporadic tasks are triggered by external events and, hence, it is unknown at design time when exactly they start.

Run-time arbitration, on the other hand, arbitrates access to communication resources at run-time. Hence, contention is expected, and the WCL analysis accounts for these contentions. One such communication architecture was proposed by Shi and Burns [126, 128] that supports wormhole switching with priority-based arbiters that allow higher priority communication to preempt lower ones. Moreover, this approach overcomes the tight coupling of latency and bandwidth suffered by TDM and TDM-like approaches, and it allows for a variety of communication task types with its use of priorities. Shi and Burns also present a WCL analysis, which we call FLA [126, 128] that determines the WCLs between communicating tasks. Their analysis includes direct and indirect interferences from other communications. This analysis is central in determining the schedulability of the communication tasks. However, FLA assumes a model where the tasks are indivisible units of communication. As a consequence, FLA does not incorporate the effects of pipelining and parallel transmission of data in its communication model. This restriction in the model results in higher upper-bounds on the communication latencies.

There exists other techniques to compute end-to-end worst-case delays on networks. These include network calculus [154], an extension of network calculus for real-time systems called real-time network calculus [144], holistic analysis [146, 89, 108], and delay calculus [55, 56].

Network calculus [154] is a deterministic queuing theory that uses max/min-plus algebra to determine performance bounds on load and service of a network. It is based on analyzing dataflows (that can be computation or communication) to generate cumulative functions of events entering a node and departing a node. Real-time calculus [144] specializes network calculus for real-time embedded systems by characterizing dataflows as interval bound functions, and incorporating methods to model schedulers [112]. Traditional worst-case bound methods in network calculus assume blind multiplexing. This requires that no assumptions are made for the arbitration of multiple tasks traversing a server. It is known that tighter upper-bounds can be computed by providing insights of the multiplexing [16]. Real-time calculus enables this with models of its arbitration schedulers. Furthermore, real-time calculus uses both upper and lower bound curves on arrival and service to compute worst-case delays, which contribute to tighter worst-case delays estimates. However, a difficulty shared by these approaches is that of analyzing tasks with certain traversal patterns such as those that are in nested and non-nested tandem [16]. A nested tandem situation is one where the path that communication tasks take are either nested within the task under analysis or disjoint. This difficulty arises because these approaches convolve arrival/service curves of each node; hence, the pipelined behavior of the network is not considered. This results in double-counting of events (repeatedly accounting for events that have already caused interferences at an earlier node) in the dataflow yielding

pessimistic worst-case bounds. Our approach differs compared to these in that the analysis is not as general as network or real-time calculus. This allows us to incorporate details of the arbitration scheme, and the pipeline behavior of the network. This enables us to remove events that result in double-counting of interferences does by understanding that once a task has interfered, then its re-interference does not extend the worst-case latency; therefore, achieving a tighter WCL.

Holistic analysis [146, 89, 108] introduces a worst-case response time analysis [80, 109] for transactional task models by considering task offsets. These task offsets are combined with jitters to characterize the arrival pattern of tasks at each node. The offset represents the earliest a task can be released, and the jitter represents the worst-case. Using the offset and jitter of a task arriving on a particular node, holistic analysis computes the offset and jitter of the task leaving that node. This becomes the arrival pattern for tasks for the next stage. Holistic analysis iteratively computes these patterns across all nodes resulting in the end-to-end worst-case latency of a given task. The primary difference between holistic analysis and the proposed approach is that holistic analysis does not take into account relationships between different resources. Thus, it does not leverage the pipelined transmission of data, which the proposed approach does in order to deliver a tighter upper-bound.

Delay calculus [55, 56, 57] proposes a method to compute end-to-end worst-case delays experienced by jobs executed over multiple stages. Unlike holistic analysis, delay calculus incorporates pipelined behavior in its analysis. However, delay calculus requires that the task executes completely before proceeding to the next stage of the pipeline. In our proposed model, we do not have this restriction. The preemption model is also significantly different than the one of proposed. A job in delay calculus may be preempted midway during execution, and resumed later once the higher priority job completes. This is different from the model proposed in this work, where a data unit is either preempted before transmission or is successfully routed to the output port. Hence, a data unit is either preempted in its entirety or not. We find that out model respects router models where midway transmissions are usually not preempted. Additionally, delay calculus only partially considers tandem situations. We find that the proposed model can more accurately model the communication resources such as NoCs that support switching techniques that operate at the flit-level such as wormhole switching..

2.2.4 Buffer Space Requirements

Since NoC designs usually target specific applications (or application classes), multiple works investigate customizing NoC designs to optimize performance while limiting cost [12, 18]. Some of these works focus on limiting buffer space in NoCs. Hu and Marculescu [51] propose an algorithm for customizing buffer space in NoC routers at a system-level. Given a buffer space budget, they assign buffers to input channels to maximize performance. The proposed work, however, does not consider real-time requirements of tasks and does not ensure timeliness. Manolache et al. [90] propose a technique for changing the mapping of data packets to network links and the release timing of packets to reduce buffer sizes

and ensure timeliness. While the work does not consider wormhole switching (flit-level preemption) or VC resource allocation, it can be extended to apply to them. The authors use the WCRT analysis proposed by Palencia and Gonzalez [108] to compute the buffer space requirements. This WCRT analysis, however, assumes all tasks sharing resources (including indirectly interfering tasks) to be directly interfering with each other, this produces higher buffer space upper bounds. A similar result applies to WCL bounds as shown in [65]. Kumar et al. [72] present a simulation based algorithm for reducing buffer sizes while considering latency requirements. The authors use simulation to capture the contention between tasks on network resources which does not provide worst-case guarantees for real-time tasks. Al Faruque and Henkel [4] propose an approach for reducing VC buffers which also does not provide real-time guarantees.

Goosens et al. [44] propose the Æthereal TDM-based NoC. They provide a full implementation with buffers for handling best-effort and guaranteed services. Coenen et al. [27] present an algorithm to find the minimal buffer sizes required to decouple computation and communication in the TDM NoC using credit-based flow control while maintaining real-time guarantees to tasks. Similarly, in this thesis, we attempt to bound the buffer space requirements for priority-aware networks.

2.2.5 Path Selection

Several algorithms exist for path selection in a network. Among these are Shortest Path, widest shortest path (WSP), and shortest widest path (SWP), which are greedy approaches [47]. The WSP algorithm selects the shortest path with least interference (most residual capacity). The SWP selects the shortest amongst all low interference paths.

Another class of algorithms consider other tasks while selecting a path, but are more computationally expensive. Examples are the minimum interference routing algorithm (MIRA) [61], light minimum interference routing [38], and profile-based routing [139]. MIRA routes a task such that it does not create much interference with a route that may be critical to satisfy new tasks. Light minimum interference routing algorithm operates like MIRA, but with a reduced computation complexity. Profile-based routing uses a traffic profile of the network to predict requirements for tasks, and solves a multi-commodity network flow problem. Distributed routing algorithms also exist [129], which are online algorithms that either require a global state that leads to high communication overhead and performance degradation, or do not share a global state and compensate with a large number of control messages and subsequently do not scale.

There is research on path selection for worm-hole switched networks [79, 130, 59, 99]. Some of these methods attempt to find contention-free paths or minimize total cost, sometimes leading to higher ratios of unschedulability of tasks [99]. Others attempt to minimize the maximum contention value, which is similar to the techniques used in MIRA [61, 38]. However, all of these approaches consider the path of a task as an indivisible unit; thus, they require a computationally expensive enumeration of paths for path selection.

Chapter 3

Increasing the Effectiveness of Time-Aware Instrumentation

We mentioned in Chapter 1 that time-aware instrumentation of the OLPC case study revealed that large portions of the program are unavailable for instrumentation, because instrumenting them could affect the WCET and thus violate existing timing constraints. This raises the question of how to measure the effectiveness of a time-aware instrumentation approach, so that we can compare the different time-aware instrumentation approaches. Therefore, in this chapter, we propose ETPsem as a new metric for time-aware instrumentation.

A central element for time-aware instrumentation is to identify regions in the program which can be instrumented. We propose an algorithm that identifies instrumentable edges in a program's control-flow. An instrumentable edge is one that lends itself for time-aware instrumentation. Different approaches can use these edges for program instrumentation. We demonstrate the utility of this algorithm by introducing Branch Block Creation and CFG Cloning as two such approaches to increase the effectiveness of time-aware instrumentation at the expense of code size.

We also propose a slack-based conditional instrumentation technique for debugging hard real-time programs. This instrumentation technique preserves functional behavior, and temporal constraints of the original program while allowing the instrumentation of variables on the WCP. We investigate methods to select instrumentation points so as to meet certain constraints. We present hardware and software implementations of the slack-based conditional instrumentation technique.

3.1 Overview of Time-Aware Instrumentation

Time-aware instrumentation aims to instrument a program while minimizing changes to the timing behavior on the WCP. Since the execution time of the program differs from one execution path to another, the idea is to instrument programs only in locations that

have minimal impact on the program's WCET. In the optimal case, this means adding zero overhead to the program on the WCP.

The work flow for time-aware instrumentation differs from standard instrumentation in that it has extra steps to consider timing. Figure 3.1 shows the work flow of our approach. Initially, we analyze the program's source code, establish timing information, and generate its CFG. The instrumentation tool analyzes the program and considers timing information. The instrumentation tool instruments the program based on a time-aware instrumentation technique such as the one proposed in previous work [39] or our proposed slack-based conditional instrumentation. The tool outputs an instrumentation configuration which the framework uses to compute an expected instrumentation coverage for a given set of variables. After running the tool, the developer checks whether the achieved expected coverage is acceptable. If the results are satisfactory, the developer will execute the instrumented program. Otherwise, if the expected coverage is insufficient, then the developer will have two means by which to attempt increasing the coverage. First, the developer can use our approaches to transform the program into a program that is more suitable for instrumentation. Second, the developer can change the debugging budget given to the instrumentation. This increases the number of instrumentation points. If neither of these two is successful, then the framework will report that it is unable to instrument and meet the desired coverage.

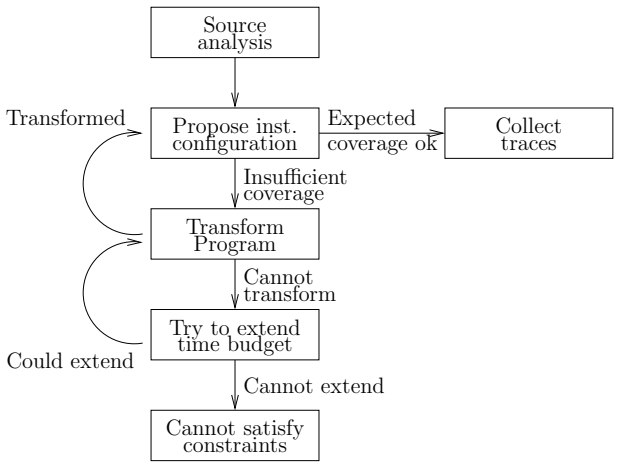


Figure 3.1: Work flow for time-aware instrumentation

The instrumentation process uses time differences between execution paths and different basic blocks to ensure that the instrumented program stays within the original program's time limits on the WCP. Sometimes, however, due to processor anomalies, cache behavior, etc., the timing might change. After each complete instrumentation attempt, the framework will analyze the WCET behavior to check if it exceeds the execution time of the original WCP plus the debugging budget. If this occurs, the framework will attempt a different instrumentation configuration which reduces the coverage on non-WCPs to satisfy the timing requirements. This process will be iteratively repeated until the program meets its timing constraints.

3.2 Model and Terminology

Developers usually use instrumentation to trace information of interest such as the state changes of variables. To, for example, trace changes of a variable x, instrumentation must be applied to all places in the program that modify the variable x and add code to record its value. Time-aware instrumentation will choose which ones to instrument based on the timing constraints of the program.

We extend the abstract model of programs presented by Fischmeister and Lam [39]. The abstract model represents the source program as an extended CFG. A basic block is a portion of source code of the program with one entry point and one exit point. We augment the definition of a vertex with an associated type as shown in Definition 1. Using this definition of a vertex, we define the extended CFG as shown in Definition 2. We call this abstract model a one-state change CFG (OSCCFG).

Definition 1 (Vertex)**.** *A vertex is a basic block with at most one assignment to the same variable. We represent a vertex as a tuple $v = (i, t)$ where $i \in \mathbb{N}$ is a unique identifier, and $t \in \{None, IP\}$ is an instrumentation type associated with the vertex.*

The unique identifier allows us to distinguish and reference vertices. A vertex with the *None* instrumentation type is the default for all vertices. *IP* indicates that an IP is added to that vertex. Only one IP can be added to a vertex. Note that a vertex is a basic block of the program with the additional requirement that each basic block modifies any variable at most once within it. A traditional CFG contains vertices with multiple state changes to the same variables via assignments within a vertex. We split such vertices into multiple vertices with only one state change to the same variable in each vertex, and construct edges between them.

Definition 2 (One state-change CFG)**.** *An OSCCFG is a directed graph $G := \langle V, E, v_s, v_x \rangle$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges that represent flow of control, and $v_s, v_x \in V$ are unique start and exit vertices, respectively.*

A path of an OSCCFG G describes a traversal of the graph as shown in Definition 3. We denote the set of all paths from v_s to v_x as $\mathbb{P}_{v_s,v_x}$, and the WCP as the path with the largest WCET estimate. To extract the sequence of vertices from a path p_{v_s,v_x} we employ the helper function $vertices : p_{v_s,v_x} \to V^{[]}$. Notice that we superscript domains with [] to denote a sequence and {} for a set.

Definition 3 (Path)**.** *A path p_{v_s,v_d} from source vertex v_s to destination v_d in an OSCCFG G is a sequence of vertices $\langle v_s = v_1, v_2, \ldots, v_{n-1}, v_n = v_d \rangle$ with $n \in \mathbb{N}$ being the number of vertices forming the path.*

Figure 3.2 shows an example ETP of a program. The WCET of a program is an upper-bound on the execution time of any path in the program. The difference between the WCET and the actual execution time of any program instance is commonly called slack. Note that the actual execution time is a run-time characteristic. The static time

window, α, is the difference between the program's WCET and deadline such that $\alpha = Deadline - WCET$. For safety concerns, systems are designed with a a window α that is sufficient to act as a safety assurance margin. In addition to the differences between lower execution times and the WCET, instrumentation can also make use of the debugging time budget available to a program. A program's debugging time budget β is usually a small percentage of the static time window α. The debugging budget β is a percentage of the CPU resources dedicated to debugging and must be accounted for in the schedulability analysis. Therefore, we can utilize the debugging budget for instrumentation in the manner described in Section 3.1.

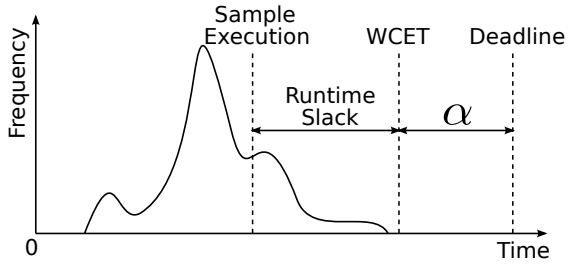


Figure 3.2: Example of a program's execution time variation

The instrumentation process might increase the WCET of a program by an overhead O. Perturbations in the WCET are acceptable as long as they are less than or equal to the debugging budget β, specified by the developer. Depending on the extent to which the system is loaded, the debugging budget may permit programs to absorb small increases in the WCET, and still ensure that the temporal deadlines are correct. So for single-task applications without interrupts, we must ensure that the overhead O is below the debugging budget β. For instrumenting concurrent applications, we must ensure the schedulability of the whole workload after adding all overheads to the WCETs of the instrumented program functions. The specific way to distribute the available debugging budget among the tasks is up to the developer. However, a naive way to distribute the budget would be to assign weights to tasks (for instance following the tasks importance) and distribute the budget according to these weights.

3.3 ETP Shift-Effectiveness Metric

Related work [39] used a coverage (also named reliability) criterion as a metric for the quality of instrumentation. The instrumentation coverage of an instrumented program is the ratio of the amount of information extracted at run time to the desired amount. So, for example, when the developer wants to trace 100 variable assignments and the instrumentation only yields 30 assignments, then the instrumentation coverage will be 0.3.

This metric fails to capture the potential for extracting information at an abstract level, because it only compares concrete solutions. For example, to compare two instrumentation techniques using instrumentation coverage, this metric measures the values for the different techniques for a particular execution of the program. Instead, a more useful metric can

capture the quality of the different techniques over a wide range of inputs and, therefore, for different program executions. This metric should potentially estimate the shift in the program's ETP per unit coverage for each instrumentation technique.

This section presents a new metric for time-aware instrumentation [63]. The metric complements the previously explored coverage metric. ETPsem captures the potential for instrumentation and thus defines the optimal bound on time-aware instrumentation for any function based on its ETP. Figure 3.3a shows such ETPs for a fictive function. Values on the x-axis show the different execution times of the program and the y-axis shows the frequency at which the execution time occurs when executing this function.

Time-aware instrumentation bases on the idea of a *"right shift"* in the ETP during instrumentation. The ETPs of the OLPC case study demonstrate this right shift in Figure 1.1. The ETP of the instrumented program exhibits a right shift from the uninstrumented program's ETP. The reason is that instrumentation utilizes paths with lower execution times (compared to the WCET). Instrumenting these paths increases their execution times and thus shifts the ETP to the right.

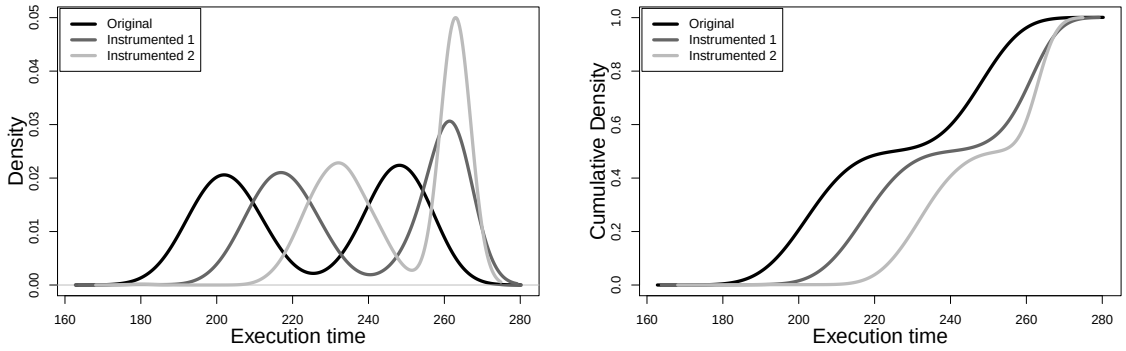
ETPsem uses this observation to quantify the theoretic optimum for time-aware instrumentation. The insight is that any software-based instrumentation inserts code in the programs and thus shifts the ETP. For the same coverage, the less the shift in the ETP, the more effectively the method has used the slack (disregarding any execution time anomalies that might exist). ETPsem uses time/coverage as its basis with a double integral over time.

Figure 3.3 illustrates the ETPsem. Figure 3.3a shows the ETPs of the original and different instrumented programs. Figure 3.3b shows the cumulative distribution function for Figure 3.3a. The more the shift in the ETP of the instrumented program, the further its cumulative distribution from that of the uninstrumented program. From a slack utilization standpoint, an optimal instrumentation is one where all paths in the profile become fully utilized for instrumentation and thus exhibit the same execution time as the worst-case path. From a coverage perspective, an instrumentation technique is more effective when it extracts more information and, thus, has higher coverage. Integrating the cumulative distribution curves once more and dividing the values by the instrumentation coverage obtains the effectiveness of the instrumentation. As the effectiveness of the instrumentation increases, the value of the ETPsem metric decreases because of higher coverage and better slack utilization (smaller area under the cumulative distribution curves).

The *theoretical* optimal value on ETPsem is zero (see Section 3.3.1 for details). If after an instrumentation, ETPsem results in 0, this means that no more instrumentation is possible. Note that it might be the case that an instrumentation technique can extract all data of interest without fully utilizing the slack on each path. In that case, ETPsem's value will be greater than zero.

ETPsem is a measure of how successful an instrumentation technique is in consuming slack in the program (between lower execution times of non-WCPs and the WCET) for the sake of tracing a certain amount of information. In general, the smaller value, the more efficient the instrumentation technique is as it has better slack utilization per unit coverage. Note, however, that for the same instrumentation coverage, a higher ETPsem

value might be viewed as as a more effective instrumentation. For example, assume two instrumentation methods applied to the same program result in the same coverage but different ETPsem values. The one with a higher value means less slack utilization, i.e., a more efficient utilization of time, to extract the same information.



(a) Sample ETPs (b) Cumulative distributions

Figure 3.3: Illustrative example of the ETPsem

3.3.1 Formalized ETPsem

So far, we informally introduced ETPsem. Here, we provide the optimality criterion and prove that the optimal value is zero.

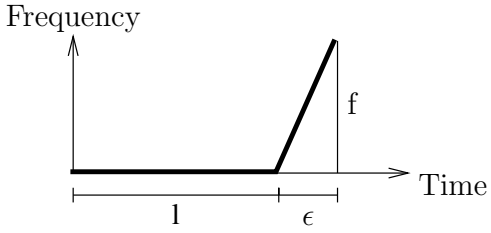


Figure 3.4: Execution time profile to prove Theorem 1

Figure 3.4 shows an ETP which we will use for our proof. We use a piecewise continuous function by first setting the initial value of the function to zero and then creating a linear increase after time l. As explained before, the more the shift in the ETP after instrumentation, the better the score on ETPsem must be. In Figure 3.4, ϵ is the critical element in this setup as it controls how much leeway (or slack) there is left for the right shift. If ϵ becomes zero, then there will be no more room left for any instrumentation and ETPsem has achieved its maximum and thus optimal value.

Definition 4 (ETPsem)**.** *ETPsem is defined by the function*

$$\Upsilon(t) := \frac{1}{\varrho} \iint \mathtt{ETP}(t)\, d^2t$$

where ϱ is the instrumentation coverage and ETP is the execution time profile of the instrumented program.

Definition 4 shows the formalized ETPsem metric. Note that ETPsem, $\Upsilon(t)$, is only of interest at the deadline d. This is because we are interested in the utilization of slack in the program for instrumentation up to the deadline d.

Theorem 1 (Optimality of ETPsem)**.** *Given a program function to be instrumented and a deadline d, the optimal value for $\Upsilon(d)$ is* 0.

We demonstrate that Theorem 1 holds by computing ETPsem for an ETP as shown in Figure 3.4 and argue that as we set ϵ to 0, the resulting value of ETPsem becomes 0. Our selected ETP is representative for any possible ETP. The following proof is sufficient to show that for any ETP a lower value of the metric indicates a more effective instrumentation and that 0 is the theoretical optimum.

Proof. Let Figure 3.4 describe our ETP for a function with a deadline of d with:

$$ETP(t) = \begin{cases} 0 & \text{if } t \leq l, \\ \frac{f}{\epsilon}(t-l) & \text{if } t > l. \end{cases}$$

$ETP(t)$ is a density function, thus $\int ETP(t)\,dt = 1$. Since $\frac{\epsilon f}{2} = 1$, we can substitute f in $ETP(t)$ with $f = \frac{2}{\epsilon}$ and get:

$$ETP(t) = \begin{cases} 0 & \text{if } t \leq l, \\ \frac{2}{\epsilon^2}(t-l) & \text{if } t > l. \end{cases}$$

The lead time to the shortest path is irrelevant, because the function is zero and so does not contribute to the metric as $\int_0^l \int ETP(t)\,d^2t = 0$. Thus, we only consider $t \geq l$ and, therefore, can use a shifted x-coordinate $x = t - l$ as follows:

$$F(x) = \frac{2}{\epsilon^2}(x)$$

We compute ETPsem as $\Upsilon(x) = G(x)/\varrho$, so we integrate $F(x)$ and receive:

$$G(x) = \iint F(x)\,d^2x = 1/3\,\frac{x^3}{\epsilon^2} + c$$

The length of ϵ controls the shape of the ETP. Given that $F(0) = 0$ and $G(0) = 0$, we substitute x with the limits of the integration 0 and ϵ in G and receive:

$$G(x)\Big|_0^\epsilon = (1/3\,\epsilon)$$

Obviously, the steeper the increase (and thus the shorter ϵ), the less time remaining for instrumentation and the closer the results are to optimality. Since:

$$\lim_{\epsilon \to 0} G(x)\Big|_0^{\epsilon} = \lim_{\epsilon \to 0} 1/3\,\epsilon = 0$$

we show that as less time becomes available for instrumentation, the value of ETPsem approaches 0.

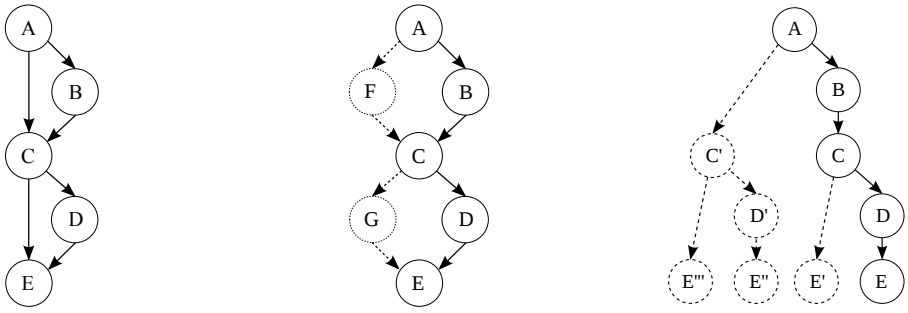
As ϵ approaches 0, $ETP(t)$ becomes an impulse. Any ETP will become an impulse as all paths have the same execution time. Therefore, our chosen ETP is representative for any ETP. □

3.4 Program Transformation

In this section, we investigate program transformation as a means of increasing the effectiveness of time-aware instrumentation [63].

3.4.1 Edge Detection for Program Transformation

In a program's CFG, uninstrumentable non-WCP edges are ones that lie on non-WCPs and connect to basic blocks of the WCP. They are uninstrumentable because instrumenting any of the basic blocks to which they connect changes the WCET of the program. These uninstrumentable non-WCP edges limit the shift in the ETP and are the basic elements used by methods that increase the coverage and ETPsem. Figure 3.5a shows a sample CFG with five basic blocks A, B, C, D and E. Assuming that $\langle A, B, C, D, E\rangle$ is the WCP, then the uninstrumentable non-WCP edge set includes $\langle A, C\rangle$ and $\langle C, E\rangle$, because these edges, although being non worst-case edges, they share all their basic blocks with the WCP.



(a) Original OSCCFG (b) Branch Block Creation (c) CFG Cloning

Figure 3.5: Example of program transformation

We propose an algorithm to identify such edges as the basic building block for improving time-aware instrumentation. Function 1 shows this algorithm. The algorithm takes as input the OSCCFG $G\langle V, E\rangle$ and the WCP p_{v_s,v_d} of the OSCCFG, and it returns the set of

edges of interest. The helper function $children : v \rightarrow V^{\{\}}$ returns the set of direct successor vertices of a vertex v. The queue operations *enqueue* and *dequeue* enqueue an element into and dequeue an element from a queue, respectively.

This algorithm finds non-WCP edges that have subpaths of the WCP connecting their head and tail vertices. In Function 1, lines 5-6 iterate through all edges of the directed OSCCFG G that have both vertices on the WCP p_{v_s,v_d}. For each edge e, line 7 queues the direct successors of the head vertex of the edge in the queue Q except for the tail of the edge e and line 8 marks the vertex as *visited*. Then, line 9 iterates on all vertices in the queue Q. Line 10 dequeues a vertex v from Q and lines 11-12 expand the direct successors of v and mark them as *visited*. Line 13 checks each of the direct successors of v, if the direct successor is the tail of the edge e, then line 14 will add edge e to set B. Otherwise, line 17 would enqueue the direct successor in Q, if it was not visited before. This algorithm is polynomial in time with respect to the number of vertices.

Function 1 Edge Identification

Input: OSCCFG G, p_{v_s,v_d}
Output: $E^{\{\}}$

```
1: Let S be the set of visited vertices
2: Let B ⇐ ∅ be the set of edges for creating basic blocks
3: Let Q be an empty queue
4:
5: for e = (v_a, v_b) ∈ E do
6:    if v_a ∈ p_{v_s,v_d} and v_b ∈ p_{v_s,v_d} then
7:       Q ⇐ children(v_a) \ v_b
8:       S ⇐ {v_a}
9:       while Q ≠ ∅ do
10:         v_x ⇐ dequeue(Q)
11:         S ⇐ S ∪ {v_x}
12:         for v ∈ children(v_x) do
13:            if v = v_b then
14:               B ⇐ B ∪ {e}
15:               break while loop
16:            else if v ∉ S then
17:               enqueue(Q, v)
18:            end if
19:         end for
20:      end while
21:   end if
22: end for
23: return B
```

3.4.2 Branch Block Creation

Branch Block Creation is a program transformation technique that uses the edge detection mechanism described in Section 3.4.1. Branch Block Creation creates locations in the

program for instrumentation. This increases the number of instrumentable basic blocks in the program.

Overview

Figure 3.5 illustrates how Branch Block Creation transforms programs to increase the code locations available for time-aware instrumentation. Figure 3.5a shows a sample OSCCFG with five basic blocks A, B, C, D, E. Assuming that $\langle A, B, C, D, E\rangle$ is the WCP, $\langle A, C, E\rangle$, $\langle A, C, D, E\rangle$, and $\langle A, B, C, E\rangle$ are non-WCPs that cannot be instrumented because they share all their basic blocks with the WCP. Applying Branch Block Creation modifies the OSCCFG as shown in Figure 3.5b. We create two new basic blocks F and G on edges $\langle A, C\rangle$ and $\langle C, E\rangle$ (detected by Function 1), respectively. This creates new non-WCPs with basic blocks F and G that can be used for instrumentation given that the execution time of these paths stays less than or equal to that of the WCP $\langle A, B, C, D, E\rangle$.

The Branch Block Creation algorithm may modify the program's WCP, depending on the target architecture and compiler. After instrumentation, the end of basic block B (which used to be a fall-through block) now contains a new unconditional branch instruction to jump past the code at F. This instruction did not exist in the original program. Therefore, the creation of basic block F results in an overhead of one unconditional branch instruction on the WCP $\langle A, B, C, D, E\rangle$ (same happens due to G). Note that the location of the unconditional branch instruction, whether in the *if* or the *else* block, is architecture specific. If the instruction is in the *if* block then it will modify the WCP; otherwise it will not. Note also that even if the compiler adds the instruction to the *if* block, inverting the condition will move the instruction to the non-WCP (the instrumentation block) leading to an unmodified WCP. Therefore, modifying the WCP is avoidable. But since the avoidance is either architecture specific or requires code modification, this work assumes that the Branch Block Creation modifies the WCP.

Algorithm

The algorithm for Branch Block Creation iterates on the set of edges obtained as output from Function 1 and creates basic blocks on these edges. We use these created basic blocks for instrumentation either by instrumenting every basic block for modified variables or through the minimization of instrumentation points [39].

Branch Block Creation may add overhead on the WCP and this overhead must stay below the program's debugging budget β (assuming that Branch Block Creation adds instructions on the WCP). This may lead to an increase in the WCET of the program. Thus, We want to choose only a subset of the basic blocks to create such that the overhead O is within the given budget β for instrumentation. Equation 3.1 describes this optimization

problem:

$$\text{Max} \sum_{i=1}^{n} b_i * (vars_i * frequency_i)$$
$$\text{subject to} \sum_{i=1}^{n} b_i * (overhead_i * frequency_i) \leq \beta$$
$$\text{where } b_i \in \{0, 1\} \text{ for } i = 1, 2, \ldots, n. \tag{3.1}$$

$vars_i$ is the number of traced variables at the created basic block i, $frequency_i$ is the number of times the basic block i executes (determined by the WCET analysis tool), and $overhead_i$ is the overhead on the WCP caused by creating the basic block i. n is the total number of created basic blocks from Function 1, and b_i is the binary variable.

Solving the problem of finding a subset of the basic blocks to create is non-deterministic polynomial time (NP)-Complete. We can show this by first polynomially reducing the binary knapsack problem to this problem, thus proving that its NP-Hard, and then showing that the problem lies in NP. The knapsack problem has n items as input. The ith item has a value u_i and a weight w_i. The solution is a subset of items, with a total value $\sum u_i \geq U$ where U is a target value, that can be placed in the knapsack such that their total weight $\sum w_i \leq W$ where W is a maximum weight. In the decision version of our problem, we have n basic blocks and each basic block i has a value $u_i = vars_i * frequency_i$ corresponding to the total number of traced assignments and a weight $w_i = overhead_i * frequency_i$ corresponding to the overhead it adds. The solution in our case is a subset of the basic blocks with $\sum u_i \geq U$ that we can create such that their total overhead $\sum w_i \leq \beta$. Hence, a polynomial reduction is straight-forward from that point. We map the items to the basic blocks, the value of each item to the total number of traced assignments, and the weight of each item to the overhead of creating a basic block. The maximum weight the knapsack can carry maps to the maximum overhead that the WCP can tolerate which is β. Now, to prove that our problem $\in$ NP, we show that it is verifiable in polynomial time. Given a subset of basic blocks, we can in linear time compute the total overhead $\sum w_i$, and in linear time compute the sum of values $\sum u_i$, then in constant time check if $\sum u_i \geq U$ and $\sum w_i \leq \beta$. Therefore, our problem is NP-Complete. We solve the problem using binary integer programming (BIP).

The set of basic blocks to be created on the edges returned from Function 1 is given as input to this optimization problem. The output is the set of basic blocks to actually create and use for instrumentation. We use this subset of created basic blocks for instrumentation to satisfy the program's debugging budget β.

3.4.3 CFG Cloning

In this section, we propose CFG Cloning as another transformation technique that uses the edge detection algorithm outlined in Section 3.4.1. CFG Cloning facilitates instrumentation on non-WCPs that share basic blocks with the WCP. CFG Cloning does not add instructions to the WCP and offers more instrumentation flexibility at the expense of code size.

Overview

We illustrate the concept of CFG Cloning using the example OSCCFG in Figure 3.5a. Again, we assume that $\langle A, B, C, D, E\rangle$ is the WCP. Although the OSCCFG contains three other non-WCPs, we cannot instrument them, because they share all their basic blocks with the WCP.

CFG Cloning duplicates whole subgraphs of the OSCCFG to permit instrumenting them. Figure 3.5c shows the OSCCFG after we do CFG Cloning. First, for the edge $\langle A, C\rangle$, which does not fall on the WCP, we duplicate the basic block C and its subgraph. Edge $\langle C, E\rangle$ as well does not belong to the WCP and so we duplicate basic block E. It is worth noting that the edge $\langle C, E\rangle$ has been duplicated before (in the subgraph of C), and, therefore, we duplicate it twice, once for each occurrence. We choose to duplicate cloned occurrences because they represent different execution paths in the program and, hence, increase the number of locations at which instrumentation can be inserted. Now, each of the three non-WCPs that used to share basic blocks with the WCP, have their own paths with some unshared basic blocks.

Algorithm

The algorithm for CFG Cloning iterates on the set of edges obtained as output from Function 1 and copies the tail basic blocks of these edges along with their subgraphs. The algorithm removes each of these edges and creates new edges from the head basic blocks to the copied subgraphs. Function 2 implements the algorithm for CFG Cloning. The algorithm takes as input the OSCCFG $G\langle V, E\rangle$ and the WCP p_{v_s,v_d}. The output is a new OSCCFG G'. The helper functions *children*, *enqueue* and *dequeue* are as defined in Section 3.4.1. The function $copy : v \rightarrow v'$ copies a vertex v. Function $original : v' \rightarrow v$ returns the original vertex v from which v' was copied. Functions *addEdge: G,e* and *removeEdge: G,e* add and remove an edge e to/from a graph G, respectively. Function $create : v \rightarrow G$ creates new graph G with a head node v and function $addChild : G, v, v'$ adds v' as a child to vertex v in graph G.

In Function 2, line 7 iterates through all edges returned from a call to Function 1. For each edge e, lines 8-9 copy the tail vertex v_b of the edge and create a new graph S. Lines 11-26 then iterate through the children vertices of v_b until the algorithm copies the subgraph of v_b to S. After that lines 27-28 remove the edge e and create an edge from v_a the head vertex of e to the head of the new graph S. Finally, the instrumentation algorithm can instrument the new OSCCFG G'. This algorithm is exponential in time with respect to the number of vertices in the OSCCFG G.

One drawback of CFG Cloning is the potentially large increase in code size. In general, this increase is exponential, and Figure 3.5c already indicates this. This is because when a basic block is duplicated all its subgraph is duplicated as well. Moreover, we duplicate basic blocks as well as any copies of them that are created from the duplication of any ancestor basic block. Although we limit the CFG Cloning to the scope of functions and loops, the exponential duplication can still cause problems.

Function 2 CFG Cloning

Input: OSCCFG G, p_{v_s,v_d}
Output: OSCCFG G'

Let $B \Leftarrow \emptyset$ be the set of edges for creating basic blocks
Let $S \Leftarrow \emptyset$ be a graph
Let Q be an empty queue

$G' \Leftarrow copy(G)$
$B \Leftarrow EdgeIdentification(G', p_{v_s,v_d})$
for $e = (v_a, v_b) \in B$ **do**
    $v'_b \Leftarrow copy(v_b)$
    $S \Leftarrow create(v'_b)$
    $enqueue(Q, v'_b)$
    while $Q \neq \emptyset$ **do**
        $v' \Leftarrow dequeue(Q)$
        $v \Leftarrow original(v')$
        for $c \in children(v)$ **do**
            $c' \Leftarrow copy(c)$
            if $(v, c) \in B$ **then**
                $B \Leftarrow B \cup (v', c')$
            end if
            if $c' \notin S$ **then**
                $addChild(S, v', c')$
                $enqueue(Q, c')$
            else
                $addEdge(S, (v', c'))$
            end if
        end for
    end while
    $removeEdge(G', e)$
    $addChild(G', v_a, v'_b)$
end for
return G'

Given that there is a certain limit on the increase in code size, we want to choose only a subset of the basic blocks that the algorithm duplicates while maximizing the amount of traced information. We model this problem as a non-linear programming (NLP) problem as shown in Equation 3.2.

$$\begin{aligned} &\text{Max} \sum_{i=1}^{n} b_i * vars_i * frequency_i \\ &\text{subject to} \sum_{i=1}^{n} b_i * code_i \leq code_{max} \\ &\text{where } b_i = \prod_{j \in S_i} b_j \quad \text{where } b_i \in \{0,1\} \text{ for } i = 1, 2, \ldots, n \end{aligned} \tag{3.2}$$

Here, b_i is a binary variable designating a duplicated basic block i, S_i is the set of duplicated basic blocks upon which the existence of basic block i depends, $vars_i$ is the set of traced variables at basic block i, $frequency_i$ is the number of times basic block i executes, and $code_i$ is the amount of code added by duplicating basic block i. n is the total number of basic blocks available for duplication. For example, for the OSCCFG in Figure 3.5c, we have three possible duplications C', E', and E''' (from E''). The code added by C' equals the total size of basic blocks C, D, E, while the code added by E', E''' is equal to the size of basic block E only. The existence of E''' would be valid only if C' was chosen for duplication.

After obtaining the new OSCCFG G' from the CFG Cloning algorithm, the duplicated basic blocks are passed as input to the optimization problem in Equation 3.2. The output is only a subset of all duplicate basic blocks. The set of duplicate blocks that have not been selected will be removed from the CFG G' and their subgraphs. The edges that were removed for creating these vertices will have to be reconstructed. This can easily be done because the edges are already stored in the set B in Function 1.

3.4.4 Experimentation

We explore the two transformation methods in practice using the SNU real-time benchmark suite [3]. This benchmark suite contains 17 C programs that implement numeric and digital signal processing (DSP) algorithms. The benchmarks have on average 117 lines of code and 34 basic blocks. We extended the benchmarks with a wide range of inputs to generate reasonable ETPs.

We apply the program transformation techniques to all benchmarks before instrumenting them. For the instrumentation of the programs, we use the technique proposed by Fischmeister et al. [39]. We compare the instrumentation of the transformed benchmarks against the instrumentation without transformation. Note that the transformation and instrumentation process is fully automated. We use CIL [101] for static code analysis and CFG extraction.

All experiments were run on a Keil MCB1700 board running an NXP LPC1768 microcontroller unit (MCU) which is a 100 MHz ARM Cortex-M3 microcontroller. This 32-bit

microcontroller has an MPU, 512kB on-chip Flash ROM, 64kB RAM, a nested vectored interrupt controller, and an eight channel general purpose DMA controller. The board has a 10/100 Ethernet Port, a USB 2.0 full-speed Device controller, two CAN interfaces, two serial ports, an SD/MMC card interface, a 5-position Joystick and push-button, an amplifier and speaker, up to 70 general purpose input-output pins, a 20-pin JTAG, a 10-pin Cortex debug connector, and a 20-pin Cortex debug and ETM trace connector.

Trace data from each benchmark was logged in a buffer and sent off-chip to a PC monitor for analysis. Note that a task sends data off chip at the end of a super loop as in a cyclic executive system. We use RapiTime [2] to analyze the WCET of the programs. We trace all variable assignments except for function arguments, constants, and loop counters. We set a debugging budget β for each program that is 2% of its WCET.

The goal of experimentation is to quantitatively assess the transformation techniques using the following metrics:

- **ETPsem:** This metric indicates the extent by which an instrumentation method utilizes time for instrumentation coverage.
- **Average instrumentation coverage:** Instrumentation coverage shows the effectiveness of an instrumentation method in capturing variable assignments. For each benchmark, we calculate the coverage for every input and compute the average across all inputs.
- **Instrumentation time:** This metric shows the time the tool spends in parsing the code, instrumentation, optimizing for debugging budget β, and any retries required. The instrumentation tool will retry the instrumentation with reduced overhead, if the WCET overhead after instrumentation exceeds the debugging budget β.
- **Increase in code size:** Every instrumentation point adds extra code to the program. The less the increase in code size, the more effective the instrumentation approach is in utilizing code space for instrumentation.
- **Number of retries:** It shows how often the instrumentation tool reduces the instrumentation coverage due to exceeding the debugging budget β. A small number of retries is essential for the applicability of the approach.

Results

Figure 3.6 shows the average instrumentation coverage for the different instrumentation approaches. The error bars show the maximum and minimum coverage over all executions of each benchmark. For the benchmarks: *fft1k*, *fibcall*, *insertsort*, *jfdctint*, and *matmul*, none of the instrumentation approaches was able to extract any information. This happens, because either the program has a single path which is the WCP that cannot be instrumented, or the program has multiple paths but adding any instrumentation code modifies the program's WCP.

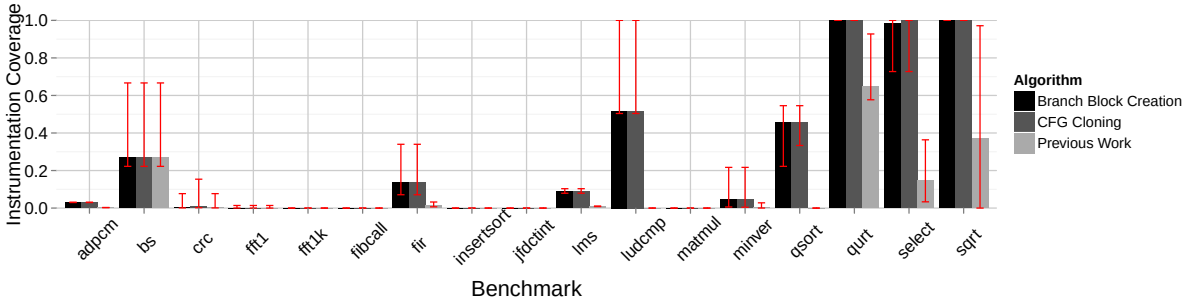


Figure 3.6: Average instrumentation coverage for program transformation

Table 3.1: ETPsem and number of retries

Benchmark	ETPsem			Retries		
	Previous	Creation	Cloning	Previous	Creation	Cloning
adpcm-test	8.719×10^9	7.583×10^8	7.441×10^8	0	0	4
bs	4.417×10^1	4.417×10^1	4.417×10^1	1	1	1
crc	1.725×10^6	1.724×10^6	8.609×10^5	1	1	1
fft1	3.325×10^9	3.325×10^9	3.341×10^9	0	0	0
fir	8.576×10^6	8.241×10^5	8.241×10^5	1	0	0
lms	9.950×10^7	1.139×10^7	1.139×10^7	0	0	0
ludcmp	-	9.863×10^6	9.863×10^6	0	0	0
minver	2.105×10^8	1.614×10^7	1.620×10^7	0	0	0
qsort-exam	-	1.739×10^2	1.297×10^2	0	0	2
qurt	2.251×10^3	1.513×10^3	1.513×10^3	0	0	0
select	9.324×10^2	1.321×10^2	1.021×10^2	0	1	3
sqrt	7.397×10^3	2.470×10^3	2.470×10^3	0	0	0

Table 3.1 shows the results for ETPsem and the number of retries for each of the instrumentation approaches. Table 3.2 shows the results for the instrumentation time and the increase in code size for each of the instrumentation approaches. We omit the data of the five benchmarks for which none of the instrumentation approaches was able to extract any traces. ETPsem is undefined for two benchmarks with previous work which fails to extract any information from these benchmarks. The instrumentation for all benchmarks is limited by a 2% debugging budget β, and if exceeded, the instrumentation tool will repeat the instrumentation while reducing coverage. The benchmarks *qsort-exam* and *select* show the increase in code size when using CFG Cloning.

Figure 3.7 illustrates the benefit of the proposed instrumentation approaches. The vertical line represents the WCET plus a 2% debugging budget β. The figure shows the uninstrumented ETP of the *qsort-exam* benchmark. It also shows the shifted ETPs with four different instrumentation approaches; previous work, Branch Block Creation, CFG Cloning, and naive instrumentation. The time-aware instrumentation techniques shift the ETP within the debugging budget β. Naive instrumentation instruments for vari-

Table 3.2: The overhead on the WCP and increase in code size

Benchmark	Instrum. Time [mS]			Code Size Inc.[bytes]		
	Previous	Creation	Cloning	Previous	Creation	Cloning
adpcm-test	167	177	420	8	84	468
bs	30	38	48	16	16	16
crc	32	44	61	20	28	608
fft1	57	69	79	28	44	88
fir	108	91	91	36	48	52
lms	59	66	70	32	40	44
ludcmp	62	67	70	0	20	108
minver	100	104	109	24	144	512
qsort-exam	55	63	104	31	40	1628
qurt	32	41	51	24	32	36
select	69	101	135	20	144	2,000
sqrt	29	33	35	24	32	36

ables of interest without taking timing into account. Although, the instrumented program achieves full coverage, its ETP fails to obey the timing requirement.

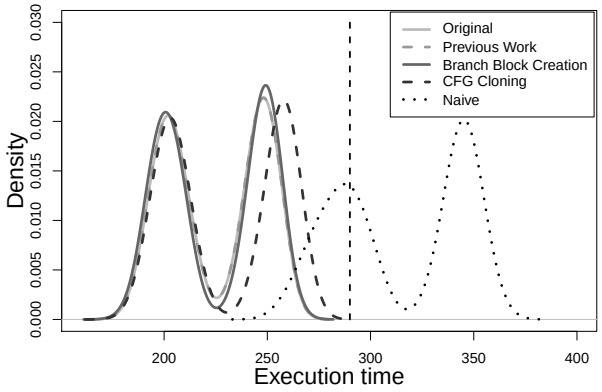


Figure 3.7: Execution time profiles for *qsort-exam*

Discussion

In 11 out of 12 instrumentable benchmarks, Branch Block Creation and CFG Cloning always increase the instrumentation coverage. For the benchmark *bs*, all instrumentation methods perform alike. In many cases, both transformation techniques perform equally well, in terms of coverage, except for *select* in which CFG Cloning performs better. The reason is that Branch Block Creation has less basic blocks to instrument on the execution path, whereas CFG Cloning clones basic blocks along with their subgraphs thus more instrumentable basic blocks. Hence, using CFG Cloning adds more flexibility to the instrumentation process leading to an increase in the instrumentation coverage for some executions of the programs.

ETPsem is effective in identifying efficient instrumentation methods. The better the utilization of slack on non-WCPs for the obtained coverage, we get a smaller value of ETPsem. For the *adpcm-test* benchmark, for example, both Branch Block Creation and CFG Cloning increase the coverage from 0.003 to 0.03 compared to previous work. According to their ETPsem, however, Branch Block Creation and CFG Cloning shift the ETP by 6.3% and 5.4% more than previous work, respectively. In such case, it is up to the developer to decide whether such a shift in the program's ETP is acceptable for the corresponding increase in coverage. In some cases, the coverage obtained by Branch Block Creation and CFG Cloning are equal but the values obtained from ETPsem are substantially different such as for benchmark *qsort-exam*. The reason is that although both essentially extract the same amount of information, Branch Block Creation utilizes less slack for instrumentation. For the *fft1* benchmark, although all three instrumentation methods achieve the same coverage, CFG Cloning has a higher ETPsem. This means that CFG Cloning is better at utilizing the program's slack to extract the same information. ETPsem is thus useful for choosing an instrumentation metric over the other if, for example, their coverage match.

The instrumentation time of the benchmarks is acceptable and has a maximum of 420 mS. As a test for scalability, we concatenated all benchmarks into one C file of about 3200 lines of code. The instrumentation time was 653 mS. In most cases, the first instrumentation attempt was successful in honoring the debugging budget β. The tool had to adjust the instrumentation in some cases with a maximum of 4 retries.

Out of 17 benchmarks, 5 are not instrumentable even after transformation. Two benchmarks are instrumentable only after transformation. Branch Block Creation and CFG Cloning increase the instrumentation coverage, on average over all benchmarks, compared to previous work by 5.8 and 5.9 times, respectively. CFG Cloning increases the program code size considerably but gives more flexibility to instrumentation and has a better utilization of slack in most of the cases. ETPsem is an indicator of the effectiveness of the instrumentation approaches.

3.5 Slack-based Conditional Instrumentation

Slack-based conditional instrumentation is a mechanism that allows developers to instrument programs on the WCP [68]. As such it permits them to extract information from the program at run time, even if the current execution flow includes some blocks of the WCP.

The assumptions underlying the idea of slack-based conditional instrumentation are that (1) timing constraints imposed on applications are typically conservative and (2) the WCP, even when executed, rarely uses the WCET [152]. We conclude from the first assumption that there is a static time window α between the WCET and the deadline. This time window is static because it is independent of the actual execution time at run time. It is common in safety critical applications, for instance, to keep the CPU utilization low [102]. From the second assumption, we conclude that there is run-time (dynamic) slack between the actual execution time and the WCET [150]. Slack-based conditional instrumentation will use the run-time slack by executing the instrumentation code, if the

system has sufficient slack available; hence, the term *conditional* in slack-based conditional instrumentation.

We present three scenarios for slack-based conditional instrumentation:

- **Scenario 1: Basic method.** We insert instrumentation points at all variable assignments that we want to trace.
- **Scenario 2: Minimizing the number of instrumentation points.** It is crucial to add as little to the program as necessary regardless whether we add program code, timing overhead, or memory overhead. This scenario discusses minimizing the overall number of instrumentation points in the program.
- **Scenario 3: Minimizing the number of instrumentation points in the presence of timing constraints**. This scenario builds upon the previous one and is relevant for real-time applications. Assuming the application has less time budget than needed to trace all variable assignments, here we investigate adding to the program only a subset of the instrumentation points to produce, on average, the maximal amount of trace information during runs.

For this work, we implement the slack-based conditional instrumentation for all scenarios using both software and hardware solutions.

3.5.1 The Underlying Concepts

In this section, we refer to conditional instrumentation code points, which are conditionally executed at run-time based on available run-time slack, as conditional instrumentation points (CPs). We also refer to instrumentation code points used in previous work [39] and used in Section 3.4, which are always executed at run time, as IPs. We assume that we can reliably estimate the WCET of instrumentation code points. Hence, there are two types of instrumentation points that we can insert into a program: CPs and IPs.

We extend the model presented in Section 3.2 to support conditional instrumentation points. We extend the definition of a vertex to support a third instrumentation type *CP* which denotes a conditional instrumentation code point as shown in Definition 5. Only one instrumentation point, either an IP or a CP, can be added to a vertex.

Definition 5 (Vertex)**.** *A vertex is a basic block with at most one assignment to the same variable. We represent a vertex as a tuple* $v = (i, t)$ *where* $i \in \mathbb{N}$ *is a unique identifier, and* $t \in \{None, IP, CP\}$ *is an instrumentation type associated with the vertex.*

We typically insert IPs at vertices that lie on paths other than the WCP. For instrumenting vertices on the WCP, we always use CPs. It might occur that after instrumentation, the WCP changes. We discover this through rerunning the WCET analysis after instrumentation. If the WCP changes, we will convert all instrumented vertices on the new WCP from IPs to CPs. Although, in this work, we use both IPs and CPs for instrumentation, it is possible to use only CPs to instrument the whole program. Using only CPs

for instrumentation facilitates instrumenting multiple WCP programs and eliminates the need for the conversion of IPs to CPs when the WCP changes. It, however, also leads to higher overhead on paths other than the WCP due to the conditional execution of CPs as compared to IPs.

CPs check whether there is sufficient slack time available to execute the instrumentation code, and if there is, then the program will execute the instrumentation code; otherwise, the program will skip it. We call the portion of code in the CP that checks whether sufficient slack is available to execute the instrumentation code as the overhead of the CP. The overhead of a CP is always executed.

Adding CPs on the WCP may lead to an increase in the WCET because of the additional overhead O. However, WCETs are typically lower than the application deadline. As mentioned in Section 3.2, perturbations in the WCETs are acceptable as long as they are less than or equal the debugging budget β. In practice, when we are instrumenting a single-task application, we first apply Scenarios 1 or 2. If they lead to an increase in the WCET beyond the application's deadline, then we will use Scenario 3 to choose a subset of the CPs for the available debugging budget β. For instrumenting concurrent applications, we must ensure the schedulability of the whole workload as mentioned in Section 3.2.

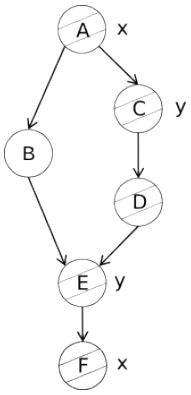
An Illustrative Example

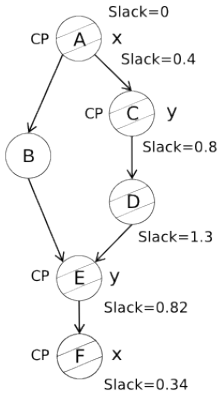
We illustrate slack-based conditional instrumentation with the example shown in Listing 3.1. The program code in Listing 3.1 shows that, after incrementing x, when the value of variable x is greater than 10, the program will update variables c, y, and z. Otherwise, it will increment the value of variable z. At the end of the code fragment, the program increments z, updates y, updates z, then increments x. We annotate Listing 3.1 with labels A, B, C, D, E, and F that identify vertices for its OSCCFG $\mathcal{G}$ shown in Figure 3.8.

```
A: x++;
   if ( x > 10 ) {
C:   c = z;
     z = y;
     y = c;
D:   c = x;
   } else {
B:   z++;
   }
E: z++;
   y += z;
F: z = c;
   x++;
```

Listing 3.1: C program without instrumentation

There are two paths in $\mathcal{G}$: $p_1 = \langle A, B, E, F \rangle$ and $p_2 = \langle A, C, D, E, F \rangle$. Let us assume that path p_2 is the WCP in this example, which we show as shaded vertices and we want to trace all state changes to variables x and y. We compare 3 cases of instrumentation to trace all state changes of x and y in this example, namely IPs on WCP, CPs on WCP (Scenario 1),





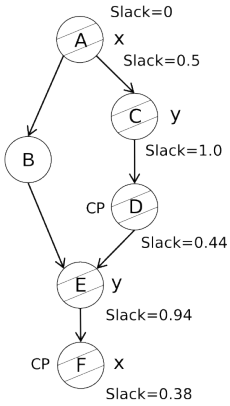


Figure 3.8: Original program Figure 3.9: All CPs on WCP Figure 3.10: Minimal CPs on WCP

and minimal CPs on WCP (Scenario 2). In order to compare the different instrumentation cases, consider the following *hypothetical* setting (used only for illustration). Assume that each basic block in $\mathcal{G}$ has a WCET of 2 time units and run-time execution time of 1.5 time units. Let the cost of recording and retrieving the current state of the program be 0.8 time units and recording a single variable to a buffer be 0.08 time units. Let the cost of checking whether sufficient run-time slack exists be 0.1 time units. For example, to check whether sufficient run-time slack exists and if so, record a variable x to the buffer, the cost incurred would be $0.1 + 0.8 + 0.08 = 0.98$ time units. Assume that the debugging budget assigned to the program by the developer is 1 time unit. Hence, in the worst case, the instrumented program must finish execution by $2 * 5 + 1 = 11$ time units. Note that instrumentation points are executed at the end of basic blocks but before any branches are executed. Using the above mentioned *hypothetical* setting, the 3 cases are as follows:

- **Case I: All IPs on WCP.** Since A, C, E and F modify either x or y, we add IPs to these basic blocks. Using the above setting and upon execution of the program, this results in an execution time of $7.5 + (0.88) * 4 = 11.02$ time units, which exceeds the deadline assigned to the program.

- **Case II: All CPs on WCP (Scenario 1).** In this case, we add CPs to basic blocks A, C, E and F, shown in Figure 3.9. Figure 3.9 also shows the buildup of run-time slack as program executes. The cost of checking for run-time slack is 0.1, and the cost of recording a single variable (including checking for run-time slack) is 0.98 according to the *hypothetical* setting mentioned above. If a basic block annotated with CP has sufficient run-time slack to execute the instrumentation then the instrumentation block is executed, hence reducing available run-time slack by 0.98, otherwise the run-time slack is reduced by 0.1. As seen in Figure 3.9, A and C have insufficient run-time slack to execute the instrumentation code to record x and y, respectively. At each of the basic blocks A and C, after the basic block executes, the run-time slack increases by 0.5 but decreases by 0.1 to check for sufficient run-time slack at the CP. Basic blocks E and F have sufficient run-time slack to execute instrumentation code to record y and x, respectively. Each of the CPs at E and F consume 0.98 of the run-time slack to instrument the variable at that CP.

- **Case III: Minimal CPs on WCP (Scenario 2).** In this case, we add CPs only at basic blocks D and F which record both variables x and y as shown in Figure 3.10. The cost of recording both x and y at a basic block, including checking for run-time slack, is $0.8 + 0.08 * 2 + 0.1 = 1.06$ according to the setting mentioned above. It can be seen that sufficient run-time slack is available at D and F to record variables x and y.

One can infer from the above example the following conclusions. The addition of IPs on the WCP could lead to a violation of the deadline assigned to the application, which is not desirable in a safety-critical real time setting. With addition of CPs on the WCP, there is a higher chance of execution of the instrumentation code if placed at a vertex that is further away from the start vertex (closer to the exit vertex). This is simply due to the fact that run-time slack builds up as basic blocks are executed during the program execution. Scenario 2 makes use of this by minimizing CPs and delaying the tracing of variables (more details in Section 3.5.2). Lastly, with every CP introduced in the program, there is a mandatory cost of checking whether sufficient run-time slack exists at each CP. Suppose if the cost of checking for sufficient run-time slack is high enough for the program to miss its deadline, then one has to selectively insert CPs in the program. We use Scenario 3 as a remedy for the above mentioned problem, to select CPs such that maximum state changes of variables of interest are recorded (more details in Section 3.5.3).

3.5.2 Minimization of CPs on the WCP

In this section, we describe Scenario 2 which minimizes the number of CPs that capture all possible state changes of the variables that we want to trace on the WCP. This scenario maintains a set of modified variables. We delay capturing the state change of this set of modified variables till just before any of the variables gets overwritten. When we delay capturing a state change, the program executes more code, which probably leads to gaining more run-time slack as illustrated in the example discussed in Section 3.5.1. Gaining more run-time slack will increase the likelihood of the execution of the CPs. We denote the set $traceVars \subseteq VARS$ as the variables we want to trace where $VARS$ is the set of all variables in the program. Our approach extracts a set of vertices such that each vertex contains at most one state change of any variable in $traceVars$. We annotate each of these vertices as a CP. This ensures that the instrumented program will capture (if run-time slack permits) all possible state changes for all variables in $traceVars$.

We illustrate this approach by revisiting the OSCCFG shown in Figure 3.10. Path $p_2 = \langle A, C, D, E, F \rangle$ is the WCP, which we show as shaded vertices. This approach identifies subpaths $\langle A, C, D \rangle$ and $\langle E, F \rangle$ that capture one state change of each of x and y. We change the instrumentation type of the destination vertices of these paths D and F to CP.

We show this algorithm in Function 3. It takes as input the variables that we want to trace, $traceVars$, and the WCP between start vertex v_s and exit vertex v_x, p_{v_s,v_x}. The output is the set of vertices $V^{\{\}}$ that have to be instrumented. We introduce several helper

functions in describing this algorithm. To extract the set of variables being assigned new values in a vertex, we use function $modifiedVars(v) : V \rightarrow var^{\{\}}$. We use the function $predecessor(v) : V \rightarrow V$ to extract the vertex that has an incident edge on v on the WCP. We use functions $scopeBegin : V \rightarrow \{true, false\}$ and $scopeEnd : V \rightarrow \{true, false\}$ to identify the beginning and end of scopes, respectively. A scope corresponds to a loop on the WCP and can be identified by static analysis [117, 86] (also applies to continue and break statements). Note that if a vertex marks the beginning of multiple scopes, it will be split into multiple vertices such that each new vertex marks the beginning of only one scope. Function $getScopeVars : V \rightarrow var^{\{\}}$ is used to extract the set of variables that are modified within a scope where the input argument to $getScopeVars$ corresponds to the beginning of the scope. The stack operations *push* and *pop* are used to push an element and pop an element from a stack, respectively.

The core idea of the algorithm is to delay the recording of a variable change until the point where at least one of the variables of interest gets overwritten. Function 3 iterates through the vertices on the WCP p_{v_s,v_x} in order. While iterating through the vertices, set M holds the variables of interest that have been modified without any of the variables being overwritten. Set I holds the set of vertices to be conditionally instrumented along with the variables to instrument at each vertex. For each vertex v, the algorithm extracts the set $modVars$ which is the set of variables of interest that vertex v modifies. First, the algorithm checks whether vertex v begins a new scope. If vertex v begins a new scope and this scope modifies any of the variables in set M, then the algorithm will choose to instrument all variables in M before entering the scope, i.e., at the vertex preceding the beginning of the scope (excluding the loop's back-edge). After a new scope starts, the set M is pushed into a stack S. If a vertex modifies any of the variables in the set M, then the algorithm will choose to instrument variables in M at the preceding vertex. Afterwards, set M will be updated with the set of modified variables at vertex v. If a vertex v marks the end of a scope, then the vertex v will be instrumented with the variables in set M (if any) and set M will be popped from the stack S to restore the set of modified variables before the scope started. If the exit vertex is reached and the set M is not empty, then the exist vertex will be instrumented with the variables in set M. Finally, we iterate through the set $V^{\{\}}$ obtained as output from Function 3 and add CPs to these vertices to record the specific variables associated with each vertex. The complexity of the algorithm is linear in the number of vertices on the WCP.

3.5.3 Constrained Minimization of CPs

In this section, we augment the approach from Section 3.5.2 to consider a constraint on the increase in the WCET caused by instrumenting the WCP. We minimize the number of instrumentation points to capture the maximum number of state changes of variables in *traceVars* given a debugging budget β.

We use Function 3 to get a minimal number of CPs required to trace all the state changes of variables of interest (*traceVars*). Then, we select a subset of these CPs such that we trace the maximum number of variables of interest keeping the overhead of the

Function 3 Minimization of CPs on WCP

Input: *traceVars*, p_{v_s,v_x}
Output: $V^{\{\}}$

Let $M \Leftarrow \emptyset$ be the set of variables being monitored
Let $I \Leftarrow \emptyset$ be the set of instrumented vertices
Let S be an empty stack

for $v \in vertices(p_{v_s,v_x})$ **do**
  $modVars \leftarrow modifiedVars(v) \cap traceVars$

  if $scopeBegin(v)$ **then**
    if $getScopeVars(v) \cap M \neq \emptyset$ **then**
      $I \leftarrow I \cup \{(predecessor(v), M)\}$
      $M \leftarrow \emptyset$
    end if
    $push(S, M)$
  end if

  if $M \cap modVars \neq \emptyset$ **then**
    $I \leftarrow I \cup \{(predecessor(v), M)\}$
    $M \leftarrow \emptyset$
  end if
  $M \leftarrow M \cup modVars$

  if $scopeEnd(v)$ **then**
    if $M \neq \emptyset$ **then**
      $I \leftarrow I \cup \{v, M\}$
    end if
    $M \leftarrow pop(S)$
  end if
end for

if $M \neq \emptyset$ **then**
  $I \leftarrow I \cup \{v_x, M\}$
end if
return I

CPs within the budget β. We can describe the problem of selecting a subset of CPs using Equation 3.3.

$$\begin{aligned} &\text{Max} \sum_{i=1}^{n} b_i * (\mathit{varsInCP}_i * \mathit{frequency}_i) \\ &\text{subject to } \sum_{i=1}^{n} b_i * (\mathit{overhead}_i * \mathit{frequency}_i) \leq \beta \\ &\text{where } b_i \in \{0, 1\} \text{ for } i = 1, 2, \ldots, n \end{aligned} \tag{3.3}$$

Here, $\mathit{varsInCP}_i$ is the number of variables monitored in the CP i, $\mathit{frequency}_i$ is the number of times CP i is attempted, and $\mathit{overhead}_i$ is the overhead of CP i. The total number of CPs, n, is obtained from Function 3, and b_i is the BIP variable. Notice that we do not include the execution time incurred by the instrumentation code because at run-time we determine whether we have sufficient run-time slack to execute the instructions monitoring the variables. Note also that the WCET analysis tool determines $\mathit{frequency}_i$. It is important to clarify that $\mathit{frequency}_i$ is *not* the number of times the instrumentation code inside CP i executes, but rather the frequency of executing the conditional check of the CP, i.e. the number of times the CP is attempted. Although using the value supplied by the WCET analysis tool is pessimistic, our goal is to make sure that we honor the program's timing constraints.

Solving our problem for finding a subset of CPs to create is NP-Complete. We can show this by first polynomially reducing the binary knapsack problem to our problem, thus proving that its NP-Hard, then showing that our problem $\in$ NP. The proof is similar to the one we derived in Section 3.4.2. We solve our problem using BIP. We use standard BIP tools to solve the optimization problem and instrument the resulting vertices picked by the BIP solver.

3.5.4 Implementation Approaches

We experimented with two approaches for implementing the instrumentation of a program: software-based and hardware-based. The software-based approach makes changes to the program code using traditional programming constructs, and the hardware-based approach uses special instruction-set architecture (ISA) extensions to perform the instrumentation. Although, the software-based approach is simpler to implement, we consider both approaches to compare their perturbation costs. We briefly describe these two approaches.

Software Approach

The software-based approach uses function calls in the program to extract cycle counter values. Listing 3.2 shows a simple example of a software implementation of CPs. Functions *func_a* and *func_b* have WCETs of $wcet_a$ and $wcet_b$, respectively. Assume that the WCET of all instrumentation code at labels B, C, and D is $wcet_{c_1}$, and at labels E and F is $wcet_{c_2}$.

```
int main(void){
A:  globalTime = getTime() + wcet_a;
    func_a();
B:  if (globalTime - getTime() >= wcet_c1){
C:      // Instrumentation Code
        ..............
    }
D:  globalTime += wcet_b;
    func_b();
E:  if (globalTime - getTime() >= wcet_c2){
F:      // Instrumentation Code
        ..............
    }
}
```

Listing 3.2: A software implementation of conditional instrumentation

When the program executes, it sets variable *globalTime* at label A, to the time at which function *func_a* will finish execution in the worst-case. After *func_a* completes, the instrumentation code compares *globalTime* to the current time to check whether there is sufficient run-time slack to execute the instrumentation code of *func_a*. The instrumentation code then updates *globalTime* at label D to hold the time at which function *func_b* finishes in the worst-case. The same check for instrumentation is repeated after *func_b*.

Note that *getTime* is *not* an OS function call but rather an instruction that reads a dedicated free running hardware timer on the chosen processor. Hardware timers exist in the processors used for embedded systems and are either memory- or register-mapped timers.

Hardware Approach

Our hardware-based approach requires extensions to the ISA. We describe these ISA extensions and their use with a simple example.

Hardware Extensions: We extend a cycle-accurate ARMv5 architecture platform with a 32-bit count-down timer, and we extend its ISA with two instructions. We introduce the set timer *stt* instruction, and a check time *chk* instruction. Figure 3.11 shows the *stt* and *chk* instruction encodings. The *stt* instruction has a single 16-bit immediate operand ⟨*timH:timL*⟩ while the *chk* instruction has two 8-bit immediate operands; ⟨*slk*⟩ and ⟨*raddrL:raddrH*⟩. Every clock cycle, the 32-bit timer will decrement its value by one if it is greater than zero. The *stt* instruction adds its 16-bit operand, ⟨*timH:timL*⟩, to the value already in the timer. The *chk* instruction compares its first 8-bit operand ⟨*slk*⟩ to the value of the timer, and if the first operand value is greater than the timer value then the processor will branch past the number of instructions specified in the second 8-bit operand of the instruction, ⟨*raddrL:raddrH*⟩. Otherwise, the code will execute normally without branches.

We incorporate the *stt* and *chk* instructions into the five-stage pipelined architecture consisting of *Fetch*, *Decode*, *Execute*, *Memory* and *Writeback* stages. In the *Fetch* stage,

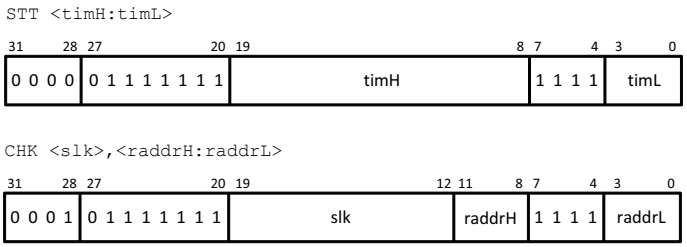


Figure 3.11: Instruction encodings of the *stt* and *chk* instructions

the processor fetches the instruction at the address in the program counter and increments the program counter by four. In the *Decode stage*, the processor decodes the instructions and reads the value of the 32-bit timer in the case of *stt* or *chk*. In the *Execute stage* of the *stt* instruction, the arithmetic logic unit (ALU) adds the value of the 16-bit operand to the timer value and the processor writes back the result to the timer in the *Writeback stage*. In the case of executing the *chk* instruction, the ALU subtracts the first 8-bit operand from the timer value in the *Execute* stage. In the same stage, the branch logic shifts the second 8-bit operand two bits to the left and adds the result to the new program counter value from the *Fetch stage*. If the result of the ALU operation is negative, then a multiplexer (MUX) will set the program counter to the output of the branch logic in the next *Fetch* stage, i.e., a branch will occur. Otherwise, the MUX output will set the program counter to the incremented value of the program counter from the previous *Fetch* stage.

Functional Operation: Listing 3.3, a rework of Listing 3.2, illustrates the use of *stt* and *chk* instructions in slack-based conditional instrumentation. Functions *func_a* and *func_b* have WCETs of $wcet_a$ and $wcet_b$ cycles, respectively. The instrumentation code, *chk* and *stt* instructions at labels B, C, and D have a WCET of $wcet_{c_1}$ cycles. The instruction count in the instrumentation code at label C is $instr_{c_1}$. The *chk* instruction and instrumentation code at labels E and F have a WCET of $wcet_{c_2}$ cycles. The instruction count in the instrumentation code at label F is $instr_{c_2}$.

```
int main(void){
A:  asm("stt wcet_a");
    func_a();
B:  asm("chk wcet_c1,instr_c1");
C:  // Instrumentation Code
    ..............
D:  asm("stt wcet_b");
    func_b();
E:  asm("chk wcet_c2,instr_c2");
F:  // Instrumentation Code
    ..............
}
```

Listing 3.3: Conditional instrumentation using *stt* and *chk* instructions

For one execution of the example shown, functions *func_a* and *func_b* will have an

actual execution time of $exec_a$ and $exec_b$ cycles, respectively. The *stt* instruction at label A sets the timer to $wcet_a$ (assuming the timer is initialized to zero at the start of program execution). The *chk* instruction at label B compares the timer ($wcet_a - exec_a$) to $wcet_{c_1}$ which is the time needed to instrument function *func_a*. If the timer is greater than or equal to the instrumentation time, the processor will execute the instrumentation code. Otherwise, the processor will branch forward $instr_{c_1}$ instructions, past the instrumentation instructions to function *func_b*. The next pair of *stt* and *chk* instructions operate similarly for function *func_b*. Notice that these instructions make use of accumulated run-time slack. If the run-time slack is insufficient to execute a CP, the timer will carry forward the run-time slack for use at the next CP.

3.5.5 Experimentation

The time-aware slack-based conditional instrumentation tool is fully implemented and automated. We use a cycle-accurate ARM simulator as a platform for the implementations. We choose this platform because we propose ISA extensions for the hardware implementation. Hence, the Unisim cycle-accurate simulator [6] offered us a convenient platform to compare both the software and hardware implementations. The software approach, however, can easily be applied to any other platform. Since we implemented our software approach for Unisim, we implemented *getTime* as a single instruction that reads the simulator's timestamp. We use the Unisim's default configurations including the latencies for register and memory accesses.

Following the time-aware instrumentation flow in Figure 3.1, the tool starts off with the identification of the basic blocks for the input program and the extraction of its OSCCFG at the C source code level. The tool then calls a WCET analysis tool to compute the WCET for each basic block in the OSCCFG. We also use the WCET analysis tool to find the WCP for the input program. We used RapiTime v2.4 [2] as our WCET analysis tool. The instrumentation tool then proposes an instrumentation configuration for either the software or hardware approaches and inserts the instrumentation points in the basic blocks based on the chosen instrumentation scenario. Finally, the tool compiles the instrumented program using the Unisim ARMv5 cross compiler, runs the cycle-level simulation, extracts the logged trace data from the instrumentation, and quantitatively analyzes the data for the sake of experimentation. The tool re-analyzes the WCET of the instrumented program. This may trigger a rerun of the instrumentation process.

We experiment with the SNU real-time benchmark suite [3], which contains 17 C benchmarks that implement numeric and DSP algorithms. They have 117 lines of code and 34 basic blocks on average. We compare our approaches with the technique proposed by Fischmeister et al. [39], which we refer to as *previous work*.

The goals of our experimentation are as follows:

- Show that conditional instrumentation extracts more trace information compared to previous work.
- Assess the effectiveness of Scenarios 2 and 3 in minimizing the overhead on the WCP.

- Quantify the overhead added by the instrumentation process in the software and hardware implementations.
- Investigate the effect of conditional instrumentation and the different implementations on the ETP of the input program.

To experiment with Scenarios 1 and 2, we run the SNU benchmark with its provided input for the software and hardware implementations. To illustrate the effect of conditional instrumentation on the execution time, we generate an ETP for one benchmark with all permutations of its input data. For Scenario 3, we use one benchmark and instrument it for every possible debugging budget until we achieve maximum coverage. We trace all variables except function arguments, constants, and loop counters by logging them to dedicated memory buffers.

Metrics

We quantitatively assess our approach using the following metrics:

- **Ratio of executed vs attempted instrumentation points on the execution path:** A program with loops and/or multiple function calls may attempt to execute the same CP several times. A program executes a CP based on the presence of enough run-time slack for its execution. The total ratio of executed to attempted instrumentation points (both IPs and CPs) measures how successful an instrumentation approach is in inserting instrumentation code that a program eventually executes.
- **Instrumentation coverage on the execution path:** Instrumentation coverage along a path shows the ratio of variable assignments that are traced after running an instrumented program to those that a developer desires to trace. Therefore, it is the probability that instrumentation on a path captures variable assignments before their re-assignment and, hence, loss of information.
- **Inserted overhead on the WCP in cycles:** In the worst-case scenario, an instrumented program executes its WCP such that there is insufficient run-time slack to execute any CPs on that path. However, there is an increase in the program's WCET because CPs add overhead to the WCP. This includes any inserted overhead due to conditional checks or dynamic slack computation.
- **Increase in code size:** Every instrumentation point adds extra code to the original source code. The less the total increase in code size, the more effective the instrumentation approach is in utilizing code space for instrumentation.

Results for Scenarios 1 and 2

We compare software and hardware implementations of Scenarios 1 and 2 against previous work.

Figure 3.12 shows the ratio of executed to attempted instrumentation points over the execution path of the benchmarks. For some benchmarks, previous work shows no value in the figure for one of the following reasons: (1) the executed path never hits an IP which leaves the ratio undefined, (2) the benchmark has only a single path, which is the WCP, and thus it has no IPs, or (3) the benchmark has multiple paths, but IPs were removed because the instrumented path became the new WCP.

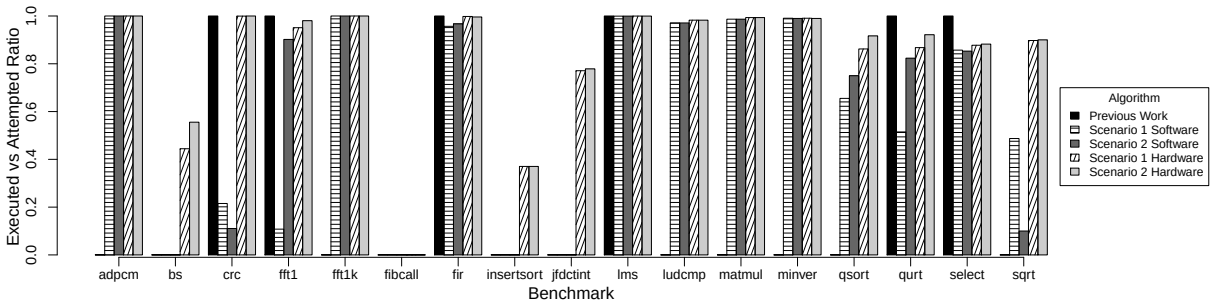


Figure 3.12: Executed vs attempted instrumentation points for Scenarios 1 and 2

The results show the benefits of conditional instrumentation. For previous work, a benchmark always executes attempted IPs; thus, the ratio when available is always 1.0. Previous work is only able to instrument six benchmarks. This is shown in Figure 3.12 where only benchmarks: *crc*, *fft1*, *fir*, *lms*, *qurt*, and *select* have values using the previous work's approach. Conditional instrumentation, even using the software solution, is able to trace all benchmarks but one (*fibcall*). While the ratio varies and it is up to the developer to decide whether the ratio is acceptable, conditional instrumentation at least provides the ability to trace.

For some benchmarks, all methods have the same ratio, but for others (*bs*, *insertsort* and *jfdctint*) the hardware implementation outperforms the software. For the *fibcall* benchmark, none of the methods executes any of the instrumentation code. This means that there is insufficient run-time slack at all CPs.

We also compare the instrumentation coverage for the software implementation of Scenario 1 against previous work in Table 3.3. This table shows the mean value, the 95% confidence interval (CI), and the standard error of mean (SEM). Even with conservative estimates, the software implementation is at least one order of magnitude better than previous work.

Table 3.3: Instrumentation coverage for the software implementation of Scenario 1

Instrumentation	**Mean**	**95% CI**	**SEM**
Previous Work	0.026	0.040	0.019
Scenario 1 - Software	0.569	0.220	0.104

Figure 3.13 presents the instrumentation coverage comparison. The hardware implementation clearly increases the instrumentation coverage compared to software. Notice

that for the hardware implementation, Scenario 2 has higher coverage compared to Scenario 1. However, the *jfdctint* benchmark violates this rule. This result shows that merging instrumentation points does not necessarily lead to a higher instrumentation coverage. The reason is that although the run-time slack might be sufficient to execute a small instrumentation code, it might be insufficient to execute larger instrumentation code where the smaller one is merged. This also explains why for the *sqrt* benchmark, using the software implementation, Scenario 1 has higher coverage than Scenario 2.

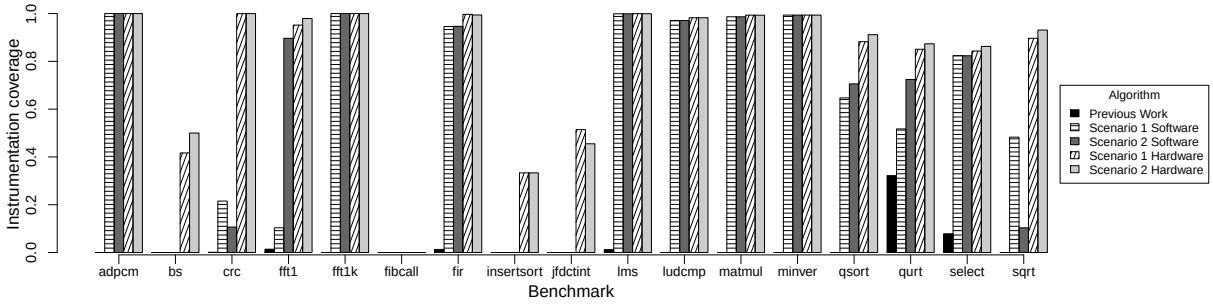


Figure 3.13: Instrumentation coverage for Scenarios 1 and 2

Table 3.4 presents the cost, in terms of the execution overhead on the WCP in cycles, O, and the increase in code size in bytes, of logging variable assignments for each of the instrumentation approaches. The values of the overhead and code size increase for the software approach are relatively high as compared to the WCET and the code size of the original programs. Although previous work leaves the WCET of the WCP unchanged and has minimal increase in code size, it also has least coverage of the proposed approaches. One value shows an overhead of -90 which means that the WCET of the program decreased after instrumentation which may happen due to variations in cache hits/misses and branch predictions [152].

We compare the increase in code size of our approaches to the software implementation of Scenario 1. The increase in code size of Scenario 2 software, Scenario 1 hardware and Scenario 2 hardware have values of 92.4%, 31.9% and 29.7% on average. In the worst-case, the hardware methods minimize the increase in code size 2.3 times as compared to the software methods.

Moreover, the hardware implementation decreases the overhead on the WCP. On average, compared to the overhead on the WCP of Scenario 1 software, Scenario 2 software, Scenario 1 hardware and Scenario 2 hardware have values of 89.4%, 21.0% and 22.7%, respectively. The minimization of instrumentation points does not always lead to a decrease in the overhead on the WCP. The reason is that adding code to the original program does not always lead to a higher execution time due to the variation in cache hits/misses and branch predictions. Thus, adding less instrumentation points does not always mean less overhead on the WCP.

Interpretation of Results: Slack-based conditional instrumentation instruments 16 benchmarks versus only six for previous work. The hardware implementation highly outperforms the software implementation and is able to gain an average of 41.5% coverage for

Table 3.4: Overhead on the WCP and the increase in code size for Scenarios 1 and 2

Benchmark	Previous Work		Software				Hardware			
			Scenario 1		Scenario 2		Scenario 1		Scenario 2	
	Code [Byte]	O [Cycle]	Code [Byte]	O [Cycle]	Code [Byte]	O [Cycle]	Code [Byte]	O [Cycle]	Code [Byte]	O [Cycle]
adpcm-test	0	0	5,956	9,150,736	5,376	7,037,912	2,064	817,030	1,968	638,385
bs	0	0	540	369	540	368	172	87	172	97
crc	20	0	992	85,363	796	82,334	288	26,915	224	26,699
fft1	40	0	1,556	4,963	1,488	5,702	504	71	480	-90
fft1k	0	0	1,004	1,598,363	944	809,057	316	101,491	300	58,187
fibcall	0	0	344	936	344	949	112	152	112	151
fir	76	0	2,208	67,581	2,048	49,607	708	20,298	680	19,588
insertsort	0	0	244	1,558	244	1,558	72	401	72	401
jfdctint	0	0	1,352	3,816	1,352	3,821	580	1,643	580	1,611
lms	76	0	2,336	716,426	2,156	587,927	776	163,416	728	114,402
ludcmp	20	0	1,136	6,570	1,068	6,628	348	1,530	324	1,561
matmul	0	0	236	5,415	236	5,415	64	1,634	64	1,634
minver	84	0	1,912	5,223	1,724	4,569	624	1,425	568	1,324
qsort-exam	0	0	636	1,240	576	1,119	192	364	176	357
qurt	48	0	1,320	2,338	1,088	2,163	428	1,590	372	1,283
select	52	0	688	1,807	628	1,474	216	496	200	426
sqrt	48	0	704	2,833	572	2,683	216	1,049	172	1,187

three benchmarks that the software implementation fails to extract any data from. The hardware implementation decreases the overhead on the WCP compared to software by 21.0% and it also decreases the increase in code size by 31.9%. In some cases, the minimization of CPs sometimes leads to a lower instrumentation coverage due to insufficient run-time slack to execute higher execution time CPs. Minimization of CPs sometimes also leads to higher overhead on the WCP which is due to timing anomalies [152].

Execution Time Profiles

To demonstrate the variation in the execution times with and without instrumentation, we present the ETPs for the *insertsort* benchmark, which implements the insertion sort algorithm. To generate the ETPs, we do the following: (1) use an input array of 6 elements and generate benchmarks for all possible permutations of the input elements, (2) compile and simulate these generated benchmarks, (3) analyze the execution time of each benchmark, and (4) repeat steps 1,2 and 3 for the software and hardware-based methods.

Figure 3.14 plots the different ETPs of the *insertsort* benchmark. The software instrumentation stretches the original ETP. The reason is the change of the instrumentation overhead as the input changes. The number of attempted CPs is least for the best-case execution times (minimum loop iterations) and, thus, the instrumentation overhead is minimal. As the execution time increases, i.e., more CPs are attempted, the overhead increases causing the stretching of the ETP. This is not the same for the hardware instrumentation ETP because the overhead of the software instrumentation is comparable to the execution

time of the instrumentation code itself unlike hardware instrumentation. The hardware instrumentation pushes the graph to the right and condenses it, i.e., the variance decreases. The overall effect of instrumentation on the original ETP is decreasing the density of the lower execution times and increasing the density of the higher execution times.

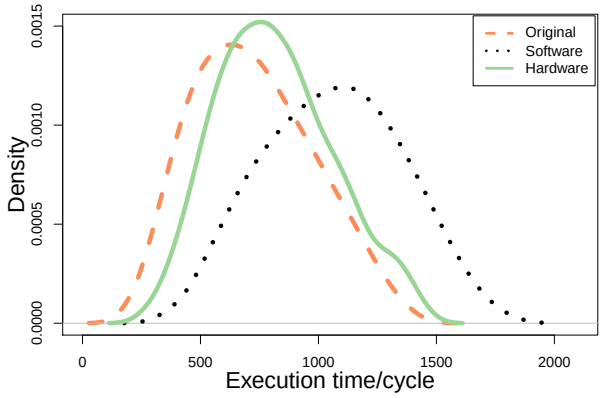


Figure 3.14: Execution time profiles for the *insertsort* benchmark

Interpretation of Results: The hardware implementation of conditional instrumentation results in the desired effect of condensing the ETP of a program. On the other hand, software implementation stretches the graph due to the high overhead of checking for run-time slack at the CPs.

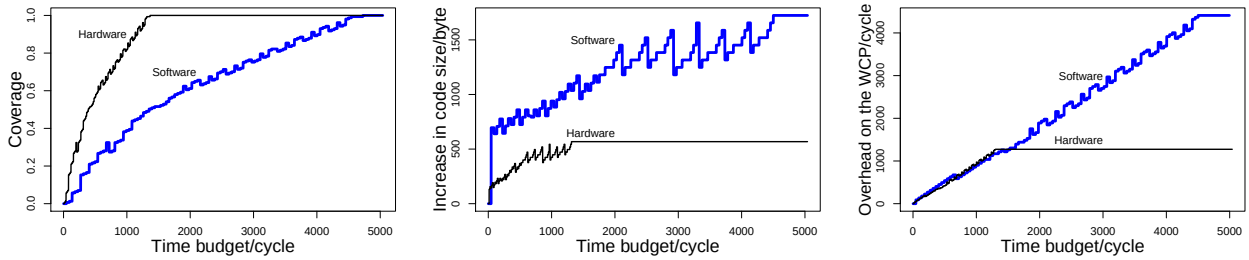
Results for Scenario 3

To experiment with the software and hardware implementations of the time constrained minimization instrumentation scenario, we present results for the *minver* program which performs a 3x3 matrix inversion. Notice that to see the effect of Scenario 3 in choosing CPs, we need to generate versions of the program each with a different time budget. In practice, the time budget available for instrumentation of a task depends on the the application and the underlying hardware. To collect results, we (1) calculate the overhead and the number of traced variables at each CP, (2) measure the frequency of execution of each CP, (3) increment the debugging budget β by one cycle (starting by 0), (4) run the algorithm given in Section 3.5.3 to instrument the program, (5) compile and cycle-level simulate the program, and (6) repeat steps 3, 4, and 5 until we reach the maximum instrumentation coverage.

Figure 3.15a shows the instrumentation coverage of the *minver* benchmark as the debugging budgetβ increases. We observe that the instrumentation coverage increases as β increases as expected. This is because more CPs can be inserted in the code. Notice that increasing β after all CPs are inserted, increases the coverage because it adds more initial slack to the program and thus the program executes more CPs. It is apparent from the figure that at some points increasing β leads to less coverage. The reason is that increasing β might lead to inserting a CP instead of a few others because the former traces more

variables. Typically, this should lead to higher coverage, however, it also means that the inserted CP needs more execution time and there might be insufficient run-time slack for its execution.

Figures 3.15b and 3.15c present the variation in the increase in code size and overhead on the WCP as β increases, respectively. Generally, code size and overhead increase as β increases but clearly there are large variations as the figures show. The reason for the sudden drops is that at a certain point increasing the budget leads to the replacement of many CPs by only one, because the latter traces more variables as compared to all the former combined, thus leading to a decrease in the added code size or added overhead.



(a) Instrumentation coverage (b) Increase in code size (c) Overhead on the WCP

Figure 3.15: Constrained minimization of CPs for the matrix inversion algorithm

Interpretation of Results: As expected, the instrumentation coverage, WCET overhead, and code size increase as the time budget for instrumentation increases. Sudden drops in code size and overhead happen when the instrumentation algorithm replaces a number of CPs with a smaller number of CPs, which maximizes our objective function and also leads to less code and less overhead. Drops in the coverage are due to the insertion of CPs that need more slack for execution and eventually fail to execute. This means that we need a better formalization of the problem in the presence of timing constraints (Scenario 3) to be able to choose optimal CPs.

3.6 Discussion

This section focuses on some high-level issues regarding the applicability of the results and the proposed techniques.

Usefulness of Partial Instrumentation: In the presentation of this work, the examples and experimentation focus on tracing data variables. Similarly, time-aware instrumentation can trace control flow and function calls. Since our main target is to ensure timeliness of the instrumented software, it imposes a constraint on the instrumentation process. This constraint interferes with achieving a fully instrumented software. Hence, a developer will sometimes want to partially sacrifice some of the logged information for

the sake of timeliness of a real-time system. A full instrumentation, however, can still be constructed from the union of multiple partial instrumentation instances. This might not be convenient in some cases, but, however, allows timely execution of the partially instrumented versions of the software versus a full instrumentation. Apart from constructing a full instrumentation, the extracted partial instrumentation are still useful. They can be used to build inductive debugging mechanisms for deployed resource and space constrained systems. Since it is hard to reproduce bugs from user bug reports, even having a partial trace can help extract information [15]. Partial traces can also be an input to additional debugging tools [123, 114]. In [114], the authors use partial traces in a portable trace-oriented debugger. In [123], the authors demonstrate the ability to build highly accurate calling context trees.

Instrumentation of multiple WCPs: Our analysis ignores the exceedingly rare case of multiple WCPs existing in a program. This case did not appear in any of the benchmarks used for experimentation. However, addressing multiple WCPs is an easy task. For program transformation techniques, our instrumentation tool would simply avoid instrumenting any WCP. The algorithm for finding instrumentable edges will then take multiple WCPs as an input. As for conditional instrumentation, We mentioned earlier that if the WCP changes after instrumentation, then the tool will convert the IPs on the new WCP to CPs. The tool can simply extend this concept to directly instrument multiple WCPs with CPs. If the target is the minimization of the number of CPs under a time budget constraint, then the tool will apply the minimization to all WCPs.

WCET analysis tools: Our analysis uses RapiTime [2] to obtain the WCET of basic blocks. RapiTime is a measurement-based WCET analysis tool and thus might underestimate the actual WCET. WCET, however, is only an input to our instrumentation tool and thus the validity of the proposed concept is independent of the accuracy of the analysis tool. The choice of RapiTime for WCET analysis was due to the availability of the tool in our labs, past experience using it, and independence of the architecture on which the software executes. It is also the de facto industrial standard applied in fields like aerospace and automotives. We can obviously replace RapiTime with a static analysis tool such as aiT [37] to obtain WCETs, but this is also known to be costly for modern architectures.

Rerunning the WCET analysis: The time-aware instrumentation techniques can modify the block layout of the program especially CFG Cloning. Modifying the layout can change memory and cache profiles, which in turn may change the WCET or even the WCP of the program. We discover such changes through rerunning the WCET analysis. This is also required to ensure that the new WCET, after inserting instrumentation points, is still below the deadline. If changes happen and the debugging budget β disallows the change, the framework will attempt a different transformation configuration. For program transformation techniques, the framework will reduce the number of cloned or inserted blocks, for instance. For conditional instrumentation, the framework will reduce the number of inserted CPs. Note that since the analysis uses a measurement-based

WCET analysis tool, it will only consider instrumentation code in a CP if run-time slack is available to execute it at run time. For static analysis tools, we remove parts that will be conditionally executed at run time, or else the tool will consider them part of a new WCP. In general, the number of instrumentation retries is usually low [62].

Extensibility to other optimization criteria: In Sections 3.4.2, 3.4.3, and 3.5.3, our algorithms only focus on optimizing for the debugging budget β or for code size. The value of inserting or cloning basic blocks, and of inserting CPs is determined by the number of traced variables and the frequency of attempting to trace these variables. Other criteria can be considered such as the usefulness of the traced variables from a tracing perspective that allows for making more optimal decisions.

Software vs hardware implementations of conditional instrumentation: The experiments show that the hardware implementation adds less code and execution time overhead to the original code compared to the software implementation. The reason is that software conditions and incrementing the global timer translate to several assembly instructions as opposed to single instructions that we propose in hardware. Although changing the architecture will modify the experimentation results due to ISA changes, the proposed concepts remain valid.

Limitations of the proposed approaches: In the experimentation, we focused on tracing scalar variables. It is possible to extract other information such as array elements, function calls, or branches. However, there can be other constraints on the instrumentation process besides the temporal constraints such as the size of the information to be extracted, memory available for buffering information, and the bandwidth available for sending information off-chip. For the proposed approaches, however, we only consider the timing constraints. Other constraints present research opportunities such as periodic partial emptying of instrumentation buffers to maximize instrumentation coverage under memory constraints.

We assume the availability of WCET analysis tools to obtain the WCET of programs as well as that of instrumentation code [152, 25, 22]. We also assume that we can obtain the WCP of a program as a sequence of vertices from the start to the exit vertex [82] (context-sensitive WCET analyzers are not used). Our tool reruns the WCET analysis for the instrumented program to ensure timeliness after changes to cache behavior due to the insertion of instrumentation points. While analyzing cache behavior is possible [49], our tool does not model or analyze cache behavior and, hence, the need to rerun the WCET analysis. The number of retries of the instrumentation process is usually low [62].

Conditional instrumentation is based on the inherent assumption that the overhead of a CP is less than the WCET of the instrumentation code inside the CP. Otherwise, replacing the CP with an IP would have less overhead. Usually, the instrumentation code involves reading variables from memory and either writing these variables to memory buffers or sending them off-chip. Hence, CPs usually have less overhead than their instrumentation code. Otherwise, our tool can be modified to replace a CP with an IP in that case.

3.7 Summary

Instrumentation for information extraction supports understanding specific aspects and behavior of the software at run time. Time-aware instrumentation tries to preserve logical correctness *and* timing constraints during instrumentation.

In this chapter, we introduce ETPsem as a new metric for measuring the performance of time-aware instrumentation techniques. We also present two program transformation approaches that can be used to increase the effectiveness of time-aware instrumentation. On average, these transformation approaches increase the instrumentation coverage by at least five times compared to direct instrumentation of the original programs. While the two approaches are straightforward, they and the new metric lay the foundation for future work for more complicated approaches as well as for instrumentation mechanisms going beyond timing and logical correctness.

In this chapter, we also investigate a slack-based conditional instrumentation mechanism that obeys timing constraints but can also extract information on the WCP. We compared hardware and software-based implementations in detail and proposed solutions to two problems of how to efficiently use slack-based conditional instrumentation. We also reported on non-intuitive results such as minimizing the number of instrumentation points may have negative side effects. Overall, however, conditional instrumentation improves over previous work on time-aware instrumentation by an order of magnitude in instrumentation coverage and several orders of magnitude in the number of executed instrumentation points at the expense of code size.

Chapter 4

INSTEP: A Static Time-Aware Instrumentation Framework

Software systems are rich in extra-functional (or non-functional [132, 85]) requirements such as timing, code sizes, communication bandwidth, power consumption, and memory consumption. Current instrumentation techniques are unfit for such systems, because these techniques ignore such extra-functional properties. Consequently, using a current instrumentation framework for such systems can introduce side effects that produce unintended behavior. For instance, embedded software run on micro-controllers that might have limited on-chip memory. Instrumenting such software programs might lead to exceeding the memory limit. Another example is time-sensitive programs in the field of real-time embedded systems. Instrumentation of a real-time program might cause it to exceed its time budget or deadline.

Changing the location of the instrumentation code in a program can have an effect on the extra-functional properties. Consider, for instance, the function in Listing 4.1. The function prints the value of z in the if and else statements, and prints w before returning. Calling this function 10 million times has an execution time of around 2.29 seconds on a 2.5GHz dual core platform. The printf() calls can be slightly modified by removing the calls at labels A and B, and printing both z and w in the call at label C. This reduces the execution time to around 1.75 seconds. Hence, the proper placement of instrumentation code can affect the performance and thus can help meet extra-functional properties like timing. Similar examples for other properties like binary size or communication channel throughput are straightforward.

```
   int simple(int x, int y){
      int z,w;
      if (x % 2 = 0){
         z = x + 100;
A:       printf("%d\n",z);
      } else {
         z = x * 9;
B:       printf("%d\n",z);
      }
      w = (z / 2) * y;
C:    printf("%d\n",w);
      return w + z;
   }
```

Listing 4.1: Simple C function for illustration

Maintaining an extra-functional property during instrumentation is complicated and managing multiple properties simultaneously is even more so. Extra-functional properties can be competing where meeting one property might break another. The instrumentation framework needs to weigh and trade off such competing properties. As an example, assume that the instrumentation framework can choose from several variables to instrument and locations in the source code where to instrument them. It might be the case that instrumenting, for example, six variables at one location minimizes the instrumentation time. However, at the same time, the system might have insufficient bandwidth to store and communicate the six variables at once and thus splitting up the instrumentation would be favorable. At that point minimizing both execution time and memory bandwidth is impossible.

We present a static instrumentation framework that gives the developer unprecedented control over what to instrument and what to preserve [62]. It thereby presents the first fully-implemented instrumentation mechanism that considers multiple competing extra-functional properties. INSTEP uses trees to represent instrumentation intents (IIs) and automata to represent cost models. The work provides insight into pruning the search space of instrumentation alternatives to find a feasible instrumentation. The experiments demonstrate the usage of IIs and cost models together with four different constraints and objectives. We experimented with multiple benchmarks as well as an industrial automotive module. The experimental results show the accuracy of INSTEP in honoring constraints and demonstrate its practicality and scalability.

The framework is directly applicable to a variety of use cases, including debugging and testing. In testing and oracle selection, the number of test inputs required to achieve a certain level of fault finding can be reduced through selecting an oracle that has a higher percentage of internal program variables [135]. However, increasing the number of internal variables used by the oracle, increases perturbation to extra-functional properties which may lead to violation of some constraints. Therefore, it is essential to choose an oracle which reduces test cases but at the same time preserves constraints. For security research, malware detection software uses instrumentation to identify malicious behavior. Instrumentation, however, introduces abnormal latencies in parts of the code. Malware may thus use the real-time clock to determine whether it is being monitored, and will stop any malicious activity as a precautionary measure, if it suspects so [54]. Emerging malware has

also begun embedding complex evasion techniques to detect monitoring environments as a means to protect itself from being discovered [24]. For instance, W32/MyDoom [91] and W32/Ratos [147] adopt self-checking and code execution timing techniques to determine whether they are under analysis or not. Hence, preserving timing while instrumenting code for malware detection can increase the efficiency of detecting malicious behavior.

4.1 Extra-Functional Instrumentation Overview

INSTEP is a static instrumentation framework that considers multiple competing extra-functional properties while instrumenting a program. Developers start by specifying their instrumentation intents. IIs specify variables of interest, their weights, and logical relations among them. Section 4.2 explains IIs in more detail. The II specification allows the framework to extract at run-time the information which is most valuable to the developer. INSTEP also permits developers to specify cost models for the different extra-functional properties of interest. A cost model is a weighted automaton that assigns different costs to actions like variable instrumentation and variable bit-width assignment. This helps INSTEP to maximize or minimize certain properties and satisfy constraints on others.

Figure 4.1 shows the block diagram for INSTEP. The framework operates in two phases: (1) the partial program derivation phase and (2) the determinising instrumentation phase. In the partial program derivation phase, INSTEP transforms the input program into a partial program based on the IIs. A partial program is one containing non-deterministic choices which have to be resolved [149].

The partial programs in INSTEP contain possible alternatives of where to execute the instrumentation. In the determinising instrumentation phase, INSTEP transforms the non-deterministic partial program into a deterministically instrumented program. This transformation is based on cost models for competing extra-functional properties together with any constraints on any property. In this phase, the framework attempts to find a feasible solution that satisfies all constraints and maximizes the objectives (or minimizes them based on their semantics).

Separating the framework into two phases fosters modularity and reuse. The use of partial programs as an intermediate representation shows promise [149] and allows for the modularity of the design to support extensions like different language processors or different back-ends. One possible extension, for example, is generating multiple instrumented programs that cover all the IIs (in case one is not enough) and at the same time honor constraints. Another reason is re-using the partial program for instrumentation after, for example, relaxation of cost models or constraints.

The development of the framework involved solving a set of challenges that are specific to its two phases of operation. Section 4.2 discusses the partial program derivation phase. It specifies the instrumentation intent representation and addresses the first challenge of deriving instrumentation alternatives from the IIs to create a partial program. Section 4.3 discusses the determinising instrumentation phase. It describes the representation of automata-based cost models and gives examples of various extra-functional

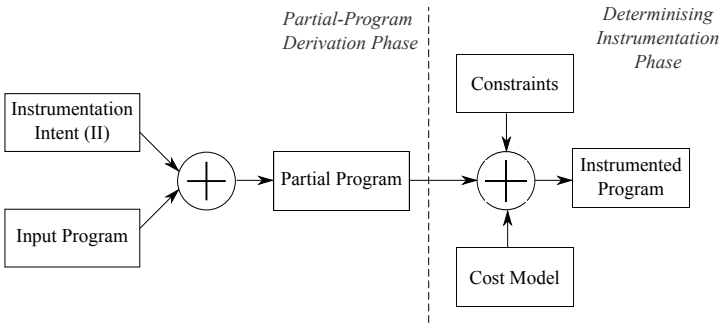


Figure 4.1: Extra-functional instrumentation framework

properties. It also addresses the second and third challenges. The second challenge is pruning the search space of the partial program. The third challenge is the formulation of an optimization problem from the cost models, the constraints, and the partial program. Solving this optimization problem yields a deterministically instrumented program.

4.2 Partial Program Derivation

In this phase, INSTEP uses two inputs: (1) the input program and (2) the instrumentation intents (IIs). With these, INSTEP extracts the program's OSCCFG and generates instrumentation alternatives based on the IIs. This section describes the inputs and the generation process in detail.

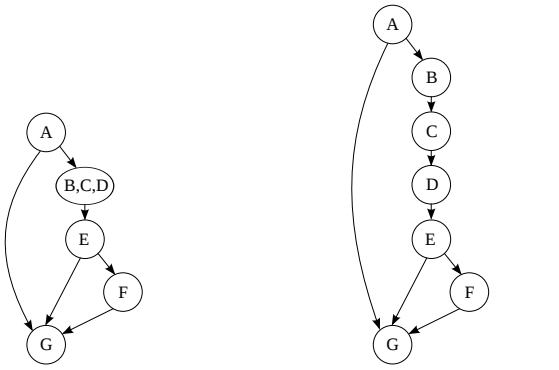
4.2.1 The Input Program

INSTEP uses CIL [101] to extract the input program's CFG. INSTEP supports data structures in MISRA C [94] compliant programs (more details in Section 4.5). INSTEP supports advanced constructs such as nested statements and recursive functions. We use the same program model as in Section 3.2. We use an OSCCFG such that each basic block contains at most one assignment to any variable of interest. Thus, if a basic block in the original CFG contains two assignments for variables mentioned in the IIs, then our model will split it into two basic blocks.

Listing 4.2 shows a sample input code which is part of the *sqrt* benchmark from the SNU benchmark suite [3]. Figure 4.2a shows the CFG for the input program before splitting any basic blocks. Basic block $\langle B, C, D\rangle$ contains all three statements B, C, and D. Assuming that the IIs contain the variables *dx, x, val, diff, flag*, Figure 4.2b shows the modified CFG after splitting basic block $\langle B, C, D\rangle$ into three separate basic blocks B, C, and D. INSTEP uses this generated OSCCFG for its transformations.

```
A: if(!flag){
B:     dx = (val-(x*x))/(2.0*x);
C:     x = x + dx;
D:     diff = val-(x*x);
E:     if(fabs(diff) <= min_tol){
F:         flag = 1;
       }
   }
G:
```

Listing 4.2: Sample input code



(a) Original CFG (b) Generated OSCCFG

Figure 4.2: Input programs

4.2.2 The Instrumentation Intent

The input IIs in INSTEP represent a set of required instrumentations specified by the developer. An II follows a tree structure specifying variables of interest specific to this particular II and values representing the importance of these variables. The tree of an II specifies a logical relation between the variables. For example, consider statement B in Listing 4.2. If a developer wants to trace variable dx, he might be interested in either variable dx or variables val and x. This intent has the following propositional logic expression: $(dx \vee (val \wedge x))$. As Figure 4.3a shows, the II for this statement consists of two branches where the AND-ed variables lie on the same branch, and the OR-ed variables lie on different branches. The developer assigns the values based on the importance or usefulness of the variables. The particular II in Figure 4.3a only uses values 1, 0, and 1 for the variables dx, val, and x, respectively. This encodes that variable val alone is useless without variable x, and that variables val and x have an equal value to dx. Figure 4.3 shows the IIs for the statements B, C, D, and F of the input program in Listing 4.2.

A node in an II tree can contain more information than just a variable's name. For example, in Figure 4.3b INSTEP requires a separation in the II between variable x on the left hand side (LHS) of statement C and variable x on the right hand side (RHS). Line numbers are also required to identify the locations of the variables.

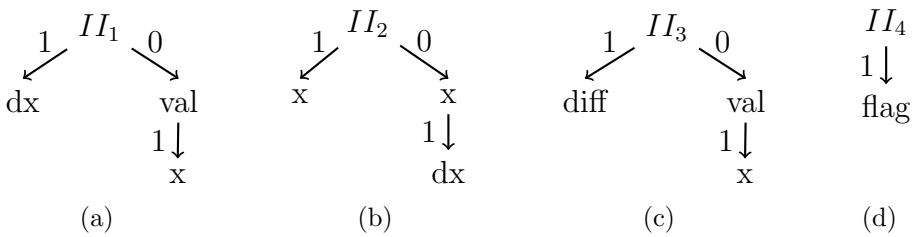


Figure 4.3: Instrumentation intents for the input program

II Specification: An II can originate from different sources. During a debugging session, the developer will most likely specify the II. A simple tool based on program slicing could generate IIs from a high-level specification. If variables are equally important, the developer can leave the II values at their defaults as in Figure 4.3a. Testing tools and tracing tools can also generate the IIs based on a high-level specification [100]. The results show that our framework is robust and tolerates inaccuracies in the model.

4.2.3 The Derivation of the Partial Program

After extracting the input program's OSCCFG and parsing the IIs, INSTEP finds instrumentation alternatives for the different variables in the IIs. An instrumentation *alternative* is one or more locations in the code where a variable can be instrumented to extract its desired state before it changes. Normally, an instrumentation alternative is a single basic block (one location) at which a variable can be instrumented (recall that each basic block contains at most one assignment to any variable of interest). An alternative, however, can be multiple basic blocks (more than one location). Consider the example in Figure 4.4 which shows a code snippet and its corresponding OSCCFG. The variables of interest are x, w, and z. One instrumentation alternative of variable x is at the end of block J (one location). Another alternative, can be after the if-condition (block K) and before the branch sink P. So, for instance, blocks L and M (together) can be instrumented to cover all sub-paths between nodes K and P and provide a valid instrumentation alternative. Therefore, if an alternative is comprised of multiple basic blocks, this means that all these blocks have to be instrumented to represent a valid instrumentation of the variable.

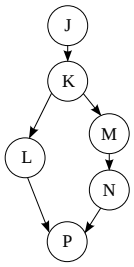
The instrumentation engine finds locations in the OSCCFG that permit instrumentation. The engine coarsely follows the following rules:

1. **Variable is on the LHS of a statement:** INSTEP inserts instrumentation alternatives at the current basic block (containing the statement) and at all the following blocks until the variable's value is overwritten.

2. **Variable is on the RHS of a statement:** INSTEP inserts instrumentation alternatives starting at the block containing the last variable's assignment prior the current basic block and until the next block that overwrites the variable's value.

```
J: x = 2;
K: if(y > 3){
L:     z = x*4;
   } else {
M:     w = x*3;
N:     z = w + 4;
   }
P:
```

(a) Program's code



(b) Program's OSCCFG

Figure 4.4: Example to illustrate instrumentation alternatives

3. **Variable is on both sides of the statement:** (as in statement C of Listing 4.2) INSTEP inserts instrumentation alternatives starting at the block containing the last variable's assignment until the block containing the statement of interest (alternatives for the RHS). It also inserts alternatives starting from the current assignment of the variable until the variable's value gets re-assigned (alternatives for the LHS).

Note that these are only the coarse rules used by the engine and that the implementation contains more detailed rules. The exact locations of the alternatives depend on the type of the basic blocks, branches in the OSCCFG, etc. Consider a variable *var* that is assigned in some basic block B. To fulfill the aforementioned rules, the instrumentation engine should be able to traverse the OSCCFG upwards (towards the root) and downwards (away from the root) from B to find instrumentation alternatives for *var*. Function 4 briefly describes how the instrumentation engine finds alternatives moving downwards from B. Function 5 outlines how the engine finds alternatives moving upwards from B. Function 5 frequently calls the Function 6 to instrument the parent(s) of a basic block. To illustrate how the instrumentation engine operates, we describe the operation of Function 4 along with a few cases out of around 366 different cases that INSTEP covers.

Functions 4 and 5 take as input: the OSCCFG G, the variable *var* for which instrumentation alternatives are to be found, and the basic block B in which *var* exists. The OSCCFG can have blocks of the following types: branch (if or switch statement) which has more than one child, loop-start, loop-break, return (a return statement), and instruction (instructions with no branches). The function *addAlt* adds the start or end of a basic block as an instrumentation alternative. Recall that an alternative can be multiple basic blocks. The proposed algorithms may also find alternatives for one of the multiple blocks that form an alternative. For instance, consider the example in Figure 4.4. If Function 4 is finding alternatives for variable x at block J, then after storing blocks L and M as one alternative, it will store N as an alternative for M. Hence, when storing an alternative, function *addAlt* keeps track of which block the alternative is for (details are omitted from the algorithm for clarity). The *enqueue* operation used in the algorithm, would only enqueue a block if it was not enqueued before.

Function 4 starts by instrumenting the end of the input block B and enqueuing its child (an instruction block always has one child). The dequeued block is handled according to

Function 4 Find Alternatives Downwards

Input: OSCCFG G, instrumentation variable *var*, basic block B
Output: instrumentation alternatives

```
 1: let Q be an empty queue
 2:
 3: call addAlt(end of B)
 4: enqueue B's child in Q
 5: while Q is not empty do
 6:    dequeue C from Q
 7:    if C is a loop-break block or a return block or an ancestor of B then
 8:       do nothing
 9:    else if C is a loop-start block then
10:       let D be the loop-break block
11:       if the loop does not modify var then
12:          enqueue D's child in Q
13:       end if
14:    else if C is an instruction block and does not modify var then
15:       call addAlt(end of C)
16:       enqueue C's child in Q
17:    else if C is a branch source then
18:       let D be the branch sink
19:       if D exists and var is not modified between C and D then
20:          enqueue D in Q
21:       end if
22:       if D does not exist or D is not C's child then
23:          call addAlt(start of C's children)
24:          enqueue each of C's children
25:       end if
26:    end if
27: end while
28: return instrumentation alternatives
```

Function 5 Find Alternatives Above

Input: OSCCFG G, instrumentation variable *var*, basic block B
Output: instrumentation alternatives

```
let Q be an empty queue

call addAlt(start of B)
call InstrumentParents(G, var, B, Q)
while Q is not empty do
    dequeue C from Q
    if C is a loop-start block then
        do nothing
    else if C is an instruction block then
        call addAlt(end of C)
        if C does not modify var then
            call InstrumentParents(G, var, C, Q)
        end if
    else if C is a loop-break block then
        let D be the loop-start block
        if the loop does not modify var then
            call InstrumentParents(G, var, D, Q)
        end if
    else if C is a branch source then
        call InstrumentParents(G, var, C, Q)
    end if
end while
return instrumentation alternatives
```

Function 6 Instrument Parents

Input: OSCCFG G, instrumentation variable *var*, basic block C, queue Q

```

if C is a branch sink then
    let D be the branch source
    if var is not modified between C and D then
        enqueue D in Q
    end if
    if D is not one of C's parents then
        call addAlt(end of C's parents)
        enqueue each of C's parents
    end if
else if C has one parent then
    if C's parent is a branch source then
        call addAlt(start of C)
    end if
    enqueue C's parent in Q
else
    call addAlt(start of C)
end if
```

its type. We describe a few cases that explain the operation of Function 4. Consider, for example, Listing 4.2 and its OSCCFG in Figure 4.2b. If INSTEP is finding instrumentation alternatives for variable *diff*, then the first instrumentation alternative will be the end of basic block D. Normally, children of an if-block are an alternative as well, however, in this case, the if-block E has a branch sink G which is also one of its children. Instrumenting the child F alone is not an alternative, because it will leave subpath $\langle E, G \rangle$ without instrumentation. Hence, INSTEP bypasses the if-block and chooses its sink G as an alternative. Note that this will only be possible, if no subpath modifies the variable *diff*. Another example is: if INSTEP finds the start of a loop, then INSTEP will continue finding alternatives following the break of the loop only if the variable is not modified inside the loop. A third example is: if the algorithm encounters a loop-break (without first encountering a loop-start), this means that block B is inside a loop. The algorithm, in that case, will not find alternatives beyond the loop break because information is missed by instrumenting outside the loop.

Function 5 operates in a manner similar to that of Function 4. It starts by instrumenting the start of the input block B and calls Function 6 to instrument the parents of B. If, for example, B is a branch sink, then the corresponding branch source will be enqueued for instrumentation as an alternative only if *var* is not modified in any branch between the source and sink blocks. Another example is: if INSTEP finds the break of a loop, then INSTEP will continue finding alternatives before the start of the loop only if *var* is not modified inside the loop.

4.2.4 The Partial Program

The partial program is an intermediate representation that contains all possible instrumentation alternatives. The derivation phase of the framework inserts the instrumentation alternatives in the input program to generate the partial program. Listing 4.3 shows the partial program for the input program in Listing 4.2 after INSTEP inserted the instrumentation alternatives. Note that the notation in the listing is only for illustration purposes, because an instrumentation alternative must hold more information. For example, the framework needs to know which alternatives must simultaneously exist if on parallel branches, for instance. For the statement C, Listing 4.3 uses $x.l$ and $x.r$ to differentiate between the left and right x variables, respectively. (II_1, val), for example, represents a location where the variable *val* from II_1 can be instrumented.

```
   (II1,val),(II1,x),(II2,x.r),(II3,val)
A: if(!flag) {
      (II1,val),(II1,x),(II2,x.r),(II3,val)
B:    dx = (val-(x*x))/(2.0*x);
      (II1,dx),(II1,val),(II1,x),(II2,x.r),(II2,dx),(II3,val)
C:    x = x + dx;
      (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,val),(II3,x)
D:    diff = val-(x*x);
      (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,diff),(II3,val),(II3,x)
E:    if (fabs(diff) <= min_tol) {
F:       flag = 1;
         (II4,flag)
      }
   }
G: (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,diff),(II3,val),(II3,x),(II4,flag)
```

Listing 4.3: Partial program

4.3 Determinising the Instrumentation

In this phase, INSTEP processes three inputs: (1) the partial program from the first phase, (2) constraints on the instrumentation, and (3) cost models for instrumentation methods. INSTEP uses these three inputs to formulate an optimization problem and attempts to solve it using local searching [14] to find a feasible solution. A feasible solution is a selection of the instrumentation alternatives that satisfies the constraints, and maximizes or minimizes other objectives that may exist. This section describes the constraints, the cost models, the formulation of the optimization problem, and the final output of INSTEP.

4.3.1 Specifying the Constraints

Constraints are restrictions on the instrumented program that might prevent it from achieving the maximum value of the instrumentation intents or the highest output for any other objective. Constraints primarily pose limits on some extra-functional properties. For example, one constraint may be a limit on the code size of the program after instrumentation. Another constraint might be a limit on the memory consumption while running an instrumented program. Finally, an upper limit of the debugging budget added to the WCET for the instrumentation [39, 63, 68] is another form of constraint. Enforcing such constraints on the instrumentation process requires knowledge of the cost functions. A cost model specifies the cost for the different aspects of instrumentation. Modern modeling systems like UML/MARTE and AADL facilitate specifying the different constraints and cost models by the developer.

4.3.2 Cost Models

In this context, cost models are simply weighted automata that describe costs of actions. The development of the cost models themselves is out of the scope of this work. Figure 4.5

shows two cost models for instrumentation points. Figure 4.5a represents a code size cost model for adding a printf() instrumentation point for integers on an ARM Cortex-M3. The cost for the first variable is 14 bytes of code and 8 bytes for each extra variable that can be added in the same printf() statement. The first variable has a cost of 14 because it includes instructions for a function call which are added once for an instrumentation point of this type. Figure 4.5b shows a code size cost model for writing the instrumented variable to a buffer array using GCC on an Intel Core i5-2520M CPU. Each variable would add 32 bytes of code to the program.

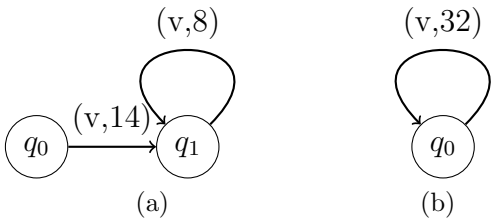


Figure 4.5: Cost models

The values in a cost model depend on many factors. Example factors include the used hardware, the type of instrumentation (e.g., printf() to the serial port of a chip, saving to a buffer, sending over TCP/IP), and the type of variable (e.g., integer, double, character). There might also be an overhead cost for including a library required for instrumentation (to use the network stack for example). The use of automata provides a general concept that, for instance, also supports extending INSTEP to consider caches and costs of reading from cache versus main memory [149]. This work considers cost models for code size of instrumentation points, timing of instrumentation points, and detection latency. A cost model for the timing of a printf() instrumentation point may look like that of code size. Detection latency is the latency between assigning the variable and instrumenting it.

Accuracy of cost models: Cost models are widely used to estimate performance costs of certain operations such as writing to memory and data transmission [149, 142]. It is clear that adding cost models for all details of the target architecture complicates the analysis but provides more precise results. We assess the effect of using inaccurate cost models on the output of INSTEP. This is discussed in more detail in Section 4.4.

4.3.3 The Formulation

INSTEP combines the cost models, the constraints, and the partial program into an optimization problem. In its current form, INSTEP supports four main extra-functional properties: II values, code size, execution time, and detection latency. Each of these can be used in objectives and constraints. For example, a developer can choose to minimize code size or set an upper limit as a constraint. It is easy to extend INSTEP to consider types of variables, different instrumentation types, and other properties such as memory

consumption and tracing bandwidth (for TCP/IP). The formulation in the examples below uses printf() to a serial port as the instrumentation type.

Decision Variables

The framework creates a boolean variable for each instrumentation variable in the IIs indicating whether the variable is instrumented or not. It also creates a boolean variable for each instrumentation alternative to indicate whether the alternative is chosen or not. Variable *flag* in Listing 4.3, for example, has two instrumentation alternatives so the framework will create three variables; *II4_flag*, *II4_flag_F_e*, and *II4_flag_G_s* denoting the instrumentation of variable *flag*, instrumenting *flag* at the end of node F, and instrumenting *flag* at the start of node G, respectively. For each variable, the framework creates the following constraint:

$$-\ var + \sum alternatives \geq 0$$

This encodes that the variable is instrumented only if one of its alternatives is chosen. For variable *flag*, in Listing 4.3, the constraint will be:

$$-\ II4\_flag + II4\_flag\_F\_e + II4\_flag\_G\_s \geq 0$$

II Values Property

INSTEP will maximize the total value of all IIs or meet a minimum value, if the developer specifies a constraint. The value of an II is the maximum of the values of all variables in an II tree. The value of a variable is the summation of the values on the path leading to the variable's node in the II tree from its root. This value of a variable is only realizable if it is instrumented and all its ancestor variables are instrumented as well. If a variable exists more than once in a tree, the maximum of these values is taken. For example, considering II_1 in Figure 4.3a, its value is equal to:

$$V1 = max(II1\_dx * 1, II1\_val * 0, II1\_val * II1\_x * (0 + 1))$$

This shows that instrumenting variable *val* alone is useless and also reflects that variable x is useful only if *val* is instrumented as well.

Code Size Property

For both timing and code size, cost models specify the costs of instrumentation for these two extra-functional properties. Hence, each instrumentation point that INSTEP inserts in the code will have a cost that needs to be considered. For example, consider the instrumentation point at the start of node A in Listing 4.3. It has four variables available for instrumentation. To formulate the cost in terms of code size, consider the cost model in Figure 4.5a. Simply, if any of the variables is instrumented at that point, a cost of 14 bytes will be incurred and 8 bytes for each extra variable. This can be formulated as follows,

taking into account that a variable can appear more than once at the same instrumentation point in a partial program:

$$code\_A\_s = (14 - 8) * (II1\_val||II1\_x||II2\_x.r||II3\_val) + 8 * (II1\_val||II3\_val) + 8 * (II1\_x||II2\_x.r)$$

This computes the extra overhead for the first variable of the instrumentation point if any variable is instrumented. It also incurs the cost of a variable only once if it exists in multiple IIs. INSTEP formulates a similar cost function for each instrumentation point with regards to timing.

The framework represents the total code size after instrumentation as the total of (1) original code size, (2) any overhead for using printf() instrumentation (cost incurred if any instrumentation point exists), and (3) the cost of all instrumentation points with respect to code size. The developer can specify minimizing code size as an objective or set a limit on the total code size that should not be exceeded.

Execution Time Property

As for timing, to respect a given debugging time budget, INSTEP requires knowledge of the WCET of the program, the WCET of the different basic blocks, function calls, and the worst-case number of executions of each basic block. This work assumes the presence of correct but maybe conservative WCET analysis tools. INSTEP extracts function calls through static analysis, and obtains all other information through the RTBx data logger and RapiTime [2], the measurement-based WCET analysis tool. The timing of the main function of the program from start to end, after instrumentation, can be either minimized as an objective or be constrained with a developer-specified debugging budget. To formulate the effect of instrumentation on the timing of the code, INSTEP formalizes the cost of a function as the maximum timing of all paths in the function. This formulation is a conservative approximation as it ignores cache effects, branch prediction, etc. Taking the maximum of all paths requires enumerating all paths which is exponential. Therefore, INSTEP traverses the OSCCFG of a function and instead of enumerating paths, it takes the maximum of subpaths from a branching source node to its sink. This is a practical over-approximation and worked well in the experiments.

INSTEP also prunes paths according to the following rules:

- If the subpaths between a branch source and sink, do not have instrumentation points or function calls, INSTEP will prune the max function to the subpath with the largest timing.
- If a subset of the subpaths has instrumentation points and/or function calls, INSTEP will only consider this subset along with the largest timing subpath.

INSTEP can further prune the OSCCFGs (through abstracting them for example), but this will only be effective for complex timing cost models that include more architecture-related information.

The cost of a function is equal to the cost of its basic blocks multiplied by the number of executions of the blocks and taking the maximum of subpaths in case of branches. The cost of a block is equal to the WCET of the block, the WCET of any instrumentation point in the block (from the cost model), and any function calls in the block. The cost of a function call is simply equal to the formulated cost of the called function. Note that the cost of calling and returning from a function is already part of the WCET of the basic block which includes the function call. The cost of the function in Listing 4.2 and Figure 4.2b would be:

$$\begin{aligned} func = {} & (A + t\_A\_s) * W_A * I_A + (B + t\_B\_s + t\_B\_e) \\ & * W_B * I_B + (C + t\_C\_e) * W_C * I_C + (D + t\_D\_e) \\ & * W_D * I_D + E * W_E * I_E + (F + t\_F\_e) * W_F * I_F \\ & + (G + t\_G\_s) * W_G * I_G \end{aligned}$$

where t_B_s and t_B_e, for example, are the costs of the instrumentation points at the start and end of node B, respectively. In practice, the path $\langle E, G\rangle$, for example, might be worse, with respect to timing, than the path $\langle E, F, G\rangle$; however, this is an approximation that INSTEP uses in its current form. If the WCET of the instrumented program exceeds the specified debugging budget, INSTEP will find the instrumentation point causing the violation, remove the instrumentation point and rerun the analysis to ensure that constraints are met. INSTEP will detect a violation, if after rerunning the WCET analysis, the WCET of a block exceeds the expected increase according to the cost models. The experiments show that the violations do not often occur, and if they do, the number of retries is low. This is because INSTEP can use one WCET analysis report (provides WCET of basic blocks) to detect multiple violations of the instrumentation process.

Detection Latency Property

The detection latency of a variable is the minimum of the detection latencies of its instrumentation alternatives. A function like $1/x$ can represent this property where x is the amount of time from the specified variable location until its instrumentation. Since the instrumentation is at the granularity of the basic blocks, the detection function will only be defined at possible instrumentation locations (start and end of blocks). A developer's goal might be minimizing latency, i.e., maximizing the detection function or specifying a constraint on the detection latency. Several paths might exist between a statement and the instrumentation alternative. In such case, the maximum latency of the different subpaths is chosen. If a variable has ancestors in the II, then the detection latency is the maximum of its detection latency and latencies of all its ancestors. The detection latency of an II is the minimum across all its nodes.

4.3.4 The Instrumented Program

After INSTEP formalizes the optimization problem, it uses local search to find a feasible solution. Local search is used because the problem is highly non-linear with a large number

of decision variables resulting from the instrumentation alternatives. Such combinatorial model is out of the scope of current state-of-the-art solvers relying on classical tree-search techniques [14]. Local search attempts to find candidate solutions by applying local changes in the search space. The solution would be an assignment to the defined boolean variables which can be easily used to transform the partial program into an instrumented program. Local search either returns an infeasible solution, if the constraints can never be satisfied, a feasible solution, or an optimal solution. A solution is infeasible if, for example, the specified constraint for code size is below the original code size. If the input constraints are at least equal to the corresponding values of the input program, i.e., if, for example, WCET constraint is at least equal to the input program WCET, then local search will always find a feasible solution.

For our experiments, we used the standard setup for local search and it worked reasonably well as Section 4.4 demonstrates. The pseudo-random number generator seed is set to zero. The simulated annealing level is set to one. The search is parallelized over two threads. The experimental results show the applicability and practicality of our approach. Finding the best configuration parameters for local search is out of the scope of this work.

4.4 Experimentation

This section presents experimentation using the fully automated framework INSTEP.

4.4.1 Experimental Setup

There are three sets of experiments:

1. We experiment with the SNU real-time benchmark suite [3]. It contains 17 C benchmarks that have 117 lines of code on average, and implement numeric and DSP algorithms.

2. We run an experiment on the web server example [1] for NXP LPC17xx ARM-based micro-controllers. This program implements a dynamic web server and has a total of 1,846 lines of C code.

3. We conduct an experiment on an automotive control module. It has 177,298 lines of C code and 6,297 basic blocks. This number excludes definitions in header files, since the industrial partner provided only parts of the overall application. Consequently, the experiments on this program only show the scalability of INSTEP and applicability to industrial code, without showing the results on the WCET analysis.

The benchmarks were run on a Keil MCB1700 board running an NXP LPC1768 MCU which is a 100 MHz ARM Cortex-M3 microcontroller. This 32-bit Microcontroller has an MPU, 512kB on-chip Flash ROM, and 64kB RAM. Section 3.4.4 describes the hardware and the on-board peripherals in more detail.

Metrics

We quantitatively evaluate the accuracy and precision of the framework using the following metrics:

- **WCET of the instrumented program:** We run a WCET analysis for the instrumented program. The WCET should be less than or equal to the input program's WCET plus a specified debugging budget β.
- **Code size of the instrumented program:** The code size of the instrumented program should be less than or equal to the constraint set on the program size after instrumentation.
- **Satisfaction of IIs:** This metric is a measure of the percentage of satisfied IIs for each benchmark. This metric is an indicator of how much instrumentation coverage INSTEP can achieve while obeying all the constraints.
- **Number of retries:** If the WCET of the instrumented program exceeds the input program's WCET plus the debugging budget β, INSTEP will remove the instrumentation points that cause a higher WCET than expected. Finding these instrumentation points is straightforward. The WCET analysis tool outputs the WCET of each basic block before and after instrumentation, and basic blocks that now violate the constraints are immediately visible. INSTEP then reanalyzes the WCET of the modified program. We report the number of retries required to produce the final instrumentation.
- **INSTEP execution time:** The size of the inputs (e.g., number of IIs, code size) can increase the time that INSTEP needs to generate the instrumented program. This metric indicates the applicability of INSTEP to large-scale software programs used in industry.
- **Number of instrumentation alternatives in the partial program:** As the code size of the input program increases, it becomes more challenging to derive an instrumented program honoring the extra-functional properties while maximizing objectives. A large program offers multiple locations for instrumenting a variable which also complicates the optimization problem.
- **Number of equations and expressions in the optimization problem:** This metric shows the complexity of the optimization problem required to instrument a large software program.

Extra-Functional Properties

The experimentation considers four extra-functional properties: the II values, code size, execution time, and detection latency. We use a `printf()` to the serial port of the microcontroller for instrumentation. Thus, the code size cost model is the one shown in Figure 4.5a. The cost for the first variable is 14 bytes of code and 8 bytes for each extra

variable that can be added in the same printf() statement. Additionally, there is an overhead cost of 460 bytes for including the library required for instrumentation. A cost model for the timing of a printf() instrumentation point is similar to that of code size but with different values. The first variable in a printf() statement costs 4,000 cycles, and each extra variable in the same printf() statement costs 3,850 cycles as measured on the target platform. Finally, the function $1/x$ represents the cost model for detection latency. Execution time and code size properties are considered constraints to the optimization problem. We limit the debugging budget (constraint) by a 10% increase in the WCET of the input program [35]. The code size constraint is arbitrarily chosen to be an additional 554 bytes to the input program size. 554 bytes are the size of the input library plus five separate printf statements where each instruments a single variable. The optimization has two objectives: maximizing II values and minimizing the detection latency.

Choice of IIs

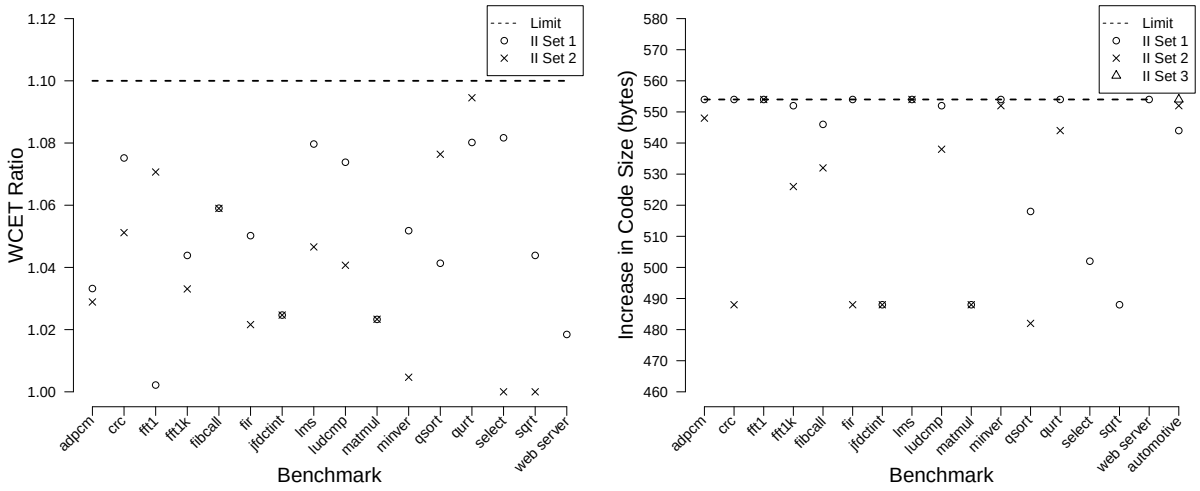
Each SNU benchmark is run with 1,100 different inputs and has two different sets of IIs: (1) a maximum of 30 input IIs, and (2) a maximum of four IIs. The IIs were randomly chosen to avoid any bias in the experimental results. Note that only six out of 15 benchmarks had enough variables to form 30 IIs. For the rest of the benchmarks and for the first set of the IIs, the maximum number of IIs available for each benchmark was input to INSTEP. The web server experiment has 79 IIs as input. This is the maximum number of IIs available in the web server software. The automotive module experiment has three versions. The objective of the first version is to instrument all assignments of an arbitrary local variable in a function, resulting in nine IIs. The second version instruments the five most occurring global variables in the program, which is equivalent to 54 IIs. Whereas the third version instruments the five most occurring local variables across all functions represented in 21 IIs.

4.4.2 Experimental Results

The results of the different sets of experiments show that instrumented benchmarks do not violate any of the constraints. The results also show that INSTEP satisfies more IIs compared to a naive instrumentation. They also demonstrate the scalability and applicability of INSTEP to industrial software.

Figure 4.6a shows the ratio of the WCET of the instrumented benchmark to that of the input. Figure 4.6b shows the increase in code size of the instrumented benchmarks. Benchmarks *select* and *sqrt* have a WCET ratio of 1 and no increase in code size for II Set 2. This means that INSTEP did not instrument any variables so as not to violate any of the constraints. The figures show that the increase in both the WCET and code size are within the specified constraints. Note that benchmarks *bs* and *insertsort* are omitted from the results, because inserting any instrumentation point in any of them would violate a constraint. Hence, INSTEP left these two benchmarks intact without instrumentation.

Table 4.1 shows the WCET, the code size, and the lines of code of the benchmarks before instrumentation. For each set of IIs, the table shows for each benchmark: the number of IIs satisfied from the input ones, the execution time of INSTEP, the number of instrumentation alternatives, the number of equations and expressions, the number of retries. In what follows, we draw conclusions from each of these metrics.



(a) WCET ratio

(b) Increase in code size

Figure 4.6: WCET ratio and increase in code size of instrumented benchmarks

INSTEP vs Naive Instrumentation

The code size constraint was set such that it comprises five printf() statements with a single variable each. This means that a naive instrumentation will most probably satisfy five IIs at most. Table 4.1 shows that, out of 34 experiments, there were 17 experiments with more than five IIs. In 14 out of these 17 experiments, INSTEP satisfied more than five IIs with a maximum of 26 for the automotive module. This indicates the strength of the framework in finding alternatives and merging them to satisfy the most IIs and honor constraints. Figure 4.7 also shows the percentages of the input IIs that INSTEP was able to satisfy. For some benchmarks, the satisfied number of IIs in the second set is less than four, while being much larger for the first set. This depends on which subset of IIs from the first set are chosen for the second. It might be the case that the chosen subset of IIs for the second set, violate constraints. In such case, INSTEP does not satisfy any of the IIs as it is the case with *select* and *sqrt* benchmarks.

Retries

Table 4.1 shows that retries were required in 16 out of 38 experiments. In most of the cases, one retry was required and at most four retries were needed with an average of 0.85 retries. The number of retries is low due to the ability of INSTEP to find (and remove)

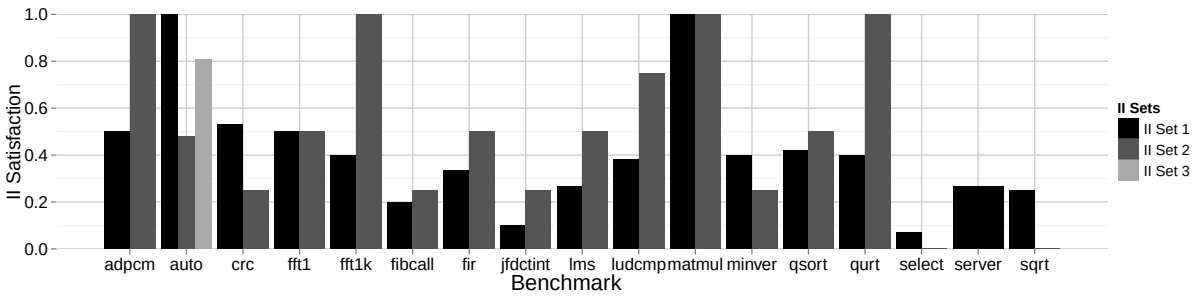


Figure 4.7: Satisfaction of instrumentation intents

multiple instrumentation points causing a violation to the debugging budget using one WCET analysis report. The reason is that one WCET analysis report is sufficient to detect violations in each basic block of the program's OSCCFG. The low number of retries shows the practical feasibility and viability of INSTEP even for large programs.

Execution Time

Table 4.1 shows the execution time of INSTEP. The execution time increases as the input program has a more complicated or larger OSCCFG, more IIs, etc. The execution time has a reasonable average of 2.64 seconds with a maximum of 32 seconds for instrumenting the automotive module. The reported time does not include the time required for applying local search. Figure 4.8 shows the time required by local search to find the best reported feasible solution within 10 minutes. In 30 out of 38 experiments, local search found the solution in only one second. All other times in the figure appeared only once with a maximum of 261 seconds. This shows that a satisfactory solution can be found in a reasonable amount time.

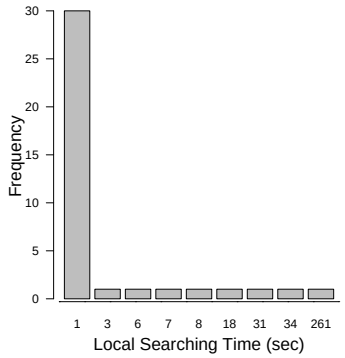


Figure 4.8: Local search time

Instrumentation Alternatives, Equations, and Expressions

Table 4.1 also shows the instrumentation alternatives, number of equations, and expressions for each experiment. The number of equations and expressions in the optimization problem

exceeded 1,300 and 33,000, respectively which shows the complexity of the problem. The number of alternatives reached 3,041 which indicates the efficiency of the tool in finding multiple alternatives. It also shows that INSTEP scales and can accommodate this large number of alternatives within reasonable time limits. INSTEP also handles the large number of equations and expressions, and local search finds satisfactory solutions that satisfy a large number of IIs in an acceptable time frame.

Inaccuracy of Cost Models

To test the effect of inaccurate cost models on the output of INSTEP, the experiments were repeated using modified cost models for time and code size. Two modified versions of the cost models were used: (1) underestimated models which reduce the cost of transitions in the original time and code size models by 500 cycles and 2 bytes, respectively, and (2) overestimated models which increase the cost of transitions by the same values for both of the original models. Underestimating the cost model can lead to more violations which increase the number of retries. In 60% of the experiments, the number of retries did not change, and increased by only one retry in the rest of the cases. Overestimating the cost model might reduce the number of satisfied IIs. In 80% of the experiments, the number of satisfied IIs did not change, and decreased by only one otherwise. This shows that inaccuracies in the cost models can be tolerated by INSTEP which can still output satisfactory results.

4.5 Discussion

This section discusses some issues regarding the limitations and applicability of INSTEP.

Logical correctness: INSTEP preserves the logical (functional) correctness of a program after instrumentation. The only modification that INSTEP makes to a program is the insertion of instrumentation points. These instrumentation points only read variables from memory locations and there is no concurrent variable access. For all experiments, the outputs from the instrumented programs matched those of the uninstrumented programs.

MISRA C compliance: INSTEP supports instrumenting data structures in MISRA C [94] compliant programs which restricts, for instance, the usage of pointers and unions. MISRA C also restricts the usage of dynamic memory allocation (`malloc()`). Extending INSTEP to consider memory consumption of the software as an extra-functional property is therefore restricted to static memory allocation.

Concurrency: With the current WCET analysis tools in place, INSTEP only supports instrumenting foreground/background systems and multi-programming systems with run-to-completion semantics. Concurrency complicates computing the budgets and testing

Table 4.1: Experimentation results for INSTEP

Benchmark	WCET [cycles]	Code Size [bytes]	Lines of Code	II Set	IIs Met	Run Time [ms]	Alternatives	Equations	Expressions	Retries
Deployed Automotive Module	-	974,848	177,298	Set 1	9/9	9,084	34	162	3,873	0
				Set 2	26/54	31,152	1,944	1,349	12,664	0
				Set 3	17/21	12,836	271	304	5,194	0
Web Server	4.148×10^6	10,590	1,846	Set 1	21/79	29,090	3,041	733	33,788	0
adpcm	5.770×10^9	8540	522	Set 1	15/30	1,169	908	229	2,745	0
				Set 2	4/4	296	47	65	429	0
crc	9.280×10^6	1,100	72	Set 1	9/17	116	61	120	332	1
				Set 2	1/4	83	25	45	161	1
fft1	3.439×10^8	4752	146	Set 1	8/16	147	70	129	427	0
				Set 2	2/4	97	20	37	180	2
fft1k	2.106×10^{11}	5,256	92	Set 1	10/25	270	261	189	878	1
				Set 2	4/4	82	17	32	132	0
fibcall	3.941×10^5	608	32	Set 1	1/5	42	22	37	103	4
				Set 2	1/4	56	19	30	88	3
fir	2.791×10^7	6,428	176	Set 1	10/30	294	211	236	840	0
				Set 2	2/4	103	20	36	165	0
jfdctint	8.104×10^4	1,440	186	Set 1	3/30	1,132	1,912	278	4,030	2
				Set 2	1/4	86	42	36	159	2
lms	5.293×10^8	6,596	158	Set 1	8/30	219	156	171	673	1
				Set 2	2/4	120	41	50	258	2
ludcmp	2.686×10^9	4,084	82	Set 1	8/21	186	100	183	576	1
				Set 2	3/4	85	20	45	181	1
matmul	1.528×10^8	1,012	32	Set 1	2/2	64	6	16	63	0
				Set 2	2/2	52	6	16	63	0
minver	8.573×10^7	4,936	143	Set 1	12/30	433	217	233	989	0
				Set 2	1/4	128	15	32	207	3
qsort	1.696×10^5	1,936	83	Set 1	8/19	227	200	131	691	1
				Set 2	2/4	87	32	33	174	1
qurt	1.751×10^6	4,148	95	Set 1	12/30	1,376	909	213	2,874	1
				Set 2	4/4	86	28	37	189	0
select	8.618×10^4	1,876	72	Set 1	1/14	180	104	104	444	2
				Set 2	0/4	102	46	43	229	0
sqrt	4.530×10^5	4,036	46	Set 1	2/8	81	42	66	185	0
				Set 2	0/4	73	27	41	138	0

whether a block will exceed its budget; however, the underlying concepts still apply and can be extended given the available tools.

Hardware tracing: One limitation on hardware tracing is that some systems do not support hardware debugging. Hardware tracing also offers traces at a low system level, e.g., instruction level. This makes software tracing more suited to debugging at a higher object level, e.g., debugging a task control block. Another aspect is the evolution of the tracing mechanism along with the software being debugged. If the software is modified or updated, a software tracing mechanism can be easily maintained along with it, as opposed to a hardware tracing module or device.

Partial tracing information: In this work, the examples and experiments focus on tracing data variables. Similarly, INSTEP can trace control flow and function calls. INSTEP focuses on extracting information while preserving the input program's extra-functional properties. This definitely limits the amount of information that INSTEP can extract from the program. INSTEP, however, attempts to maximize the satisfied IIs as the experiments demonstrate. Note also that partial traces are useful for analyzing and understanding programs, as well as for optimizations [123, 114]. Moreover, INSTEP is easily extensible to generate multiple instrumentations of the same input program to satisfy all IIs (if possible). This allows extracting more tracing information but from different program runs (which is a limitation to the debugging process).

4.6 Summary

Current tracing and instrumentation tools only preserve functional correctness of the program. Unfortunately, some application domains require tools that not only consider the functional correctness, but also consider extra-functional properties such as timing. We propose INSTEP; an instrumentation framework that preserves extra-functional properties. To generate the instrumented program, INSTEP derives a partial program based on the developer's II. Then, it formulates an optimization problem according to the input cost models and constraints, and solves the problem using local search. The design of INSTEP allows for the re-usability of partial programs and for future extensions. INSTEP, in its current state, honors four extra-functional properties. We conducted experiments on benchmarks as well as an industrial automotive module. The experimental results show the accuracy and precision of the tool in honoring constraints. They also show the practicality and scalability of INSTEP.

Chapter 5

Stage-Level Analysis for CMPs

The FLA proposed by Shi and Burns [126, 128] assumes a communication model where the tasks are indivisible units of communication. As a consequence, FLA does not incorporate the effects of pipelining and parallel transmission of data in its communication model. This restriction in the model results in higher upper-bounds on the communication latencies.

We address this issue by proposing a pipelined communication resource model for analyzing the WCLs for hard real-time systems. This model supports pipelined and parallel transmission of data over communication resources with fixed priority preemption. We also present an associated analysis, SLA, that uses the pipelined communication resource model to produce tight WCL estimates [64]. The model supports communication tasks that are either periodic or sporadic. This analysis is suitable for interconnects that use run-time arbitration such as that proposed by Shi and Burns. In Theorems 2 and 3, we present SLA under the condition that task deadlines are less than or equal to their corresponding periods. This prevents jobs of the same task from interfering with each other. In Theorem 4, we analytically and empirically prove the tightness of SLA compared to FLA. In Theorem 5, we extend the analysis to relax the deadline assumption and obtain a more generalized WCL analysis.

5.1 Resource Model

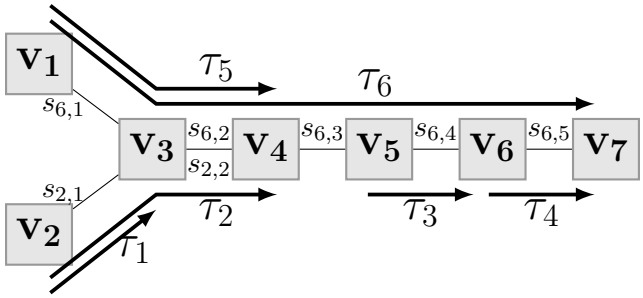
The resource model consists of pipelined communication resources that connect the computation resources. The computation tasks execute on the computation resources and communicate using communication tasks that execute on the communication resources. Our work focuses on analyzing the worst-case latencies for the communication tasks. The communication tasks that execute over the communication resources support fixed priority preemption. A communication task can execute over multiple communication resources or stages. Figure 5.1a shows a deployment of communication tasks to communication resources. Each node represents a computation resource and each edge represents a communication resource. Figures 5.1b and 5.1c show the resource model for communication tasks τ_6 and τ_2, respectively. The communication task τ_6, used for the communication from the

computation task running on node V_1 to computation task running on node V_7, executes on five stages. The communication stages are pipelined and have the following characteristics:

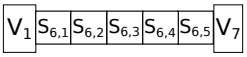
1. A path for the communication between two computation tasks is a sequence of one or more stages. A communication task accesses the stages of its path in an orderly fashion.

2. Data is transmitted on stages in parallel, i.e., when data moves from an earlier to a later stage, new data can be transmitted on the earlier stage in parallel. In Figure 5.1b, after a data unit is transmitted on stage $s_{6,1}$, it will move to stage $s_{6,2}$ and a new data unit can be transmitted on stage $s_{6,1}$.

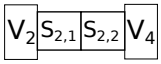
3. If a data unit of a communication task τ_i is transmitted at time t on one stage $s_{i,l-1}$, then it will be ready for transmission at time $t+R_{s_{i,l}}$ on the next stage $s_{i,l}$ (where $R_{s_{i,l}}$ is a delay associated with accessing a new stage). This data unit will be transmitted on the next stage at time $t + R_{s_{i,l}}$ unless it is preempted by a higher priority data unit.



(a) Deployment of communication tasks to resources



(b) Model for task τ_6



(c) Model for task τ_2

Figure 5.1: Motivating example

If multiple data units are ready for transmission on a stage, then the higher priority one will be transmitted on the stage. Figure 5.1a shows that tasks τ_6 and τ_2 share the same second stage $s_{6,2} = s_{2,2}$. If a data unit from τ_6 and another from τ_2 attempt to transmit on $s_{6,2}$ (or $s_{2,2,}$) at time t, the higher priority data unit will be transmitted on $s_{6,2}$. The lower priority data unit will only transmit when there are no higher priority data units pending transmission.

Since communication tasks support fixed priority preemption, then a higher priority task can preempt the execution of a lower priority task on any stage. Hence, in the proposed model, buffers are required at the computation resources and between the communication resources. We assume in this chapter that there is enough buffer space to store data. However, we can use the stage-level analysis to also derive an upper-bound on the buffer space requirements as we show in Chapter 7. Note that since tasks support fixed priority preemption, lower priority tasks might starve for communication resources. The analysis presented in this chapter finds the WCL for each communication task; hence, it can detect if starvation may occur.

5.2 Communication Task Model

We present the definitions, terminology and the model necessary for describing the stage-level analysis. In our analysis, we assume the assignment of distinct priorities to communication tasks. We assume that a set of communication tasks Γ is deployed on the communication resources. We also assume that communication tasks have fixed paths that are determined offline.

Definition 6 (Communication task set)**.** *The set of communication tasks* $\Gamma := \{\tau_i : \forall i \in [1, n]\}$ *has* n *communication tasks, where a communication task* τ_i *is a 5-tuple* $\langle P_i, T_i, D_i, J_i^R, L_i \rangle$*. This describes a communication task* τ_i *with priority* P_i*, period* T_i *between successive job transmissions, real-time deadline* D_i*, release jitter* J_i^R*, and the basic stage latency* L_i*.*

A communication task transmits data from a source computation resource to a destination. A communication task is schedulable if its WCL R_i is less than or equal to the deadline D_i. The release jitter J_i^R is the worst-case delay in a job's release time. Communication tasks can be periodic tasks with a period T_i or sporadic tasks with a minimum interarrival time T_i between jobs. The basic stage latency, L_i, is the WCL of a job of the task on one stage when it does not suffer interferences from any other tasks on that stage.

The path that the communication task τ_i traverses is a sequence of stages denoting multiple communication resources that it crosses to reach from the source computation resource to the destination.

Definition 7 (Path)**.** *A path* δ_i *for communication task* τ_i *is a sequence of stages* $(s_{i,1}, \ldots, s_{i,|\delta_i|})$*.*

Definition 8 (Subpath)**.** *A subpath* $\sigma_i(s_{i,l})$ *for communication task* τ_i *is a sequence of stages*
$(s_{i,1}, \ldots, s_{i,l})$ *such that* $\sigma_i(s_{i,l})$ *has the same first stage* s_1 *as path* δ_i*, and* $l < |\delta_i|$ *with* $s_{i,l}$ *being the last stage of the subpath.*

We use $|\delta_i|$ to denote the number of stages in path δ_i. The basic latency of a communication task τ_i along its path δ_i is its execution time on all stages along its path without suffering any interference. The term $R_{s_{i,l}}$ is the delay experienced by a data unit as it moves from stage $s_{i,l-1}$ to stage $s_{i,l}$. Since the stages are pipelined and tasks experience a delay $R_{s_{i,l}}$ when moving from one stage to another, we can compute the basic latency of a communication task as $C_i = L_i + \sum_{l=2\ldots|\delta_i|} R_{s_{i,l}}$. Note that a higher priority data unit, moving from one stage to another, can be blocked by a lower priority data unit that has already started transmission. This blocking time (at each stage) is at most the latency of transmitting one data unit. This time is constant at each stage. For clarity, we drop the blocking time from our analysis and derivations given that it is a constant that can be added to the WCLs presented in this work.

We use the notation $s \in \delta_i$ to denote that a stage s exists on the path δ_i. We also use $\delta_i \cap \delta_j$ to denote the set of stages that exist on both paths δ_i and δ_j. For $s, s' \in \delta_i$, we use tick (s') to denote a stage that precedes another stage s in the path δ_i.

We present an illustrative example in Figure 5.1a to familiarize the reader with the terminology. Figure 5.1a maps six communication tasks to communication resources: $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$ with the priorities $P_1 > P_2 > P_3 > P_4 > P_5 > P_6$. Edges between nodes represent communication resources. Note that the choice of our paths in Figure 5.1a are selected solely to illustrate and explain the stage-level analysis. Table 5.1 presents the basic stage latency, the period (or minimum interarrival time), the deadline, and the jitter for each of the tasks in Figure 5.1a. For simplicity, the tasks have zero jitter and the task deadlines are equal to their periods. Henceforth, we use the term "task" to refer to a communication task.

Table 5.1: Data for communication tasks in Figure 5.1a

Task	L	T	D	J
τ_1	2	8	8	0
τ_2	2	8	8	0
τ_3	2	8	8	0
τ_4	2	8	8	0
τ_5	2	8	8	0
τ_6	9	50	50	0

We use Figure 5.2 to show the transmission of one job of τ_6 on each of the stages in the path δ_6 for the example configuration in Figure 5.1a. We assume a per stage delay $R_{s_{i,l}} = 1$. For example, since τ_5 has a higher priority than τ_6, jobs of τ_5 will preempt the job of τ_6. White spaces on stage (v_4, v_5) represent gaps caused by interfering tasks on the predecessor stage (v_3, v_4). These tasks no longer interfere with τ_6 on stage (v_4, v_5), but the data units of τ_6 remain separated by the gaps shown due to interferences on previous stages. Note that in our derivation of an upper bound for the latency, we assume that these gaps are part of the latency of task τ_6 as we show in more detail in the following sections.

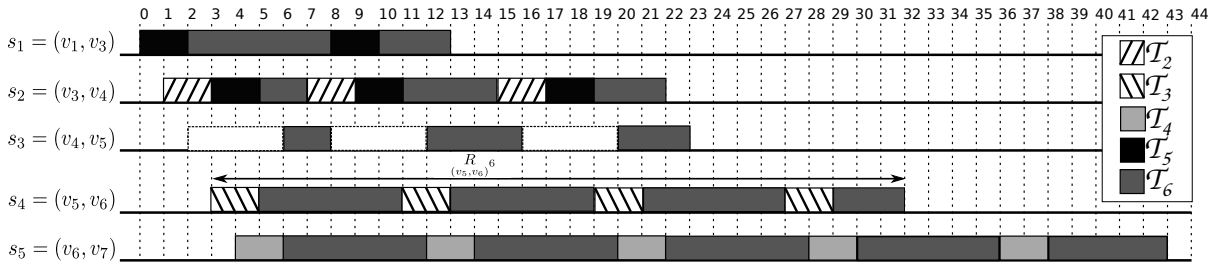


Figure 5.2: Timeline of task τ_6 in Figure 5.1a

5.3 Direct Interference

Direct interference occurs when a higher priority task τ_j preempts a lower priority task τ_i on a shared stage. We formalize direct interference at the stage-level in Definition 9, and

the set of tasks resulting in direct interferences on a stage in Definition 10.

Definition 9 (Direct interference)**.** *A task τ_i suffers direct interference from task τ_j on a stage s if and only if $s \in \delta_i \cap \delta_j$, and $P_i < P_j$.*

Definition 10 (Direct interference set)**.** *The set of tasks directly interfering with task τ_i on a stage s is $\mathbb{S}_{s_i}^{D} = \{\tau_j \mid \forall \tau_j \in \Gamma, s \in \delta_i \cap \delta_j \text{ and } P_i < P_j\}$.*

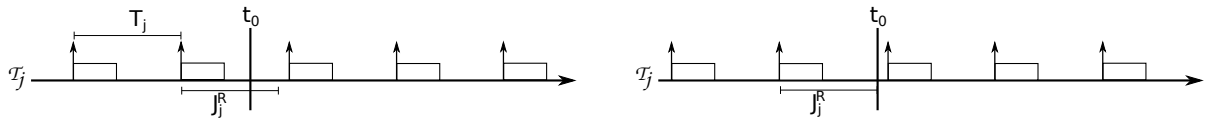
From Figure 5.1a, we observe that τ_6 has direct interference on its second stage $s_{6,2} = (v_3, v_4)$ from tasks τ_2 and τ_5 such that $\mathbb{S}_{(v_3,v_4)}^{D_6} = \{\tau_2, \tau_5\}$.

5.4 Worst-Case Latency with Direct Interference

We derive the WCL for the special case of $D_i \leq (T_i - J_i^R)$. This case prevents interference of a job of a task under analysis with other jobs of the same task. This is commonly referred to as self-blocking. Later, we relax this restriction.

We define the worst-case contribution of a higher priority task to the latency of a task under analysis in Lemma 1. We then derive the WCL for the first stage on the path of the task under analysis in Lemma 2, followed by deriving the WCL for any arbitrary stage on the path in Lemma 4. Finally, we derive the WCL for the task under analysis along its path in Theorem 2.

Figure 5.3a shows a sample activation pattern for the jobs of a higher priority task τ_j directly interfering with a task under analysis τ_i. The time instant t_0 represents the release of the job under analysis of task τ_i.



(a) Example pattern (b) Modified pattern

Figure 5.3: Activation pattern of task τ_j

Lemma 1. *The worst-case contribution from a higher priority task τ_j to the latency of a job under analysis of task τ_i on a stage $s \in \delta_i$ is achieved when the following rules are applied:*

1. *Jobs of τ_j that are activated before t_0 and have enough jitter to be released at t_0 are released at t_0.*
2. *Jobs of τ_j that are activated after t_0 are released immediately with no jitter.*
3. *The release of one of the jobs of τ_j coincides with t_0 after experiencing maximum jitter.*

Proof. These three rules lead to a larger WCL for the job under analysis. The first rule is concerned with jobs of τ_j that are activated before t_0, but can be released at or after t_0. If these jobs are released before t_0, they will either not interfere with the job under analysis or will cause less interference compared to being released at t_0. Also if these jobs are released after t_0, they might not contribute to the WCL of the job under analysis. Hence, releasing the jobs at t_0 leads to maximum interference.

The second rule states that jobs activated after t_0 should be immediately released. Since these jobs are activated after t_0, then they might interfere with the job under analysis. The earlier the jobs are released, the higher the chance they will cause interference. Hence, the immediate release of these jobs contributes most to the WCL of the job under analysis.

The third rule states that the contribution of τ_j to the latency of the job under analysis is worst when the release of one of its jobs coincides with t_0 after suffering maximum jitter. Consider, for instance, the activation pattern in Figure 5.3a. Assume (without loss of generality), that according to the first two rules, only one job of τ_j has enough jitter to be released at t_0 (first job to the left of t_0) and two jobs activated after t_0 contribute to the latency of the job under analysis (first two jobs to the right of t_0). Moving the activations of the jobs of τ_j earlier in time (as shown in Figure 5.3b) until the first job (released at t_0) has maximum jitter while still being released at t_0, contributes most to the latency of the job under analysis. The reason is that by moving activations earlier in time, jobs of τ_j that were activated after t_0 (the last job to the right) and could not interfere with the job under analysis might be able to cause interference. □

Lemma 2. *The WCL $\underset{s}{R_i}$ of a task τ_i suffering direct interferences from other tasks in $\underset{s}{\mathbb{S}_i^D}$ on the first stage $s = s_{i,1} \in \delta_i$ under the condition $D_i \leq (T_i - J_i^R)$, is given by the following:*

$$\underset{s}{R_i} = \underset{s}{I_i} + L_i$$

where

$$\underset{s}{I_i} = \sum_{\forall \tau_j \in \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{R_i} + J_j^R}{T_j} \right\rceil * L_j$$

Proof. Given $D_i \leq (T_i - J_i^R)$, then a job under analysis of task τ_i does not suffer interference from other jobs of the same task. Assume $\underset{s}{R_i}$ is the upper bound on the WCL of a job under analysis of task τ_i. Consider a higher priority task $\tau_j \in \underset{s}{\mathbb{S}_i^D}$ that directly interferes with task τ_i. From Lemma 1, the interval of time during which τ_j can interfere with the job under analysis is the WCL of the job plus the maximum release jitter of τ_j, i.e., $\underset{s}{R_i} + J_j^R$. The maximum number of jobs of task τ_j that interfere with τ_i in that interval of time is thus equal to $\left\lceil \frac{\underset{s}{R_i} + J_j^R}{T_j} \right\rceil$. The latency contribution of τ_j to the WCL of τ_i is equal to the maximum number of interfering jobs multiplied by the basic stage latency of τ_j, i.e., $\left\lceil \frac{\underset{s}{R_i} + J_j^R}{T_j} \right\rceil * L_j$. Therefore, the WCL of τ_i on stage $s_{i,1}$ is the summation of the interference

from all higher priority tasks in the set $\underset{s}{\mathbb{S}}_i^D$ plus the basic stage latency of τ_i, i.e., $\underset{s}{I}_i + L_i$. This results in a recurrence relation that can be solved iteratively to find $\underset{s}{R}_i$. □

Under the assumption that $D_i \leq (T_i - J_i^R)$ and using only direct interference, so far we have derived the WCL of a task under analysis τ_i on the first stage of its path δ_i. Next, we derive the WCL of τ_i on any arbitrary stage s along its path δ_i. The terms $\underset{s'}{R}_i$ and $\underset{s'}{I}_i$ represent the WCL and the worst-case interference of task τ_i on a stage s' preceding a stage s on the path δ_i, respectively.

Lemma 3. *The WCL $\underset{s}{R}_i$ of task τ_i on stage s is monotonically non-decreasing with respect to the sequence of stages of the task's path δ_i.*

Proof. $\underset{s'}{R}_i$ is the worst-case time interval between the first and last data units of the job of τ_i on stage s'. If no new interferences exist on stage s (compared to those on stage s'), then the WCL $\underset{s}{R}_i$ is equal to $\underset{s'}{R}_i$. The reason is that any data unit that was transmitted at a time t on stage s' will attempt transmission at time $t + R_{s_{i,l}}$ on stage s with the same ordering of data units as on s'. Assume a higher priority data unit was transmitted at time t on stage s' followed by a data unit of τ_i at time $t + 1$. If the higher priority data unit does not exist on stage s, then the slot at time $t + R_{s_{i,l}}$ will be empty. This will not cause the τ_i data unit to transmit any earlier on stage s as it will attempt transmission at time $t + 1 + R_{s_{i,l}}$ leaving a gap at the time $t + R_{s_{i,l}}$. Therefore, the WCL on stage s is greater than or equal that on stage s', i.e., $\underset{s}{R}_i \geq \underset{s'}{R}_i$. □

Lemma 4. *The WCL $\underset{s}{R}_i$ of a task τ_i suffering direct interferences from other tasks in $\underset{s}{\mathbb{S}}_i^D$ on stage $s \in \delta_i$, is given by:*

$$\underset{s}{R}_i = \underset{s}{I}_i + L_i$$

where

$$\underset{s}{I}_i = \underset{s'}{I}_i + \sum_{\forall \tau_j \in \underset{s}{\mathbb{S}}_i^D} \left\lceil \frac{\underset{s}{R}_i + J_j^R}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \underset{s}{\mathbb{S}}_i^D \cap \underset{s'}{\mathbb{S}}_i^D} \left\lceil \frac{\underset{s'}{R}_i + J_j^R}{T_j} \right\rceil * L_j$$

Proof. **Base Case:** Note that in the interference term $\underset{s}{I}_i$, and for the first stage $s = s_{i,1}$, the stage s' does not exist. Hence, for the first stage, $\underset{s'}{I}_i = 0$. Also the set $\underset{s}{\mathbb{S}}_i^D \cap \underset{s'}{\mathbb{S}}_i^D = \emptyset$, which makes the last term $\sum_{\forall \tau_j \in \underset{s}{\mathbb{S}}_i^D \cap \underset{s'}{\mathbb{S}}_i^D} \left\lceil \frac{\underset{s'}{R}_i + J_j^R}{T_j} \right\rceil * L_j = 0$. Therefore, for the first stage, the interference term becomes the one derived from Lemma 2. This provides the base case for our proof. We prove Lemma 4 by induction. Assume Lemma 4 holds for an arbitrary stage $s' \in \delta_i$. We now prove Lemma 4 for the succeeding stage $s \in \delta_i$.

Consider three higher priority tasks: τ_j, τ_k, and τ_l that interfere with task τ_i. Assume that τ_j interferes with τ_i only on stage s', τ_l interferes on stage s only, and that τ_k causes interference on both stages s' and s. For simplicity (and without loss of generality), assume that these are the only interfering tasks with τ_i on the stages s' and s such that $\underset{s'}{\mathbb{S}}_i^D = \{\tau_j, \tau_k\}$ and $\underset{s}{\mathbb{S}}_i^D = \{\tau_k, \tau_l\}$.

In Lemma 3, we illustrated that any higher priority jobs that caused interference to τ_i on s' cannot cause any more interference on stage s. If any new interference occurs on stage s, then it will delay the data units arriving from s' by an interval of time equivalent, in the worst-case, to the interference caused by the new jobs. Hence, a conservative approach to computing the latency of τ_i on stage s would be by adding the interferences from τ_k and τ_l on stage s to the latency computed on stage s':

$$\underset{s}{R}_i = \underset{s'}{R}_i + \left\lceil \frac{\underset{s}{R}_i + J_k^R}{T_k} \right\rceil * L_k + \left\lceil \frac{\underset{s}{R}_i + J_l^R}{T_l} \right\rceil * L_l \tag{5.1}$$

We, however, can further tighten the upper bound on the latency of τ_i on stage s. Note that in the Equation 5.1 (conservative approach), we computed $\underset{s}{R}_i$ to include interference from jobs of τ_k, although, some of these jobs have already been accounted for in $\underset{s'}{R}_i$. Since higher priority jobs that caused interference to τ_i on s' cannot cause any more interference on stage s, then we can achieve a tighter latency bound by ensuring that, in the computation of $\underset{s}{R}_i$, we only add interferences that were not accounted for in $\underset{s'}{R}_i$. The WCL on stage s', $\underset{s'}{R}_i$, includes interferences in the set $\underset{s'}{\mathbb{S}}_i^D = \{\tau_j, \tau_k\}$. The WCL on stage s, $\underset{s}{R}_i$, should include interferences in the set $\underset{s}{\mathbb{S}}_i^D = \{\tau_k, \tau_l\}$. Note, however, that τ_k is a common task, i.e., $\underset{s'}{\mathbb{S}}_i^D \cap \underset{s}{\mathbb{S}}_i^D = \{\tau_k\}$ so we want to only add interference from τ_l and any new jobs from τ_k that did not exist on s'. Also note that, from Lemma 3, the WCL $\underset{s}{R}_i$ is monotonically non-decreasing with respect to the stages on path δ_i. Hence, the number of jobs of τ_k causing interference on stage s can only be greater than or equal to those on stage s'. The number of jobs of τ_k that caused interference on stage s' is $\left\lceil \frac{\underset{s'}{R}_i + J_k^R}{T_k} \right\rceil$. And the number of τ_k jobs that causes interference on stage s is $\left\lceil \frac{\underset{s}{R}_i + J_k^R}{T_k} \right\rceil$. Therefore, a tighter bound can be achieved by subtracting the latency of the jobs of τ_k that already existed on stage s' and will definitely exist in the jobs interfering on stage s:

$$\underset{s}{R}_i = \underset{s'}{R}_i + \left\lceil \frac{\underset{s}{R}_i + J_k^R}{T_k} \right\rceil * L_k + \left\lceil \frac{\underset{s}{R}_i + J_l^R}{T_l} \right\rceil * L_l - \left\lceil \frac{\underset{s'}{R}_i + J_k^R}{T_k} \right\rceil * L_k$$

Since $\underset{s}{R}_i = \underset{s}{I}_i + L_i$ and $\underset{s'}{R}_i = \underset{s'}{I}_i + L_i$, then we can replace $\underset{s}{R}_i$ and $\underset{s'}{R}_i$ by $\underset{s}{I}_i$ and $\underset{s'}{I}_i$, respectively in the equation. And by generalizing the equation, we add interferences in the set $\underset{s}{\mathbb{S}}_i^D$ (corresponding to τ_k and τ_l) and subtract interferences in the set of common tasks

$\mathbb{S}^D_{s'_i} \cap \mathbb{S}^D_{s_i}$ (corresponding to τ_k). We, hence, get:

$$\underset{s}{I_i} = \underset{s'}{I_i} + \sum_{\forall \tau_j \in \mathbb{S}^D_{s_i}} \left\lceil \frac{\underset{s}{R_i} + J^R_j}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}^D_{s_i} \cap \mathbb{S}^D_{s'_i}} \left\lceil \frac{\underset{s'}{R_i} + J^R_j}{T_j} \right\rceil * L_j$$

□

Next, we show a theorem for computing the WCL of a task on its path δ_i under the assumption that $D_i \leq (T_i - J^R_i)$ and using only direct interferences. Note that if a task τ_j directly interferes with the task under analysis τ_i on a non-consecutive set of stages, then τ_j must be split into two or more directly interfering tasks such that each newly created task interferes with τ_i on consecutive stages.

Theorem 2. *The WCL R_i of a task τ_i suffering only direct interferences along its path δ_i where the last stage on the path is $s = s_{i,|\delta_i|}$ and under the condition $D_i \leq (T_i - J^R_i)$, is given by:*

$$R_i = \underset{s}{R_i} + J^R_i + \sum_{l=2...|\delta_i|} R_{s_{i,l}}$$

where

$$\underset{s}{R_i} = \underset{s}{I_i} + L_i$$

$$\underset{s}{I_i} = \underset{s'}{I_i} + \sum_{\forall \tau_j \in \mathbb{S}^D_{s_i}} \left\lceil \frac{\underset{s}{R_i} + J^R_j}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}^D_{s_i} \cap \mathbb{S}^D_{s'_i}} \left\lceil \frac{\underset{s'}{R_i} + J^R_j}{T_j} \right\rceil * L_j$$

Proof. From Lemma 3, the WCL of a task under analysis τ_i is monotonically non-decreasing with respect to the stages. Hence, the WCL $\underset{s}{R_i}$ is largest at the last stage of the path $s = s_{i,|\delta_i|} \in \delta_i$. The WCL $\underset{s}{R_i}$ is measured from the time t_0 (the release of the job under analysis of task τ_i). Since, the job could have been released after suffering maximum jitter, then the WCL measured from the activation time is equal to $\underset{s}{R_i} + J^R_i$. And since we want to find the WCL across the whole path δ_i, then we must take into account the stage delay $R_{s_{i,l}}$. Since this delay is experienced between stages, then the total delay is equal to $\sum_{l=2...|\delta_i|} R_{s_{i,l}}$. Therefore, the WCL of task τ_i, $R_i = \underset{s}{R_i} + J^R_i + \sum_{l=2...|\delta_i|} R_{s_{i,l}}$ where $s = s_{i,|\delta_i|} \in \delta_i$ is the last stage on the path δ_i. The recurrence relation derived in Lemma 4 is used to find $\underset{s}{R_i}$. □

5.5 Indirect Interference

Task τ_i suffers indirect interference on a stage from task τ_k when task τ_i has direct interference with an intermediate task τ_j, and τ_j has direct interference with task τ_k; however,

τ_i has no direct interference with τ_k. In addition, the interference between τ_j and τ_k must occur before τ_j interferes with τ_i. We formally describe this in Definition 11, and the set of indirectly interfering tasks in Definition 12. Revisiting Figure 5.1a, we point out that task τ_6 has indirect interference with task τ_1 through an intermediate task τ_2 on stage (v_3, v_4) such that $\underset{(v_3,v_4)}{\mathbb{S}}{}^{I}_{6} = \{\tau_1\}$.

Definition 11 (Indirect interference)**.** *Given two stages s and $\hat{s}$ on path δ_j, task τ_i suffers indirect interference from task τ_k on stage s if and only if $(s \in \delta_i \cap \delta_j) \wedge (\hat{s} \in \delta_j \cap \delta_k) \wedge (s \neq \hat{s}) \wedge (\hat{s} \notin \delta_i) \wedge (\hat{s} \in \sigma_j(s))$, and $P_i < P_j < P_k$.*

Definition 12 (Indirect interference set)**.** *Given two stages s and $\hat{s}$ on path δ_j, the set of tasks indirectly interfering with task τ_i on stage s is $\underset{s}{\mathbb{S}}{}^{I}_{i} = \{\tau_k \mid \forall \tau_j, \tau_k \in \Gamma, (s \in \delta_i \cap \delta_j) \wedge (\hat{s} \in \delta_j \cap \delta_k) \wedge (s \neq \hat{s}) \wedge (\hat{s} \notin \delta_i) \wedge (\hat{s} \in \sigma_j(s))$, and $P_i < P_j < P_k\}$.*

Indirect interferences are accounted for by *indirect interference jitter*. Lemma 1 introduced the conditions under which a higher priority task τ_j causes maximum interference to τ_i. It is clear that the interference caused by τ_k can delay the release of the jobs of τ_j. Assume that the interference that τ_j suffers from task τ_k is equal to I_j. From the point of view of τ_i, the maximum jitter that τ_j can suffer is equal to the sum of the interference that τ_j suffers from τ_k and its release jitter; $I_j + J^R_j$. In the worst case (with respect to τ_i), this indirect interference can further delay the jobs of τ_j activated before t_0 while the jobs activated after t_0 are immediately released. This extra jitter caused by the indirect interference is known as *indirect interference jitter*. Consider the timeline in Figure 5.2. Task τ_6 suffers direct interference on stage $s_{6,2}$ from task τ_2 and suffers indirect interference from task τ_1 through the intermediate task τ_2. Task τ_2 suffers an interference of 2 time units from task τ_1. Hence, the first release of task τ_2 on stage $s_{2,2}$ at $t_0 = 1$ has a maximum indirect interference jitter of 2 time units. The subsequent jobs of task τ_2 are released immediately at their activation time.

It is important to mention that the indirect interference jitter depends on the indirect interference set of the task under analysis. Using the same aforementioned tasks as an example, this means that I_j only contains interference from common tasks in the direct interference set of τ_j and the indirect interference set of τ_i. This is because other tasks interfering with τ_i and in the direct interference set of τ_j, $\underset{s}{\mathbb{S}}{}^{D}_{j}$, but not in the indirect interference set of τ_i, $\underset{s}{\mathbb{S}}{}^{I}_{i}$, are tasks that directly interfere with τ_i, i.e., in the set $\underset{s}{\mathbb{S}}{}^{D}_{i}$. These tasks are already accounted for by direct interference on τ_i and should not be accounted for by indirect interference jitter through τ_j. Hence, the indirect interference that τ_i suffers through τ_j is from tasks in the set $\underset{s}{\mathbb{S}}{}^{D}_{j} \cap \underset{s}{\mathbb{S}}{}^{I}_{i}$. We use the terms $\underset{s}{R}_j(\tau_i)$ and $\underset{s}{I}_j(\tau_i)$ to denote the WCL and worst-case interference of task τ_j, respectively, but only due to tasks in the set $\underset{s}{\mathbb{S}}{}^{D}_{j} \cap \underset{s}{\mathbb{S}}{}^{I}_{i}$. Therefore, the indirect interference jitter of a task τ_j with respect to a task under analysis τ_i is given by:

$$\underset{s}{J}{}^{I}_{j} = \underset{s}{R}_j(\tau_i) - L_j \tag{5.2}$$

where

$$\underset{s}{R_j}(\tau_i) = \underset{s}{I_j}(\tau_i) + L_j$$

$$\underset{s}{I_j}(\tau_i) = \underset{s'}{I_j}(\tau_i) + \sum_{\forall \tau_k \in \mathbb{S}_j^D \cap \mathbb{S}_i^I} \left\lceil \frac{\underset{s}{R_j}(\tau_i) + J_k^R + \underset{s}{J_k^I}}{T_k} \right\rceil * L_k - \sum_{\forall \tau_k \in \mathbb{S}_j^D \cap \mathbb{S}_{j}^{D} \cap \mathbb{S}_i^I} \left\lceil \frac{\underset{s'}{R_j}(\tau_i) + J_k^R + \underset{s}{J_k^I}}{T_k} \right\rceil * L_k$$

5.6 Worst-Case Latency with Indirect Interference

We can now extend Theorem 2 to account for both direct and indirect interferences. The WCL of task τ_i when it experiences both direct and indirect interferences is given by Theorem 3.

Theorem 3. *The WCL R_i of a task τ_i suffering both direct and indirect interferences along its path δ_i where the last stage on the path is $s = s_{i,|\delta_i|}$ and under the condition $D_i \leq (T_i - J_i^R)$, is given by:*

$$R_i = \underset{s}{R_i} + J_i^R + \sum_{l=2...|\delta_i|} R_{s_{i,l}}$$

where

$$\underset{s}{R_i} = \underset{s}{I_i} + L_i$$

$$\underset{s}{I_i} = \underset{s'}{I_i} + \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i}^{D}} \left\lceil \frac{\underset{s'}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j$$

Proof. Indirect interference jitter delays the release of higher priority jobs directly interfering with the task under analysis τ_i. From Lemma 1, the interval of time during which τ_j can interfere with τ_i is the WCL of the job plus the maximum jitter of τ_j. The maximum jitter of τ_j is equal to the sum of the release and indirect interference jitters of τ_j, $J_j^R + \underset{s}{J_j^I}$. Hence, the interval of time during which τ_j can interfere with τ_i is equal to $\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}$. Substituting this term for the numerators of the summations in Theorem 2 provides the WCL for task τ_i with direct and indirect interferences. □

5.7 An Illustrative Example

We use Figure 5.1a as an illustrative example to show how SLA is applied and compare it to FLA [126, 128]. Recall that the tasks in Figure 5.1a have the data in Table 5.1, and

the following priorities: $P_1 > P_2 > P_3 > P_4 > P_5 > P_6$. Table 5.2 shows the WCLs for all six tasks using both FLA and SLA. We observe that the results from SLA are less than or equal to the upper bounds computed by FLA. Task τ_6 is unschedulable using FLA.

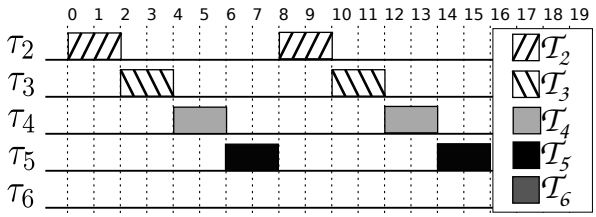


Figure 5.4: Timeline of task τ_6 in Figure 5.1a using FLA

According to FLA, τ_6 has direct interference with the tasks τ_2, τ_3, τ_4, and τ_5. Figure 5.4 shows the timeline for task τ_6 using FLA. The four tasks that interfere with τ_6 have a basic stage latency of two time units and a period of 8 time units. This means that all four tasks consume all bandwidth, i.e,. the utilization of the communication resources reaches a 100%, and the data units of τ_6 can never be sent. Thus, τ_6 is unschedulable for any deadline. This occurs because FLA assumes that τ_6 suffers simultaneous interference from τ_2, τ_3, τ_4, and τ_5 on all stages along its path. This is certainly not the case as shown in Figure 5.2. In fact, the WCL of τ_6 can be computed, but it requires performing the analysis at the stage-level. This provides a simple example where SLA performs better than FLA.

Table 5.2: Results for the example in Figure 5.1a using both SLA and FLA

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
FLA	3	5	3	3	6	-
SLA	3	5	3	3	5	43

5.8 Tightness Analysis

We expect SLA to have tighter latency bounds compared to FLA. The reason is that FLA assumes that the interference that a task suffers on any stage occurs on the whole path of the task while SLA restricts the interference only to the stages on which they happen. In what follows, we formally prove that SLA provides tighter bounds than FLA.

Theorem 4. *Given a set of tasks* Γ *and their paths,* $R_i^{SLA} \leq R_i^{FLA}$, $\forall \tau_i \in \Gamma$.

Proof. According to FLA [126], the WCL of τ_i measured from the release of τ_i is given by:

$$R_i^{FLA} = \sum_{\forall \tau_j \in S_i^D} \left\lceil \frac{R_i^{FLA} + J_j^R + J_j^I}{T_j} \right\rceil * C_j + C_i \tag{5.3}$$

The worst-case interference occurs when the higher priority tasks share all stages with the path δ_i of τ_i. This means that the set of all interfering tasks on δ_i, $S_i^D = \underset{s}{\mathbb{S}_i^D}$. In that case, using Theorem 3, $\underset{s}{R_i}$ on all stages of δ_i are equal and $R_i^{SLA} = \underset{s}{R_i} + \sum_{l=2...|\delta_i|} R_{s_{i,l}}$ (measured from the release of τ_i). Given that $C_i = L_i + \sum_{l=2...|\delta_i|} R_{s_{i,l}}$, R_i^{SLA} can be given by:

$$\sum_{\forall \tau_j \in S_i^D} \left\lceil \frac{R_i^{SLA} - \sum_{l=2...|\delta_i|} R_{s_{i,l}} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + C_i \tag{5.4}$$

For FLA, $J_j^I = R_j - C_j$ and for SLA $\underset{s}{J_j^I}$ is given by Equation 5.2. For SLA, the worst-case interference jitter occurs when $\underset{s}{\mathbb{S}_j^D} = \underset{s}{\mathbb{S}_i^I}$. In that case, $\underset{s}{J_j^I} = \underset{s}{R_j} - L_j = R_j - \sum_{l=2...|\delta_j|} R_{s_{i,l}} - L_j = R_j - C_j$. Therefore, $\underset{s}{J_j^I} = J_j^I$. Taking this into account and comparing Equations 5.3 and 5.4, the only difference between FLA and SLA are the terms L_j and C_j, and the stage delay in the summation. Since $C_j = L_j + \sum_{l=2...|\delta_j|} R_{s_{i,l}}$, and $|\delta_j| \geq 1$ then $L_j \leq C_j$. And since the stage delay has a negative sign in the numerator of the summation, therefore, $R_i^{SLA} \leq R_i^{FLA}$ and our analysis gives a tighter bound compared to FLA.

For the case when there is no interference, $R_i^{FLA} = C_i$ and $R_i^{SLA} = L_i + \sum_{l=2...|\delta_i|} R_{s_{i,l}} = R_i^{FLA}$. Since in the presence of interference, $R_i^{SLA} \leq R_i^{FLA}$, and in the absence of interference, $R_i^{SLA} = R_i^{FLA}$, then $R_i^{SLA} \leq R_i^{FLA}$. □

5.9 Relaxing the Deadline Restriction

In this section, we relax the assumption $D_i \leq (T_i - J_i^R)$. This means that jobs of the same task can interfere with one another. Since our model transmits data units of the same priority based on FIFO ordering, then a job released earlier in time will have a higher priority than other jobs of the same task that are released later.

We use the term *level-i busy period*, $\underset{s}{B_i}$, to define an interval of time during which data units of priority P_i or higher are continuously transmitted on stage s before the stage is idle. For example, in Figure 5.2, the time interval $t = 4$ to $t = 43$ on stage $s_{6,5} = (v_6, v_7)$ of task τ_6 represents a level-6 busy period. If we can find an upper bound on a level-i busy period, then we can find the maximum number of jobs of task τ_i that interfere with each other. The WCL of task τ_i is then computed as the maximum WCL of all τ_i jobs in the level-i busy period. First, we find an upper bound for a level-i busy period. Then, we define the WCL of each τ_i job in the level-i busy period.

Lemma 5. *The level-i busy period on stage $s \in \delta_i$ considering both direct and indirect*

interferences to task τ_i is given by:

$$\underset{s}{B_i} = \underset{s'}{B_i} + \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{B_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + \left\lceil \frac{\underset{s}{B_i} + J_i^R}{T_i} \right\rceil * L_i$$
$$- \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i}^{D}} \left\lceil \frac{\underset{s'}{B_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \left\lceil \frac{\underset{s'}{B_i} + J_i^R}{T_i} \right\rceil * L_i$$

Proof. **Base Case:** The rules in Lemma 1 still hold after relaxing the deadline assumption. Considering a level-i busy period, Lemma 1 holds for all tasks of priority i or higher. First, we consider the first stage $s = s_{i,1}$ on the path δ_i of the task under analysis τ_i. On this stage, the length of the busy period is the latency of all τ_i jobs plus interference from any higher priority jobs. Assuming the upper bound on the busy period is $\underset{s}{B_i}$, then according to Lemma 1, the interval of time during which jobs of τ_i can exist in the busy period is equal to the length of the busy period plus the maximum jitter of τ_i, i.e., $\underset{s}{B_i} + J_i^R$. Also, the interval of time during which jobs of a higher priority task τ_j exist in the busy period is equal to the busy period length plus maximum jitter, i.e., $\underset{s}{B_i} + J_j^R + \underset{s}{J_j^I}$. Therefore, considering all higher priority tasks, the length of the busy period on the first stage $s = s_{i,1}$ can be given by:

$$\underset{s}{B_i} = \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{B_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + \left\lceil \frac{\underset{s}{B_i} + J_i^R}{T_i} \right\rceil * L_i \tag{5.5}$$

Equation 5.5 represents the base condition for our proof. We proceed to prove this Lemma by induction, assuming that the given Lemma holds for a stage $s' \in \delta_i$. We showed in Lemma 3 that the latency of task τ_i is monotonically non-decreasing with respect to the stages of the path δ_i. Similarly, the length of the level-i busy period is monotonically non-decreasing with respect to the stages of the path δ_i. We also showed that common higher priority jobs between stages s' and s cannot cause more interference on stage s than the interference they caused on stage s'. Following the same reasoning, jobs of priority i or higher that exist in the busy period of stage s' cannot contribute more to the busy period on stage s than their contribution on stage s'. Hence, we can find the busy period on stage s by adding the latency of all jobs of priority i or higher on stage s to the length of the busy period on s' while subtracting the latency of common jobs.

Since, the length of the busy period is monotonically non-decreasing, then the number of jobs of task τ_i or any higher priority task τ_j on stage s is larger than the number of jobs of the same task on stage s'. The latency of the common jobs between stages s' and s is the latency of the higher priority jobs on stage s' of the tasks in the set $\underset{s}{\mathbb{S}_i^D} \cap \underset{s'}{\mathbb{S}_i^D}$ plus the latency of the jobs of task τ_i that exist on stage s'. This can be represented by:

$$\sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i}^{D}} \left\lceil \frac{\underset{s'}{B_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + \left\lceil \frac{\underset{s'}{B_i} + J_i^R}{T_i} \right\rceil * L_i \tag{5.6}$$

Therefore, we can find the busy period on stage s by summing contribution from jobs of priority i or higher on stage s (as represented by Equation 5.5) to the length of the busy period on stage s', $\underset{s'}{B}_i$, and subtracting the contribution from common jobs (as represented by Equation 5.6). Thus, proving the Lemma. □

So far, we have derived an upper bound on the level-i busy period on a stage s, $\underset{s}{B}_i$. To find the WCL of τ_i on a stage s, we need to compare the WCLs of all jobs in the busy period $\underset{s}{B}_i$. Thus, we need to find the maximum number of jobs of τ_i in the busy period $\underset{s}{B}_i$, $\underset{s}{p}_{B,i} = \left\lceil \frac{\underset{s}{B}_i + J_i^R}{T_i} \right\rceil$. We number the jobs of τ_i in the busy period $\underset{s}{B}_i$ from $p = 1$ to $p = \underset{s}{p}_{B,i}$, with $p = 1$ being the first job released in the busy period. In Theorem 5, we find the WCL of each job in the busy period and use them to find the WCL of τ_i.

Theorem 5. *The WCL R_i of a task τ_i suffering both direct and indirect interferences along its path δ_i where the last stage on the path is $s = s_{i,|\delta_i|}$ is given by:*

$$R_i = \max_{p=1\ldots \underset{s}{p}_{B,i}} (\underset{s}{w}_i(p) - (p-1) * T_i + J_i^R) + \sum_{l=2\ldots|\delta_i|} R_{s_{i,l}}$$

where

$$\underset{s}{w}_i(p) = \underset{s}{I}_i(p) + p * L_i$$

$$\underset{s}{I}_i(p) = \underset{s'}{I}_i(p') + \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w}_i(p) + J_j^R + \underset{s}{J}_j^I}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i'}^D} \left\lceil \frac{\underset{s'}{w}_i(p') + J_j^R + \underset{s}{J}_j^I}{T_j} \right\rceil * L_j$$

and

$$p' = \begin{cases} p & \text{if } p \leq \underset{s'}{p}_{B,i} \\ \underset{s'}{p}_{B,i} & \text{otherwise} \end{cases} \tag{5.7}$$

Proof. We first derive $\underset{s}{w}_i(p)$, the worst-case completion time of job p on stage s measured from the start of the busy period $\underset{s}{B}_i$, on the first stage $s = s_{i,1}$. Then proceed to prove the Theorem by induction.

Base Case: The time interval $\underset{s}{w}_i(p)$ represents the time at which the p^{th} job of τ_i completes transmission. It includes higher priority jobs of interfering tasks and higher priority jobs of τ_i (jobs that were released earlier than job p). The latency of τ_i jobs in $\underset{s}{w}_i(p)$ including the p^{th} job is equal to $p * L_i$. The interval during which a higher priority task τ_j can interfere with τ_i is equal to the time interval $\underset{s}{w}_i(p)$ plus maximum jitter, i.e., $\underset{s}{w}_i(p) + J_j^R + \underset{s}{J}_j^I$. Hence, considering all higher priority tasks, we can represent the time

interval $\underset{s}{w_i}(p)$ on the first stage $s = s_1$ by:

$$\underset{s}{w_i}(p) = \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + p * L_i \tag{5.8}$$

Notice that the definition of $\underset{s}{w_i}(p)$ reduces to Equation 5.8 on the first stage of the path δ_i. Equation 5.8 serves as the base condition for our proof. We now assume the Theorem holds for a stage $s' \in \delta_i$ and derive it for stage s.

Again, we use two facts that we proved in Lemma 3. The first is that $\underset{s}{w_i}(p)$ is monotonically non-decreasing. And the second is that common higher priority jobs between stages s' and s cannot cause more interference on stage s. Assuming that job p exists on both stages s and s', then we can obtain $\underset{s}{w_i}(p)$ by adding to $\underset{s'}{w_i}(p)$ the latency of jobs that exist on stage s and subtracting the latency of common jobs between stages s and s'. Note that the common jobs include jobs from τ_i as well. Hence, we can represent $\underset{s}{w_i}(p)$ by:

$$\begin{aligned}\underset{s}{w_i}(p) = \underset{s'}{w_i}(p) &+ \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + p * L_i \\ &- \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i}^{D}} \left\lceil \frac{\underset{s'}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - p * L_i \end{aligned} \tag{5.9}$$

Equation 5.9 is valid under the assumption that job p of task τ_i exists on both stages s' and s. Since $\underset{s}{w_i}(p)$ is monotonically non-decreasing, then the number of τ_i jobs on stage s are greater than or equal to that on stage s'. Thus, job p might only exist on stage s but not s'. So we introduce the term p'. If job p exists on s', then $p' = p$ and Equation 5.9 holds. However, if job p does not exist on stage s', then $\underset{s'}{w_i}(p)$ is undefined. In such case, we find common jobs in the interval $\underset{s'}{w_i}(\underset{s'}{p_{B,i}})$ which is the worst-case completion time of the last job $\underset{s'}{p_{B,i}}$ on stage s' measured from the start of the busy period $\underset{s'}{B_i}$. This is effectively the whole length of the busy period on stage s'. Therefore, we modify Equation 5.9 to use p' to refer to the job on the preceding stage s'. The equation thus becomes:

$$\begin{aligned}\underset{s}{w_i}(p) = \underset{s'}{w_i}(p') &+ \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j + p * L_i \\ &- \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_{i}^{D}} \left\lceil \frac{\underset{s'}{w_i}(p') + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - p' * L_i \end{aligned} \tag{5.10}$$

Since $\underset{s}{w_i}(p) = \underset{s}{I_i}(p) + p * L_i$ and $\underset{s'}{w_i}(p') = \underset{s'}{I_i}(p') + p' * L_i$, then substituting them in Equation 5.10 yields the definition of $\underset{s}{w_i}(p)$ in the Theorem.

Now, we need to find the WCL of each of the jobs of τ_i in the busy period $\underset{s}{B_i}$. Since $\underset{s}{w_i}(p)$ is the worst-case completion time of job p in the interval $\underset{s}{B_i}$, where $s = s_{i,|\delta_i|}$ is the last stage on path δ_i then subtracting the activation time of job p from $\underset{s}{w_i}(p)$ yields the WCL of the job. The first job $p = 1$ is released at the start of the busy period after suffering maximum release jitter, hence the activation time of the first job is at time $-J_i^R$ (relative to the start of the busy period). The second job is released immediately at its activation time, i.e., at time $T - J_i^R$, and the third job at time $2 * T - J_i^R$. Hence, the p^{th} job is activated at time $(p-1) * T_i - J_i^R$. The WCL of job p is thus $\underset{s}{w_i}(p) - (p-1) * T_i + J_i^R$. Taking the maximum latency across all jobs, $p = 1 \ldots \underset{s}{p_{B,i}}$, yields the WCL of task τ_i on the last stage of its path δ_i. Adding the stage delay to WCL gives us the WCL of τ_i along its path δ_i:

$$R_i = \max_{p=1\ldots \underset{s}{p_{B,i}}} (\underset{s}{w_i}(p) - (p-1) * T_i + J_i^R) + \sum_{l=2\ldots|\delta_i|} R_{s_{i,l}}$$

□

5.10 Experimentation

We quantitatively evaluate the proposed stage-level analysis. The evaluation is performed on a priority-aware NoC with flit-level preemption [126]. Each node in the NoC consists of a processing element and a router. Priority-aware routers have multiple VCs with associated priorities. These priorities are used to preempt the routing of packets at the flit level in lower priority VCs by flits in higher priority VCs. The router selects the output port for a data unit in the VCs based on its desired destination. Computation tasks execute on the processing elements (processing resources) and communicate using messages on the NoC links. Messages transmitted over the network links map to communication tasks executed on communication resources in our model. Data is transmitted in parallel across the links between two communicating processing elements through wormhole switching. Both SLA and FLA can be applied to the priority-aware NoC.

The quantitative evaluation for SLA is performed on a large set of synthetic benchmarks. Using synthetic benchmarks allows varying different factors and measuring their effect on SLA. It also allows us to test extremes of factors, e.g., high and low utilizations, that otherwise are difficult to test for. We perform our experiments on 4×4 and 8×8 instances of the NoC. We compare the analytical bounds of SLA to those of FLA [126]. We randomly generate communication tasks and compute the WCLs using both SLA and FLA.

The goals of our experimentation are as follows:

- Show that SLA schedules more task sets compared to FLA, and quantify the increase in schedulability.
- Quantify the improvement (tightness) of the latency bounds computed by SLA over FLA.

- Quantify the percentage of schedulable task sets that result in an improved latency bound.
- Compare the analysis time of SLA and FLA.

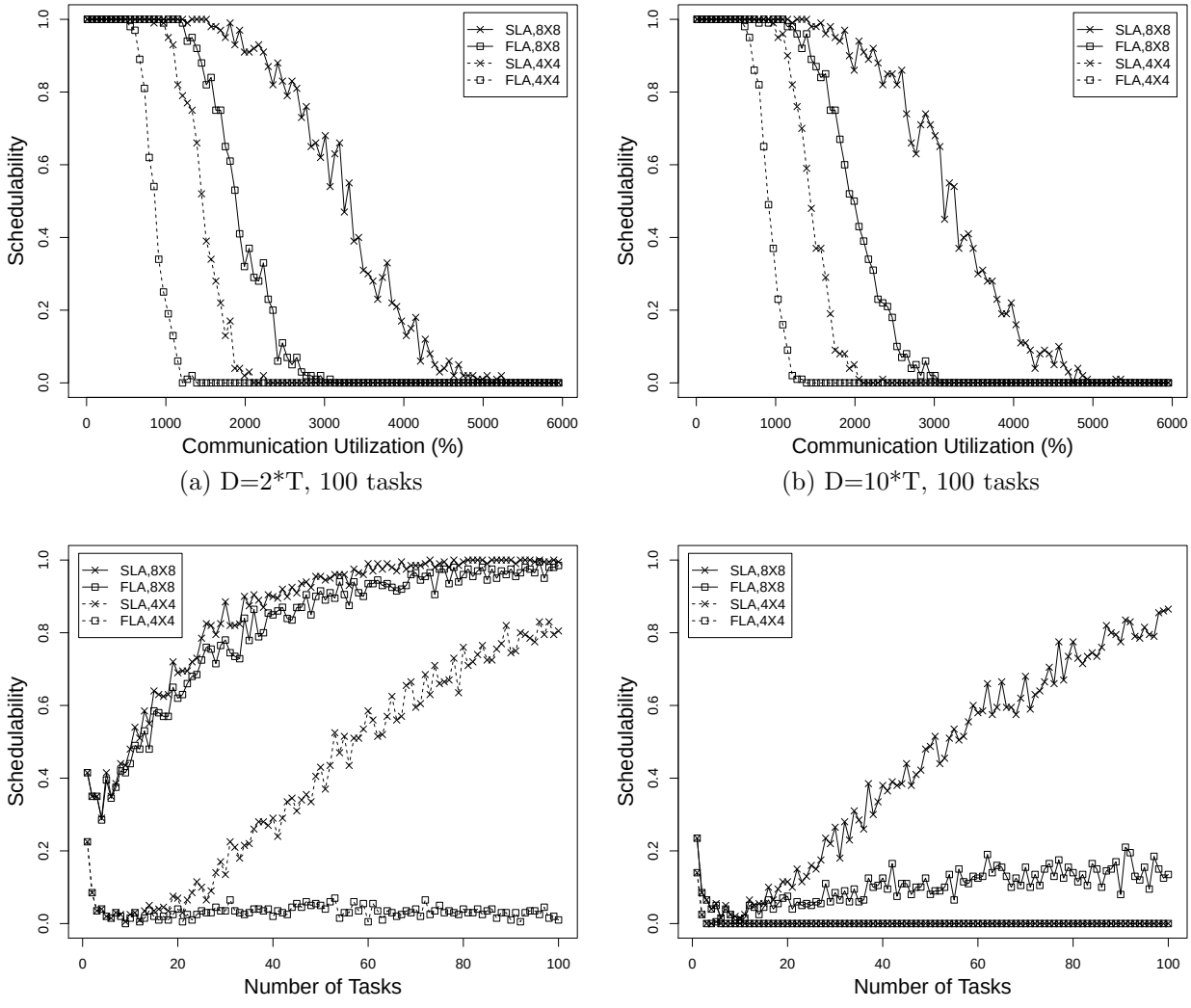
The experiment setup involves changing a number of factors to assess their effect on the analysis.

1. Number of communication tasks is in the range (1,100) in steps of 1.
2. Task period (or minimum interarrival time for sporadic tasks), T_i, is chosen in the range (1000,1000000) through a uniform random distribution.
3. Task deadline, D_i, is chosen as a multiple of the period, e.g., $2 * T_i$.
4. Communication utilization is varied in the range (10%, 6000%) in steps of 60%.This factor represents the utilization of the communication resources in the network. Full utilization of a single communication resource is represented by 100% utilization. The full utilization of a 4×4 mesh is represented by 2400% and for a 8×8 at 11200%, respectively.
5. Task release jitter, J_i^R, is set to zero.
6. An arbitrary priority assignment scheme is used for choosing task priorities.
7. Task mapping is random.
8. A shortest path algorithm is used to select paths for communication tasks between source and destination nodes.
9. A 100 test cases are generated for each possible configuration (40000 configurations).

We use the following metrics for evaluation:

- **Schedulability:** A communication task is unschedulable if: 1) its total WCL is larger than its deadline ($R_i > D_i$), or 2) no solution is found for the iterative analysis equation on any stage on the task's path. For the latter case, the stopping condition for the equation on any stage is when the latency computed in an iteration exceeds the deadline. A test case will be unschedulable if one of the tasks in its task set is unschedulable. The schedulability metric is a measure of the percentage of schedulable test cases for a particular configuration.
- **Improvement in WCLs:** For each test case, we compute the WCL of each task using both SLA and FLA. For a particular communication task, the improvement in the computed WCL bound is calculated as $1 - R_i^{SLA}/R_i^{FLA}$. We report the average improvement for a test configuration. This metric is only valid for schedulable tasks.

- **Fraction of tasks with improved latencies:** For a given test case, this metric shows the percentage of tasks that have a tighter WCL bound using SLA compared to FLA. This metric only applies to schedulable tasks.
- **Analysis time:** This is the time taken to compute the latency bounds for all tasks in a test case using both SLA and FLA. For any given configuration, we report the average analysis time over all test cases.



(a) D=2*T, 100 tasks

(b) D=10*T, 100 tasks

(c) U=1210%

(d) U=2410%

Figure 5.5: Schedulability results for SLA and FLA

Schedulability: Figures 5.5a and 5.5b show the schedulability against the communication utilization for task sets with 100 tasks and deadlines of $2 * T$ and $10 * T$, respectively. Both graphs demonstrate approximately the same trend. For a 4×4 NoC instance, both SLA and FLA are able to schedule task sets for very low utilizations. As the utilization

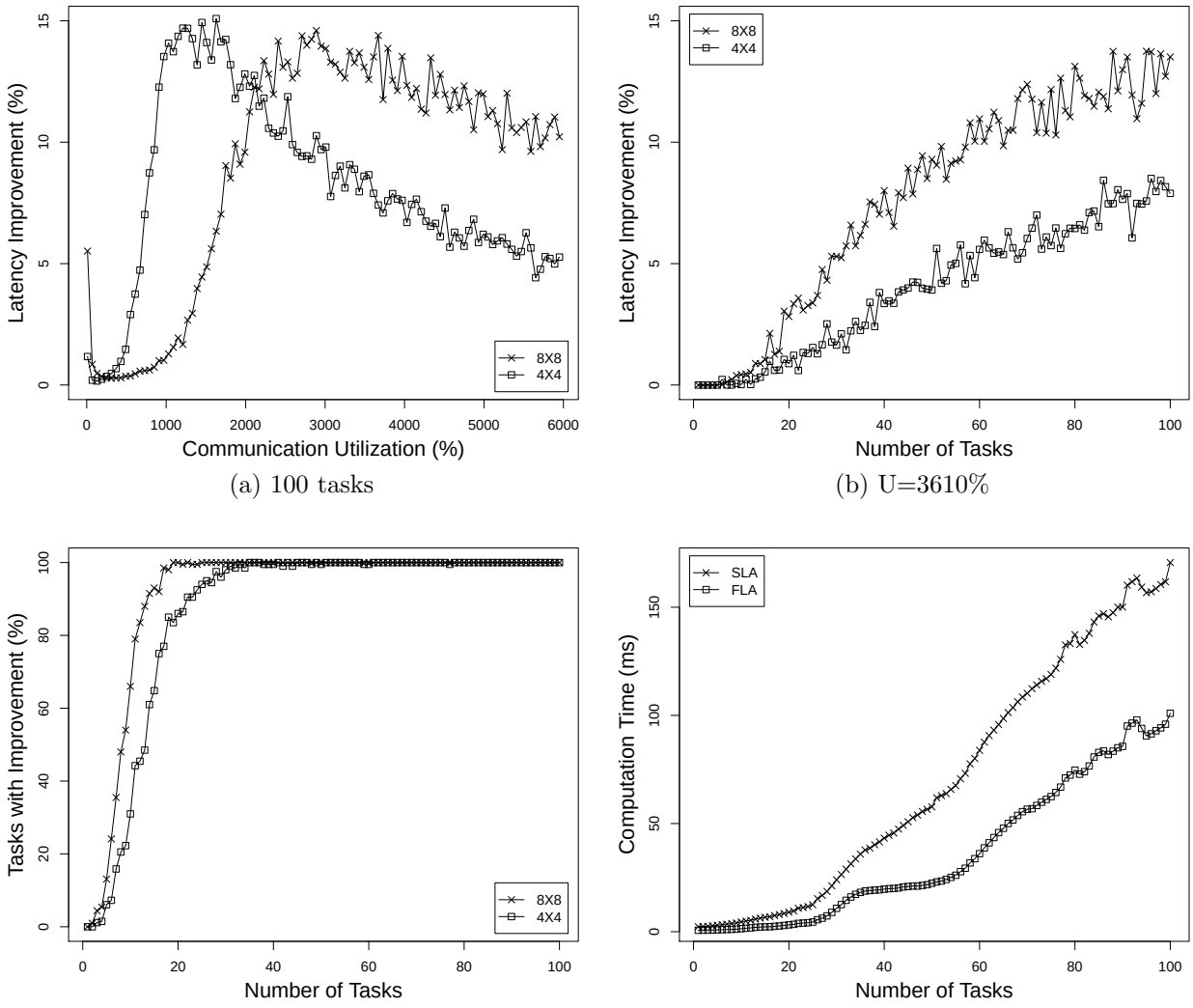
increases, SLA schedules more task sets compared to FLA. The reason is that SLA performs the analysis on the stage-level, thus reducing the latency bounds and increasing the schedulability of task sets. The same behavior is observed for an 8×8 NoC instance. However, the gap in schedulability between SLA and FLA widens compared to the 4×4 instances. This is because 8×8 NoC instances have more communication resources and tasks can traverse longer paths which leads to more interferences. Also higher communication utilizations means that tasks have larger basic stage latencies which leads to larger interferences. As interferences increase, the tightness of SLA manifests more and leads to a larger schedulability gap compared to FLA.

Figures 5.5c and 5.5d show the schedulability against the number of tasks for utilizations 1210% and 2410%, respectively. At a communication utilization of 1210%, SLA performs slightly better than FLA in terms of schedulability in 8×8 NoC instances. As the number of tasks increase, the ratio of schedulable task sets increases for both SLA and FLA. The reason is that the utilization is divided amongst more tasks, thus reducing the per task utilization and accordingly interferences, hence, scheduling more task sets. For a 4×4 NoC instance, SLA schedules more task sets than FLA which has a very low schedulability ratio even as the number of tasks increases. The reason is that at $U = 1210\%$ in 4×4 NoC instances, FLA is still not able to schedule all tasks due to the high interferences while SLA can perform better because it provides a tighter analysis. For a communication utilization of 2410%, both SLA and FLA cannot schedule any of the task sets with more than 10 tasks in 4×4 NoC instances due to high interferences. In an 8×8 NoC instance, SLA schedules more task sets compared to FLA as the number of tasks increases. This further demonstrates the tightness of SLA over FLA.

Latency Improvement: Figure 5.6a shows the latency improvement against the communication utilization for task sets with 100 tasks. As the utilization increases, the improvement in latencies computed by SLA over FLA increases to about 15%. The reason is that increasing the utilization, increases the interferences between tasks, leading to a wider gap in the computed latency bounds between SLA and FLA. This happens up to a certain utilization turning point, after which the improvement starts decreasing again as the utilization increases. This is because beyond that turning point, more interferences cause more tasks to be unschedulable. This turning point is at a higher utilization value in 8×8 NoC instances due to the existence of more communication resources.

Figure 5.6b shows the latency improvement against the number of tasks for a communication utilization of 3610%. Increasing the number of tasks, increases the interferences, thus, leading to a higher improvement in latency bounds. The improvement in latency is higher in 8×8 NoC instances because more tasks can be scheduled due to the presence of more communication resources.

Figure 5.6c shows the ratio of tasks with improved latency bound against the number of tasks at a communication utilization of 3610%. As the number of tasks increases, more interference occurs, and more tasks have an improved latency bound using SLA compared to FLA. The percentage of tasks with improved latency increases as the number of tasks increases, and reaches approximately a 100%.



(a) 100 tasks

(b) U=3610%

(c) U=3610%

(d) Computation time

Figure 5.6: Latency and computation time results for SLA and FLA

Computation Time: Figure 5.6d compares the average computation times of both SLA and FLA. The analysis time for SLA is approximately double that of FLA. We find this to be reasonable for the quality of results delivered by SLA. This is acceptable since SLA performs the analysis at the stage-level compared to the flow-level using FLA.

Summary: For all $4,000,000$ test cases, the WCL is reduced on average by 5.2%, and the number of schedulable test cases is increased by 34.0%. Any task that is schedulable using FLA is also schedulable using SLA. The ratio of latency bounds (SLA to FLA) is less that or equal 1.0 for all schedulable tasks. This means that SLA is at worst the same as FLA, which verifies our tightness analysis. The analysis time of SLA is on average double that of FLA. The average analysis time over all test cases for SLA is 71.0 mS and 36.4 mS for FLA. We use the Wilcoxon matched pairs test to reason about the significance of our

results. The p-value for both the analysis and the schedulability is less than 2.2×10^{-16}.

5.11 Summary

This chapter presents a stage-level WCL analysis for pipelined communication interconnects. The proposed SLA accounts for direct and indirect interference that arise from interferences between tasks. We analytically show that SLA will in the worst-case provide results that are equivalent to that of FLA, but otherwise provide tighter estimates. We illustrate this with an example that for a fixed topology, task mapping, and their respective paths, the number of schedulable tasks when using SLA is higher than FLA. Our results show an average improvement over FLA in the WCL analysis by approximately 5% and in schedulability of tasks by 34%.

Chapter 6

Offset-based WCRT Analysis for CMPs

Hard real-time applications must guarantee that their temporal requirements are met at all times. This requires a WCRT analysis, which provides a method to compute the upper-bounds on the amount of time it takes an application to complete execution. Such an analysis is essential in determining whether a hard real-time application meets its application deadlines. If it does, then the application is deemed schedulable; otherwise, unschedulable. The requirement to deliver tight WCRT estimates is paramount when developing such an analysis because it improves schedulability. Therefore, researchers proposed various WCRT techniques aiming to provide tight and accurate WCRTs of such distributed hard real-time systems [146, 108, 89].

These efforts make the fundamental assumption that the communication occurs over a single shared bus interconnect. A shared bus interconnect consists of a single communication resource that only allows mutually exclusive access. This presents a traditional communication resource model, but, it does not apply to computing platforms prevalent today. Nowadays, platforms consist of multi-processor systems with multiple processing resources that are typically connected using communication resources such as a NoC. Modelling the interconnect as a single shared bus interconnect does not accurately model the communication resources available in such platforms. Furthermore, it does not capture the pipelined nature of the communication resources that allow for parallel transmission of data between processing resources across multiple stages of the communication resources. This prohibits accurately predicting the latencies offered by communication resources such as NoCs resulting in gross over-estimates for the WCRTs. Hence, recent research [64] focuses on deriving tight worst-case latencies on multi-processor platforms by considering the pipelined nature of the communication resources.

We extend both FLA and SLA by introducing dynamic offsets for dependencies between computation and communication tasks modelled using a directed acyclic graph (DAG) [65]. Section 6.2 presents a holistic analysis for computing the dynamic offsets and jitters. In Section 6.4, we present the theory for offset-based flow-level analysis (OFLA) which includes two variants of the analysis. The first is an exponential analysis, and the second is

a polynomial one. We also propose offset-based stage-level analysis (OSLA) in Section 6.3, a WCRT analysis technique that extends SLA. OSLA extends SLA to use dynamic offsets to accurately model the dependencies between the computation and communication tasks of an application. We present the theory behind OSLA, which also includes two variants of the analysis; exponential and polynomial. For both OFLA and OSLA, we apply the approximations proposed by Maki-Turja and Nolin [89] to the polynomial analyses.

6.1 System Model

In this section, we describe our task model. We use the same processing and communication resource model used for SLA as in Section 5.1. We also present an overview of offsets and jitters of tasks.

6.1.1 Task Model

In this chapter, we compute the WCRTs of applications while we computed the WCLs of communication tasks in Chapter 5. Hence, our task model focuses on characterizing applications including both computation and communication tasks. This requires extending the task model presented in Section 5.2.

Definition 13 (Real-time system)**.** *A real-time system is a set of n applications* $\mathcal{A} := \{A_1, A_2, ..., A_n\}$ *where each application* $A_i \in \mathcal{A}$ *is denoted by a 4-tuple* $\langle G_{A_i}, D_i, T_i, J_i^R \rangle$*. This describes an application* A_i *with task graph* G_{A_i}*, end-to-end deadline* D_i*, period* T_i*, and release jitter* J_i^R*.*

An application is a DAG with a task graph $G_{A_i} = \langle \Gamma_i^C, \Gamma_i^M \rangle$ consisting of a set of nodes Γ_i^C that represent computation tasks and a set of edges Γ_i^M that represent communication tasks, respectively.

Definition 14 (Computation task)**.** *A computation task of* A_i*,* $\tau_{ik} \in \Gamma_i^C$ *is denoted by a 3-tuple* $\langle C_{ik}, v_{c_{ik}}, P_{ik} \rangle$*. This describes a task* τ_{ik} *that has priority* P_{ik} *and a WCET* C_{ik} *when executed on some processing resource* $v_{c_{ik}}$*, and priority* P_{ik}*.*

Definition 15 (Communication task)**.** *A communication task of* A_i*,* $\tau_{ik} \in \Gamma_i^M$ *is denoted by a 3-tuple* $\langle L_{ik}, \delta_{ik}, P_{ik} \rangle$*. This describes a task* τ_{ik} *that has priority* P_{ik} *and a WCL* L_{ik} *when transmitting data across a series of contiguous communication resources* δ_{ik} *between the source and destination processing resources:* $v_{s_{ik}}$ *and* $v_{d_{ik}}$*.*

The worst-case transmission latency L_{ik} is the latency that an instance of the task τ_{ik} takes to transmit data on a single communication resource when it does not suffer interferences from any other tasks. Assuming a stage delay of one cycle for clarity, the worst-case transmission latency of task τ_{ik} along its path is $C_{ik} = L_{ik} + |\delta_{ik}| - 1$ where $|\delta_{ik}|$ is the number of stages on the path δ_{ik} of communication task τ_{ik}. Note that if a communication task τ_{jl} interferes with another task τ_{ik} on a non-contiguous set of stages,

then the interfering task τ_{jl} must be split into two or more tasks such that each of the new tasks interferes with task τ_{ik} on a contiguous set of stages. We use the notation τ_{ikc} to refer to the c-th instance (job) of task τ_{ik}. For clarity of presentation, we use the function schedule $\Theta(t, s)$ to denote the assignment of data units of the various jobs to the communication resource. More specifically, each time slot t on a stage s of the communication resources is either transmitting a datum or is idle.

Regarding a NoC implementation, computation tasks will execute on the processing elements of the NoC. Communication tasks are messages communicated between the computation tasks. Messages are transmitted on a set of contiguous links that form paths between the processing nodes.

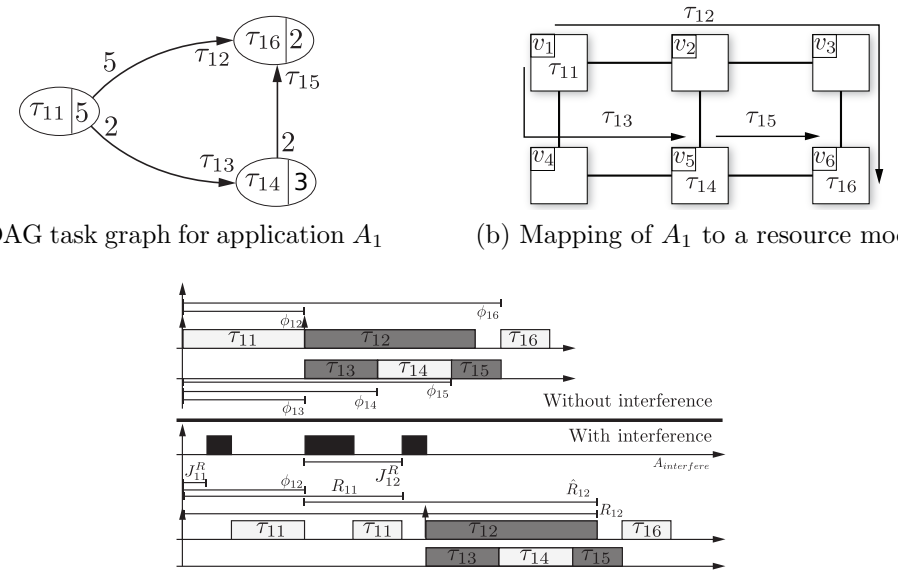
We restrict the task graph to be single rooted, and for the root to be a computation task that is activated at the application's period, and has a maximum release jitter J_i^R. The release jitter J_i^R is the worst-case delay in the release time of the application or the first task in the application's task graph. The exit task is also a computation task of the task graph without any successor computation tasks. Application A_i is schedulable if and only if the WCRT of each exit task is less than or equal to the deadline D_i. Notice that a communication task enforces precedence constraints between other computation and communication tasks. For example, a communication task τ_{ik} executes only after its source computation task completes execution. The destination computation task only begins after both the source computation task and the corresponding communication task complete execution. We do not place any restrictions on the deadlines, the release jitters, and the periods such that the deadline and/or the release jitter can be larger than the period. We assume distinct priority assignment to the tasks, but, we do not enforce any specific priority assignment to the computation and communication tasks, i.e., the priority assignment does not have to follow the order or precedence of the tasks in the task graph G_{A_i}.

Figures 6.1a and 6.1b present an illustrative example of a DAG as the task graph for an application A_1 and its mapping onto a pipelined mesh communication resource model. Ellipses identify computation tasks, and arrows represent communication tasks. For example, computation task τ_{11} has a WCET of 5 time units, and communication task τ_{13} has a WCL of 2 time units. Figure 6.1b shows the mapping of A_1 onto a 2×3 mesh. As an example, τ_{11} and τ_{16} are mapped onto v_1 and v_6, respectively, with a communication task τ_{12} that traverses the path $\langle (v_1, v_2), (v_2, v_3), (v_3, v_6) \rangle$.

6.1.2 Offsets and Jitters

Our task graph represents precedence dependencies between computation tasks through communication tasks. We use offsets and jitters to ensure that these precedence constraints are satisfied. We also support dynamic offsets as introduced by Palencia and Gonzalez [108].

The application's root task is activated periodically with a period T_i. Each task τ_{ik} in the application is activated after a specific time interval from the activation of the root vertex. We call this time interval, offset ϕ_{ik}, which is the best-case release time of task τ_{ik}. This occurs when the preceding tasks execute for their WCET without suffering



(a) DAG task graph for application A_1

(b) Mapping of A_1 to a resource model

(c) Example timing diagrams with and without interference

Figure 6.1: Illustrative example application A_1: Task graph and its mapping

interferences from higher priority tasks. This assumes that a task's best-case execution time is equal to its WCET without interference. The offset for the root task is zero. A task's offset is equal to the sum of the WCETs of the tasks along the path leading to that task starting from the root vertex. The offset of a specific job instance τ_{ikc} is measured from the activation of the corresponding instance of the root task, τ_{i1c}. There might exist multiple paths leading to a task, the offset, in that case, is equal to the maximum offset of all paths leading to that task. If interference exists, then the task release can be delayed from its best-case release time. The release jitter of a task is the maximum difference between its activation time and its release time, i.e., it is the difference between the best-case and the worst-case release times of the task. Again, if multiple paths lead to the task, then the release jitter is equal to the maximum jitter from all paths. The worst-case release time of a task τ_{ik}, from the activation of the root vertex, is the sum of its offset ϕ_{ik} and release jitter J^R_{ik}. Note that the release jitters of the tasks depend on their WCRTs, and the computation of WCRTs depends on the release jitters. This, therefore, requires iteratively computing WCRTs and assigning jitters until either a fixed point is reached or the application is unschedulable. In Section 6.2, we introduce a holistic analysis [146] that iteratively computes the jitters followed by the WCRT of each task.

Figure 6.1c shows a schedule for the task graph in Figure 6.1a to illustrate offsets, jitters, and response times. This schedule has two applications A_1 and an interfering application $A_{interfere}$. Application $A_{interfere}$ has a higher priority than A_1. Up arrows denote release times. The timing diagrams show execution sequences of A_1 without and with interference. Notice in Figure 6.1c (without interference), that τ_{12} can at best start after the WCET of task τ_{11} because, according to the task graph in Figure 6.1a, τ_{12} is dependent on τ_{11} completing its execution. Hence, the offset ϕ_{12} of task τ_{12} is its best case release time which is equal to 5 time units.

Jitters are used to model worst-case temporal dependencies among tasks. In the case with interference (lower diagram in Figure 6.1c), τ_{11} suffers a release jitter J_{11}^R of 1 time unit. Task τ_{11} releases at time 1; however, it could be released at any time from 0 up to and including its maximum release jitter, which is the application's release jitter $J_{i1}^R = J_i^R$. Although task τ_{11} is released at time 1, it experiences interference from a task in $A_{interfere}$ delaying its execution to time 2. Task τ_{11} suffers another interference from another task of $A_{interfere}$ at time 5 causing τ_{11} to delay its end time to 9. The worst-case response time R_{11} of τ_{11} is, thus, 9. Task τ_{12} releases at time 9, which is the end time for τ_{11}. The difference between the release time of τ_{12} and its offset ϕ_{12} provides the release jitter J_{12}^R. The release time of τ_{12} coincides with some other interfering task of $A_{interfere}$ such that it gets delayed even further. The WCRT of τ_{12} is denoted by R_{12}, which is 17. We use $\hat{R}_{12}$ to denote the WCRT of τ_{12} from its activation time. This is equal to 12 time units.

6.2 Holistic Analysis

Holistic analysis [146] presents an iterative technique to compute the offsets and jitters of tasks. This technique starts with an assignment of offsets and jitters, and computes the response times of the task, which then provide updates on jitters of following tasks. The iterative method is necessary because the offsets and jitters depend on the response times of preceding tasks. We use this holistic analysis [146] technique to combine the response times of computation tasks executing on processors, and computation tasks transmitting data across links on the NoC to determine end-to-end worst-case response times.

Algorithm 7 presents the key steps in our holistic analysis. We assume that the application set $\mathcal{A}$, the mapping of tasks to the resources, and the communication routes are given. The algorithm starts by computing initial offsets and jitters for each task. The first task τ_{i1} of each application has a release jitter equal to the application's release jitter ($J_{i1}^R = J_i^R$), and since the first task is the root vertex of the applications DAG, the task's offset is zero ($\phi_{i1} = 0$). All other tasks are assigned a jitter of zero and an offset derived using ComputeOffsets (see Section 6.2.1). Then, the algorithm computes the WCRT of every task in every application using ComputeWCRT. For communication tasks, ComputeWCRT uses either OFLA (Section 6.4) or OSLA (Section 6.3). For computation tasks, ComputeWCRT employs traditional offset-based WCRT analysis for tasks on uniprocessors [108]. Please see [108] for further details. Note that we restrict the set of higher priority tasks that interfere with a task under analysis τ_{ab} to those tasks that are mapped to the same processing element as τ_{ab}. The task jitter is derived using ComputeJitters (see Section 6.2.2), which uses the WCRT of the precedent tasks (denoted by pre). The algorithm iterates until it reaches a fixpoint. This is when the jitters and WCRTs do not change, or a task misses the application deadline. Note that if no deadline is missed, then the algorithm is guaranteed to converge to a fixpoint because the WCRT analysis is monotonic in the release jitters [108], i.e., increasing the jitters can not cause the computed WCRT for any task to decrease. Finally, the WCRT of the application is obtained using ComputeAppWCRT as the maximum WCRT among all exit tasks of the application. An unschedulable application is one whose WCRT is larger than the deadline.

Function 7 Holistic analysis

Input: Application set $\mathcal{A}$, mapping of tasks from $\mathcal{A}$ to resources, and communication routes.
Output: Offsets, jitters, WCRT for each task, and end-to-end WCRT for all applications.

```
1:
2: for all A_i ∈ 𝒜 do
3:    Offsets(τ_i1) ← 0
4:    Jitters(τ_i1) ← J_i^R
5:    for all τ_ik ∈ Γ_i^C ∪ Γ_i^M, k > 1 do
6:       Offsets(τ_ik) ← ComputeOffsets(τ_ik)
7:       Jitters(τ_ik) ← 0
8:    end for
9: end for
10: for all A_i ∈ 𝒜 do
11:    while ¬fixpoint(Jitters, ResponseTime) do
12:       for all τ_ik ∈ Γ_i^C ∪ Γ_i^M do
13:          ResponseTime(τ_ik) ← ComputeWCRT(τ_ik)
14:          Jitters(τ_ik) ← ComputeJitters(τ_ik, ResponseTime(pre(τ_ik))
15:          if ResponseTime(τ_ik) > D_i then
16:             Unschedulable(A_i) ← TRUE
17:             break while loop
18:          end if
19:       end for
20:    end while
21: end for
22: for all A_i ∈ 𝒜, ¬Unschedulable(A_i) do
23:    AppResponseTime(A_i) ← ComputeAppWCRT(A_i)
24: end for
```

6.2.1 ComputeOffsets: Computing Offsets

Unlike prior works [108, 89] that only support a linear temporal dependency between tasks of a transaction, we support DAGs, which allow for parallel dependencies between tasks. Consequently, we compute offsets for tasks as follows.

$$\phi_{ik} = \max_{\forall \rho_{ik} \in \Pi_{ik}} \sum_{\forall \tau_{ij} \in \rho_{ik}} C_{ij} \tag{6.1}$$

where ϕ_{ik} is the offset for task τ_{ik}, Π_{ik} is the set of all paths in the application's DAG, G_{A_i}, starting from τ_{i1} to τ_{ik}, and ρ_{ik} is the set of tasks in an individual path of Π_{ik}. There may exist multiple paths to a particular task; hence, the computed offset must consider the worst-case scenario over all possible paths leading to that task. This amounts to the summation of WCETs/WCRTs of tasks (C_{ij}) for each path, and finding the path with the largest value (the *max* operation). For example, there are two paths that lead to task τ_{16} in Figure 6.1a. The offset for τ_{16} is the maximum between path $\langle \tau_{11}, \tau_{12} \rangle$ and path $\langle \tau_{11}, \tau_{13}, \tau_{14}, \tau_{15} \rangle$. Using Equation 6.1 and a stage delay of one time unit, we select the worst-case offset between 12 and 13, respectively, leading to an offset ϕ_{16} that is equal to 13.

6.2.2 ComputeJitters: Computing Jitters

Similar to offsets, we compute the release jitter of every task (excluding the root task) by incorporating the WCRTs of preceding tasks on the path, and the application's release jitter. The release jitter J_{ik}^R of task τ_{ik} is defined as follows:

$$J_{ik}^R = \max_{\forall \rho_{ik} \in \Pi_{ik}} \Big(\sum_{\forall \tau_{ij} \in \rho_{ik}} R_{ij} \Big) - \phi_{ik} \tag{6.2}$$

Equation 6.2 selects the maximum of the sum of WCRTs in every path from the root task τ_{i1} to task τ_{ik}; thereby, selecting the largest of the WCRT contributions.

6.3 Offset-based Stage-Level Analysis

OSLA computes the WCRT for communication tasks by considering the pipelining and parallel data transmission of jobs on the pipelined communication resources. We present an exponential and a polynomial analysis. We also prove that each of these analyses gives a safe upper bound for the WCRT of the communication task under analysis.

6.3.1 Direct and Indirect Interference

We use the same definitions of direct and indirect interference as in Sections 5.3 and 5.5, respectively. We use the symbol $\mathbb{S}_i^D(\tau_{ab})$ to denote the set of tasks of application A_i that directly interfere with task τ_{ab} along its path δ_{ab}. We use the symbol $\mathbb{S}_{i_s}^D(\tau_{ab})$ to denote the set of tasks of application A_i that directly interfere with task τ_{ab} on stage s of its path δ_{ab}. The indirect interference set $\mathbb{S}_{ij}^I(\tau_{ab})$ is the set of tasks indirectly interfering with task τ_{ab} through the intermediate task τ_{ij}. Tasks in the indirect interference set $\mathbb{S}_{ij}^I(\tau_{ab})$ do not share any stage with task τ_{ab}, but they must still be considered in the analysis because they can delay τ_{ij}. We account for indirect interference by computing an indirect interference jitter term $J_{ij}^I(\tau_{ab})$ for task τ_{ij} in Section 6.3.4. The indirect interference jitter $J_{ij}^I(\tau_{ab})$ is then summed to the release jitter J_{ij}^R to obtain the maximum jitter suffered by task τ_{ij} before reaching the stage on which it causes interference to task τ_{ab}.

6.3.2 Derivation of a Response Time Estimate

We mentioned in Section 5.1 that the task transmission schedule is divided into time slots. The time slot t is the time interval $[t, t+1)$ (we use both notations interchangeably). We also explained in Section 6.1 that if a data unit of τ_{ab} is transmitted in the interval $[t, t+1)$ on a stage s_1, then it becomes ready for transmission in the interval $[t+1, t+2)$ on the next stage s_2 of δ_{ab}. When a data unit becomes ready for transmission in a slot t, then it will actually be transmitted in that slot unless it is preempted by a higher priority data unit. For clarity of presentation, as mentioned in Section 6.1, we use the function schedule

$\Theta(t, s)$ to denote the assignment of a data unit of a job τ_{abc} of task τ_{ab} to a particular slot t on stage s.

We focus on deriving an upper-bound $\hat{R}_{ab}$ to the response time of the task under analysis τ_{ab}. Since indirect interferences are accounted for by indirect interference jitter, for clarity, we only consider directly interfering tasks. The ordered set of stages $\{s_1, \ldots, s_{|\delta_{ab}|}\}$ comprises the stages traversed by τ_{ab} along its path δ_{ab} where $|\delta_{ab}|$ is the number of stages in δ_{ab}.

In our derivation, we first consider the transmission of any job τ_{abc} of task τ_{ab} in any valid schedule $\Theta(t, s_k)$ on all stages of δ_{ab}. We discuss how to compute an upper-bound $\hat{R}_{ab}$ to the response time of τ_{abc} for that specific schedule; for clarity, we measure the response time from the activation time of τ_{abc} itself. Next, we show that the upper-bound can be maximized by modifying the pattern of release times of jobs of the tasks in $\mathbb{S}_i^D(\tau_{ab})$, as well as the jobs of τ_{ab} itself. Finally, we show that independent of the schedule (e.g., $\Theta(t, s_k)$) and the specific job instance c of τ_{ab}, the proposed release time modification always yields a pattern within a finite set of *critical activation patterns*. Hence, we can derive a safe response time upper-bound for τ_{ab} by computing the maximum value of the response time upper-bound over all critical activation patterns.

We first introduce two model transformations to help us in our discussion. The transformations do not alter the transmission semantic of the model, but they simplify reasoning about the correctness of our proposed analysis. Note that based on our resource model, we consider that jobs of the same communication task are transmitted in FIFO order. Hence, when analyzing the job under analysis τ_{abc}, we simply assume that the priority of any job $\tau_{abc'}$ that follows τ_{abc}, i.e., with $c' > c$, is lower than the priority of τ_{abc}. The second transformation involves the schedule $\Theta(t, s_k)$. Note that if any job τ_{ijp} is released on stage s_k at time t, the job can not start executing on a successive stage s_{k+l} (with $k + l \leq |\delta_{ij}|$) before time $t + l$. Therefore, release times can not be directly compared across different stages. To solve this issue, we define a *stage-normalized schedule* where the transmission schedule on successive stages is moved earlier in time so that release times coincide across all stages.

Definition 16 (Stage-normalized schedule)**.** *Given a schedule $\Theta(t, s_k)$ over all stages in δ_{ab}, the corresponding stage-normalized schedule is $\bar{\Theta}(t, s_k) = \Theta(t + k - 1, s_k)$.*

An example of a stage-normalized schedule is shown in Figure 6.3. The reported schedule is the stage-normalized version of the schedule presented in Figure 6.2, and it will be used as a running example throughout this section. Up arrows in both figures represent release times, and the job under analysis, τ_{abc}, is shown in black Any datum transmitted at time t on stage s_2 in Figure 6.3 is transmitted at time $t + 1$ in Figure 6.2; any datum transmitted at time t in s_3 in Figure 6.3 is transmitted at time $t + 2$ in Figure 6.2; and so on. Also note that as a consequence of this model transformation, a datum transmitted in time slot t on stage s_k will now be transmitted in the same slot t on stage s_{k+1} if the schedule is not busy transmitting a higher priority datum. This property will significantly simplify the proofs of Lemmas 7 and 9, since it allows us to compare the busy/idle state of the schedule on two stages s_l, s_q independent of the distance $q - l$ between the stages.

We next formalize the concept of a busy interval on stage s_k, which is common to analysis of systems with fixed-job priority.

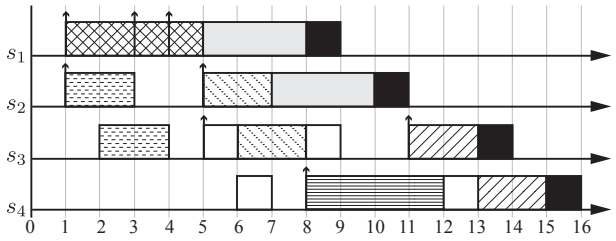


Figure 6.2: An example schedule (up arrows are release times)

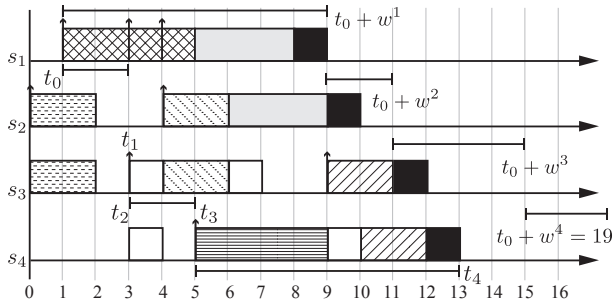


Figure 6.3: An example of the stage-normalized schedule

Definition 17 (P_{abc}-level busy interval)**.** *We say that $[t, t')$ is a P_{abc}-level busy interval on stage s_k in $\bar{\Theta}$ if the stage-normalized schedule continuously transmits jobs with priority greater than or equal to P_{abc} in slots $\langle t, \ldots, t' - 1 \rangle$ and τ_{abc} is not completed before slot $t' - 1$. An interval $[t, t')$ that is not P_{abc}-level busy is then P_{abc}-level idle.*

P_{abc}-level busy intervals allow us to determine the interference caused by higher priority jobs on τ_{abc} in every stage s_k of δ_{ab}. For simplicity and since lower-priority jobs do not affect the schedule of τ_{abc} in any way, assume that all jobs in Figure 6.3 have higher priority than the job under analysis. Similarly, note that by definition, jobs that arrive on stage s_k after τ_{abc} finishes executing on that stage do not need to be included in the busy interval. Then as an example, $[0, 2)$, $[3, 7)$ and $[9, 12)$ are all maximal-length P_{abc}-level busy intervals on s_3. Also note that $[3, 5)$, as well as any other interval contained in a maximal-length busy interval, is a P_{abc}-level busy interval by itself. In single-resource systems, the concept of busy interval helps the analysis because the response time of a job is necessarily bounded by the length of the unique busy interval in which it appears. However, in our situation, job τ_{abc} is transmitted within different P_{abc}-level busy intervals on each stage. To effectively use the concept of a busy interval, we thus define a new abstraction, called a *busy chain*, which concatenates P_{abc}-level busy intervals across all stages. As we will later prove, the busy chain is defined in such a way that each higher priority job can interfere with τ_{abc} only once.

Definition 18 (Busy chain)**.** *Let $t_{|\delta_{ab}|}$ be the time at which the job under analysis τ_{abc} finishes execution on its last stage $s_{|\delta_{ab}|}$ in $\bar{\Theta}$, i.e., its last datum is transmitted in slot*

$t_{|\delta_{ab}|}-1$ *(interval* $[t_{|\delta_{ab}|}-1, t_{|\delta_{ab}|})$*). For each stage* s_k *in* δ_{ab} *starting from* $s_{|\delta_{ab}|}$ *and towards* s_1*, let* t_{k-1} *be the earliest possible time such that* $[t_{k-1}, t_k)$ *is a* P_{abc}*-level busy interval on* s_k*. If the stage-normalized schedule is not* P_{abc}*-level busy in slot* t_k-1 *on* s_k*, then* $t_{k-1}=t_k$*. Hence,* $\langle [t_0,t_1),\ldots,[t_{|\delta_{ab}|}-1, t_{|\delta_{ab}|})\rangle$ *is the busy chain for* τ_{abc} *in* $\bar{\Theta}$*.*

It is easy to see that every job τ_{abc} admits a unique busy chain for a given stage-normalized schedule $\bar{\Theta}$. An example of a busy chain is reported in Figure 6.3. Starting from stage s_4 where $t_4=13$, the earliest time for which we have a P_{abc}-level busy interval on stage s_4 ending in t_4 is the time $t_3=5$. On stage s_3, we again find the largest time interval $[t_2,t_3)$ such that it is a P_{abc}-level busy interval. This time is $t_2=3$. On stage s_2, the time slot $[2,3)$ is P_{abc}-level idle. This means that there is not a P_{abc}-level busy interval on stage s_2 that ends at time $t_2=3$, hence, $t_1=t_2=3$. On stage s_1, the time t_0 that creates the longest P_{abc}-level busy interval ending in $t_1=3$ is $t_0=1$. Therefore, the busy chain for job τ_{abc} is $\langle[1,3),[3,3),[3,5),[5,13)\rangle$. Note that since t_{k-1} is defined as the earliest possible time such that $[t_{k-1},t_k)$ is a P_{abc}-level busy interval on s_k, it follows that $\bar{\Theta}$ must be P_{abc}-level idle in slot $t_{k-1}-1$ (interval $[t_{k-1}-1,t_{k-1})$) on stage s_k. In Figure 6.3, the schedule is always idle before $t_0,\ldots,t_3$ in stages $s_1,\ldots,s_4$, respectively. Finally, to show that the definition is consistent, we prove that the busy chain always contains the execution of τ_{abc}, i.e., the finishing time of τ_{abc} on any stage s_k is no earlier than the start time t_{k-1} of the busy chain on that stage.

Lemma 6. *Let* t_k^f *be the time at which the job under analysis* τ_{abc} *finishes executing on stage* s_k *in* $\bar{\Theta}$*, then* $t_k^f \geq t_{k-1}$*.*

Proof. The lemma proceeds by induction on the stage number k, starting from $k=|\delta_{ab}|$ and proceeding backwards until $k=1$. Note that according to Definition 18, $t^f_{|\delta_{ab}|}=t_{|\delta_{ab}|}\geq t_{|\delta_{ab}|-1}$, hence the base case immediately follows.

Assuming that the hypothesis holds for each stage up to stage s_{k+1}, we now show that it also holds for stage s_k. By definition of a busy chain, the schedule on stage s_{k+1} is P_{abc}-level idle in $[t_k-1,t_k)$. Since $t^f_{k+1}\geq t_k$, this implies that no job with priority greater than or equal to τ_{abc} is transmitted in slot t_k-1 on stage s_{k+1}. Hence, all data of τ_{abc} must be transmitted on stage s_k in slots t_k or after; otherwise, a datum of τ_{abc} would be transmitted in slot t_k-1 on stage s_{k+1} and the slot would not be P_{abc}-level idle. This implies that $t_k^f\geq t_k\geq t_{k-1}$, concluding the proof. □

Let $\hat{t}$ be the activation time of τ_{abc}. The response time of τ_{abc} in $\bar{\Theta}$ is $t_{|\delta_{ab}|}-\hat{t}=(t_{|\delta_{ab}|}-t_0)-(\hat{t}-t_0)$. We can thus obtain an upper-bound on the response time of τ_{abc} by fixing $\hat{t}$ and t_0 and computing an upper-bound on the length of the busy chain $t_{|\delta_{ab}|}-t_0$. This is similar to how the maximum length of the busy interval is used to bound response time in [108]. The following lemma proves the key property of the busy chain for interfering higher priority jobs.

Lemma 7. *Consider the busy chain of* τ_{abc} *in* $\bar{\Theta}$*. A given datum of any job* τ_{ijp} *cannot be transmitted both within the* P_{abc}*-level busy interval* $[t_{l-1},t_l)$ *on* s_l *and within the* P_{abc}*-level busy interval* $[t_{q-1},t_q)$ *on* s_q *with* $q>l$*.*

Proof. The proof is similar to the induction step for Lemma 6. Note that since τ_{ijp} is executed in the busy chain, it must hold that $P_{ijp} > P_{abc}$. Consider a datum of τ_{ijp} that is transmitted on stage s_l in slot t'_l with $t_{l-1} \leq t'_l < t_l$. In the stage-normalized schedule, the datum becomes ready on s_{l+1} at the same time $t'_l < t_l$. By definition of a busy chain, the schedule on s_{l+1} is P_{abc}-level idle in $[t_l - 1, t_l)$; since $t^f_{l+1} \geq t_l$ according to Lemma 6, this implies that no job with priority greater than or equal to τ_{abc} is transmitted in slot $t_l - 1$ on stage s_{l+1}. Hence, the datum must be transmitted on stage s_{l+1} in a slot $t'_{l+1} < t_l - 1$; otherwise, it would be transmitted at $t_l - 1$ and the slot would not be P_{abc}-level idle. If $q = l+1$, this concludes the proof; otherwise, note that $t'_{l+1} < t_l - 1$ implies $t'_{l+1} < t_{l+1}$. We can then repeat the same argument to show that on stage s_{l+2}, the datum is transmitted in slot $t'_{l+2} < t_{l+1} - 1$. By induction, we can then obtain $t'_q < t_{q-1} - 1$, concluding the proof. □

Intuitively, Lemma 7 implies that every datum of an interfering job τ_{ijp} needs to be counted only once towards the length of the busy chain of τ_{abc}. However, different data units of the same job can be transmitted within the busy chain on different stages. For example in Figure 6.3, the first datum of the task released at time 3 on s_3 is transmitted within $[t_2, t_3)$ on stage s_3, while the second datum is transmitted within $[t_3, t_4)$ on s_4. We use this property to compute the desired upper-bound to $t_{|\delta_{ab}|} - t_0$ in Lemmas 8 and 9. These two lemmas provide a way to compute the length of interfering jobs on each stage.

Definition 19 (Interfering job set)**.** *Let $\underset{k}{S}^J$ be the set of all jobs with priority higher than or equal to P_{abc} that are transmitted on stage s_k, with the exclusion of τ_{abc} itself.*

Definition 20 (Workload)**.** *Let $\bar{W}_{\underset{k}{S}^J}(t, t')$ be the sum of the transmission times of all jobs in set $\underset{k}{S}^J$ that are released in the interval $[t, t')$ in the schedule $\bar{\Theta}$.*

Note that the set $\underset{k}{S}^J$ does not include the job under analysis τ_{abc}, but it includes any previous jobs of task τ_{ab}.

We are now ready to compute an upper bound on the length of the busy chain $t_{|\delta_{ab}|} - t_0$. Our methodology works by induction: we first compute the maximum length of the busy chain segment $[t_0, t_1)$ on stage s_1 in Lemma 8 (base case). We then compute an upper bound to $[t_0, t_k)$ for each stage s_k in Lemma 9 (induction step), up to stage $s_{|\delta_{ab}|}$. The main intuition is that the busy chain is formed by a sequence of P_{abc}-level busy intervals on each stage; hence, on each stage s_k, the transmission times of jobs in $\underset{k}{S}^J$ must be sufficient to continuously transmit in the interval $[t_{k-1}, t_k)$. We thus bound the length of $[t_0, t_k)$ by computing the sum of transmission times of jobs that can be transmitted within continuous P_{abc}-level busy intervals up to stage s_k. We will show that we can use the defined workloads $\bar{W}_{\underset{1}{S}^J}, \ldots, \bar{W}_{\underset{k}{S}^J}$ to compute such sum (note that workloads are defined based on release times of jobs, not when they are transmitted); furthermore, since Lemma 7 stipulates that a job datum cannot contribute to the busy chain on more than one stage, we will need to ensure that each job's transmission time is counted only once.

Lemma 8. *Consider the busy chain of τ_{abc} in schedule $\bar{\Theta}$. Then for any value Δ such that:*

$$\Delta = L_{ab} + \bar{W}_{\underset{1}{S}^J}(t_0, t_0 + \Delta),$$

Δ is an upper-bound to $t_1 - t_0$.

Proof. Note that by definition of a busy chain, the schedule is P_{abc}-level busy on s_1 in $[t_0, t_1)$ and P_{abc}-level idle in $[t_0 - 1, t_0)$. Hence, no job in $\underset{1}{S}^J$ released on s_1 before t_0 can be transmitted in slot t_0 or after. Furthermore, clearly no job in $\underset{1}{S}^J$ released at or after $t_0 + \Delta$ can be transmitted in $[t_0, t_0 + \Delta)$. Therefore, $\Delta = L_{ab} + \bar{W}_{\underset{1}{S}^J}(t_0, t_0 + \Delta)$ is the sum of the transmission time of all jobs in $\underset{1}{S}^J$ that can be executed in $[t_0, t_0 + \Delta)$, plus τ_{abc} itself. We consider three possible cases:

1. There is a P_{abc}-level idle slot in $[t_0, t_0 + \Delta)$. Then by definition $t_1 < t_0 + \Delta$.
2. The schedule is P_{abc}-level busy in $[t_0, t_0 + \Delta)$ and τ_{abc} is released in $[t_0, t_0 + \Delta)$. Then it follows that τ_{abc} must finish exactly at $t_0 + \Delta$ and the schedule is P_{abc}-level idle after $t_0 + \Delta$, implying $t_1 = t_0 + \Delta$.
3. The schedule is P_{abc}-level busy in $[t_0, t_0 + \Delta)$ and τ_{abc} is not released in $[t_0, t_0 + \Delta)$. This is impossible, since $\bar{W}_{\underset{1}{S}^J}(t_0, t_0 + \Delta) < \Delta$ implies that there are not enough data units of jobs in $\underset{1}{S}^J$ to transmit continuously in the interval $[t_0, t_0 + \Delta)$ without considering τ_{abc} itself.

In summary, considering the first two cases, we have $t_1 \leq t_0 + \Delta$, concluding the lemma. □

Since we are interested in the tightest possible upper-bound to the response time of τ_{abc}, we simply compute the minimal value w^1 of Δ for which Lemma 8 holds as:

$$w^1 = \min\{\Delta | \Delta = L_{ab} + \bar{W}_{\underset{1}{S}^J}(t_0, t_0 + \Delta)\}. \tag{6.3}$$

As an example, when we apply Lemma 8 to the stage-normalized schedule in Figure 6.3, we obtain $w^1 = 8$ and thus $t_0 + w^1 = 9$, which safely over-approximates the length of the busy chain on s_1. It is easy to see that the computed bound must have finite length as long as the sum of the utilization (e.g., L_{ij}/T_i) of jobs in $\underset{k}{S}^J$ is less than one; therefore, under such assumption we can always compute a valid value for w^1.

Lemma 9. *For $1 \leq k \leq |\delta_{ab}|$, w^k is an upper-bound to $t_k - t_0$, where:*

$$w^k = \min\{\Delta | \Delta = L_{ab} + \bar{W}_{\underset{k}{S}^J}(t_0, t_0 + \Delta) + I^{k-1} - \bar{W}_{(\underset{k-1}{S}^J \bigcap \underset{k}{S}^J)}(t_0, t_0 + w^{k-1})\} \tag{6.4}$$

where $I^{k-1} = \begin{cases} 0 & \text{if } k = 1 \\ w^{k-1} - L_{ab} & \text{otherwise} \end{cases}$

Proof. Note that for $k = 1$, Equation 6.4 reduces to Equation 6.3. Also, the intersection $\left(\underset{k-1}{S}^J \bigcap \underset{k}{S}^J\right)$ represents the set of jobs with priority higher than or equal to P_{abc} that are transmitted on s_k as well as on the previous stage s_{k-1}.

The proof proceeds by induction. Assume that for all stages up to s_{k-1}, the hypothesis holds and furthermore w^{k-1} includes the transmission time of all jobs in $\underset{k-1}{S}^J$ released in $[t_0, t_0 + w^{k-1})$ plus τ_{abc}. By Lemma 8 and definition of $\bar{W}_{\underset{1}{S}^J}(t, t')$, this is true for $k - 1 = 1$ (base case). We need to prove that it holds for stage s_k (induction step).

We show that the maximum sum of transmission lengths of τ_{abc} plus jobs transmitted in the busy chain on $\langle s_1, \ldots, s_k \rangle$ in $[t_0, t_0 + w^k)$ is w^k; we can then use the same three cases as in the proof of Lemma 8 to prove that w^k is a valid upper bound to $t_k - t_0$. No job in $\underset{k}{S}^J$ released at or after $t_0 + w^k$ can be transmitted in $[t_0, t_0 + w^k)$. We will next prove that no job in $\underset{k}{S}^J$ released before t_0 can be executed in the busy chain on stage s_k (P_{abc}-level busy interval $[t_{k-1}, t_k)$). Finally, according to Lemma 7, any datum of a job contributing to the busy chain on stages $s_1, \ldots, s_{k-1}$ can not contribute to the busy chain on s_k. Therefore, we can upper-bound the sum of the transmission lengths of τ_{abc} plus jobs transmitted in the busy chain in $[t_0, t_0 + w^k)$ by taking the workload $\bar{W}_{\underset{k}{S}^J}(t_0, t_0 + w^k)$, summing the maximum length of the busy chain w^{k-1} up to stage s_{k-1} (which includes τ_{abc}), and subtracting the transmission time of jobs that are released in $\underset{k}{S}^J$ within $[t_0, t_0 + w^k)$ but were already counted in w^{k-1}, which is $\bar{W}_{(\underset{k-1}{S}^J \bigcap \underset{k}{S}^J)}(t_0, t_0 + w^{k-1})$; this is equivalent to computing w^k according to Equation 6.4. This concludes the induction step, since we have also shown that w^k indeed includes the transmission time of all jobs in $\underset{k}{S}^J$ released in $[t_0, t_0 + w^k)$ plus τ_{abc}.

We still need to prove that no job in $\underset{k}{S}^J$ released before t_0 can be executed in the P_{abc}-level busy interval $[t_{k-1}, t_k)$. Assume that a job $\tau_{ijp} \in \underset{k}{S}^J$ is released before t_0 on stage s_l, with $l \leq k$. Let t'_l be the slot during which the last datum of τ_{ijp} is transmitted on s_l. Then it must be that $t'_l < t_{l-1} - 1$, otherwise, τ_{ijp} would be transmitting during the P_{abc}-idle slot $[t_{l-1} - 1, t_{l-1})$ given that $t^f_l \geq t_{l-1}$ according to Lemma 6. We then use the same reasoning as in Lemma 7 to show that $t'_k < t_{k-1} - 1$, where t'_k is the slot during which the last datum of τ_{ijp} is transmitted on s_k. □

Figure 6.3 shows the values of $w^1, \ldots, w^4$ computed for the figure's schedule. Let us consider $w^2 = L_{ab} + \bar{W}_{\underset{2}{S}^J}(t_0, t_0 + w^2) + w^1 - L_{ab} - \bar{W}_{(\underset{1}{S}^J \bigcap \underset{2}{S}^J)}(t_0, t_0 + w^1)$. As previously discussed, w^1 includes the transmission times of all jobs on s_1 including the task under analysis τ_{abc} (in solid black). $\bar{W}_{\underset{2}{S}^J}(t_0, t_0 + w^2)$ includes the transmission times of the jobs between the time interval $[4, 10]$. Notice that the job in $[0, 2)$ is not included since it is released at time $0 < t_0 = 1$. We then subtract $\bar{W}_{(\underset{1}{S}^J \bigcap \underset{2}{S}^J)}(t_0, t_0 + w^1)$ which comprises the jobs that were included in w^1 but are also transmitted on s_2. This results in $w^2 = 10$ and $t_0 + w^2 = 11$. Similarly $t_0 + w^3 = 15$, and $t_0 + w^4 = 19$. Note that $w^1, \ldots, w^4$ significantly over-approximate the length of the busy chain; this is because the data transmitted on stage

s_1 in $[3, 8)$ and the datum transmitted on s_2 and s_3 in $[5, 6)$ is counted in the workload despite not being part of the chain. However, as we show in the next section, the over-approximation allows us to greatly reduce the number of different job release time patterns that we need to check to find the worst-case response time of τ_{ab}.

6.3.3 Critical Activation Patterns

Lemma 9 gives us a way to compute an upper-bound $\bar{R}$ on the response time of τ_{abc} based on the pattern of release times of interfering jobs in $\bar{\Theta}$: (1) we first compute the bound $w^{|\delta_{ab}|}$ for the length of the busy chain $t_{|\delta_{ab}|} - t_0$, and (2) we then obtain $\bar{R} = w^{|\delta_{ab}|} - (\hat{t} - t_0)$. Note that $\bar{R}$ represents the response time of τ_{abc} in the stage-normalized schedule $\bar{\Theta}$. We can compute the response time $\hat{R}$ in the original schedule Θ as $\hat{R} = \bar{R} + |\delta_{ab}| - 1$. Finally, since $\hat{R}$ and $\bar{R}$ are measured from the activation time of τ_{abc}, we can obtain the response time from the activation of the root vertex of application A_a as $R = \hat{R} + \phi_{ab}$. Unfortunately, this procedure is not feasible, since we would need to compute $\bar{R}$ for all jobs τ_{abc} of τ_{ab}, and all possible release patterns to obtain the worst-case. To address this, we present the following lemma to show that we only need to consider a finite set of release patterns and jobs to determine the worst-case. The key idea is that we can create a worst-case pattern by releasing each interfering job as soon as possible at or after t_0; this maximizes the workloads computed in Lemmas 8 and 9.

Lemma 10. *Consider applying the following rules to the release pattern of jobs in $\bar{\Theta}$:*

1. *Every job τ_{ijp} that is activated before t_0 and can be released at or after t_0 is released at t_0.*
2. *Every job τ_{ijp} that is activated after t_0 is released immediately at its activation time.*
3. *The activation time of every application A_i is moved earlier in time until one job of A_i is released at time t_0 after suffering maximum jitter.*

Then the response time bound $\bar{R}$ computed for the modified release pattern will be no less than the response time bound computed for the original pattern.

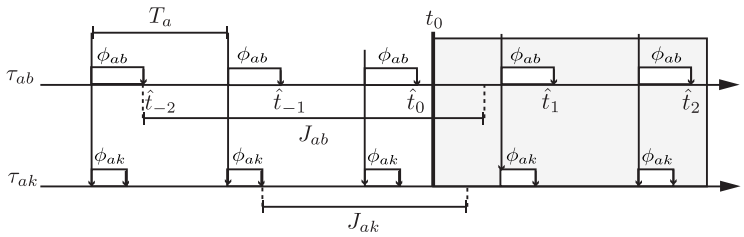
Proof. The first rule is concerned with any job τ_{ijp} that is activated before t_0, but can be released at or after t_0. If job τ_{ijp} is released before t_0, it will either not interfere with the job under analysis τ_{abc} or will cause less interference compared to being released at t_0. Also if τ_{ijp} is released after t_0, it might not interfere with τ_{abc}. Hence, releasing τ_{ijp} at t_0 leads to maximum interference.

The second rule states that any job τ_{ijp} activated after t_0 should be immediately released. Since τ_{ijp} is activated after t_0, then it might interfere with the τ_{abc}. The earlier τ_{ijp} is released, the higher the chance it will interfere with τ_{abc}. Hence, the immediate release of τ_{ijp} causes the most interference.

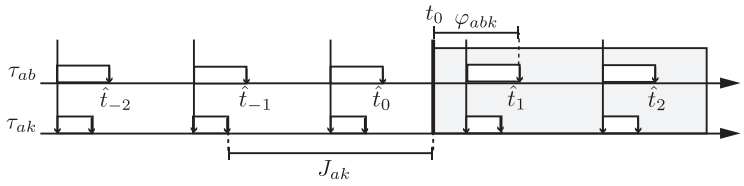
The third rule states that the contribution of application A_i to the latency of τ_{abc} is worst when the release of one of its jobs coincides with t_0 after suffering maximum jitter.

Moving the activations of the jobs of A_i earlier in time until one job has maximum jitter while still being released at t_0, contributes most to the latency of τ_{abc}. The reason is that by moving activations earlier in time, jobs of A_i that were activated after t_0 have a higher chance of causing more interference to τ_{abc}.

Consider any interval $[t_0, t_0+\Delta)$ as in Lemmas 8 and 9. If a job was released in $[t_0, t_0+\Delta)$ in the original pattern, then it will still be released in $[t_0, t_0+\Delta)$ in the modified pattern. This is because Rules 1 and 2 force any job that could be released within $[t_0, t_0+\Delta)$ to indeed be released within the interval. Furthermore, Rule 3 can not cause any job released at or after t_0 to be released before t_0. Therefore, the value of $\bar{W}_{\substack{S^J \\ k}}(t_0, t_0+\Delta)$ for any set $\underset{k}{S^J}$ computed in the modified pattern will be greater than or equal to the value computed in the original pattern. It is then easy to see that for all stages s_k, the computed value of w^k can not decrease after applying these three rules. Since $\bar{R} = w^{|\delta_{ab}|} - (\hat{t} - t_0)$, to conclude the proof it suffices to note that the activation time $\hat{t}$ of τ_{abc} in the modified pattern can not be larger than in the original pattern. This is because Rule 3 can move the activation time of τ_{abc} to occur earlier but not later in time. □



(a) Original release pattern



(b) Modified release pattern

Figure 6.4: Release patterns of jobs of application A_a

Figure 6.4a shows an example timeline including t_0, the beginning of a busy chain. It also shows activation times (down arrows) $\hat{t}_{-2}, \ldots, \hat{t}_2$ for several jobs of τ_{ab}, as well as jobs of another higher priority task τ_{ak} of application A_a. Note that the first job of τ_{ak} cannot be released at or after t_0. Rule 1 applies to the first three jobs of τ_{ab} and the second and third job of τ_{ak}; they are activated before t_0 and have enough jitter to be released at or after t_0. These jobs are released at t_0. Rule 2 applies to the last two jobs of both tasks; they are activated after t_0. These jobs are released immediately at their activation times. Finally, Figure 6.4b shows the modified pattern after applying Rule 3; the activation time of A_a is moved earlier in time until the second job of τ_{ak} is released at t_0 after suffering maximum jitter.

We call the modified pattern, obtained from Lemma 10, a *critical pattern*. Let $\tau_{abc'}$, with $c' \leq c$, be the first job of τ_{ab} released at or after t_0. The number of critical patterns for τ_{abc}

where $\tau_{abc'}$ is the first such job is then $(|\mathbb{S}_a^D(\tau_{ab})|+1)\cdot\prod_{\forall i\neq a}|\mathbb{S}_i^D(\tau_{ab})|$. The application under analysis A_a is activated at a time such that either a job of any interfering task in $\mathbb{S}_a^D(\tau_{ab})$ or $\tau_{abc'}$ is released at t_0 after suffering maximum jitter, providing $|\mathbb{S}_a^D(\tau_{ab})|+1$ combinations. Every other application A_i is activated at a time such that a job of any interfering task in $\mathbb{S}_i^D(\tau_{ab})$ is released at t_0 after suffering maximum jitter, providing $|\mathbb{S}_i^D(\tau_{ab})|$ combinations. Each of the possible critical activation patterns is characterized by a tuple v of indices, one per application. Each index $v(i)$ identifies the task of application A_i that coincides with the beginning of the busy chain. Lemma 10 represents the equivalent of Theorems 1 and 2 in Palencia and Gonzalez [108], which prove that a critical instant for the task under analysis can be built by releasing one task of each application at the critical instant after suffering maximum jitter. Intuitively, the beginning of the busy chain t_0 represents the equivalent of the critical instant, except that it accounts for transmission on multiple stages and must thus include patterns that would not be valid in single-resource systems. For example, note in Figure 6.2 that the schedule is not P_{abc}-level idle at time t_0 on stages s_2 and s_3.

Given a critical pattern, the index of the first job $\tau_{abc'}$ released at or after t_0 is relevant because it determines the activation time $\hat{t}$ of τ_{abc}. However, the time difference $\hat{t}-t_0$ used to compute $\bar{R}$ depends only on the difference $c-c'$. In other words, the same critical release patterns apply to all jobs τ_{abc} of τ_{ab} as long as we vary the number of jobs of τ_{abc} that are released in $[t_0, t_0+w^{|\delta_{ab}|})$.

6.3.4 Indirect Interference Jitter

Given a task under analysis τ_{ab}, the phase φ_{ijk} between any task τ_{ij} and the beginning of the busy chain of a critical activation pattern created by task τ_{ik} is given by:

$$\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik}^R + J_{ik}^I(\tau_{ab}) - \phi_{ij}) \bmod T_i$$

where ϕ_{ij} is the reduced offset of task τ_{ij} to the period 0 to T_i and $J_{ik}^I(\tau_{ab})$ is the interference jitter suffered by τ_{ik} and is given by:

$$J_{ik}^I(\tau_{ab}) = R_{ik}(\tau_{ab}) - L_{ik} - J_{ik}^R - \phi_{ik}$$

The interference jitter $J_{ik}^I(\tau_{ab})$ represents the interference suffered by task τ_{ik} only due to tasks in the indirect interference set of task τ_{ab}, $\mathbb{S}_{ik}^I(\tau_{ab})$. We use the notation $R_{ik}(\tau_{ab})$ to denote the response time of task τ_{ik} only due to interference from higher priority tasks in the set $\mathbb{S}_{ik}^I(\tau_{ab})$. As an example, Figure 6.4b shows the phase $\varphi_{abv(a)} = \varphi_{abk}$ between τ_{ab} and the beginning of the busy chain created by τ_{ak}, assuming that $J_{ak} = J_{ak}^R + J_{ak}^I(\tau_{ab})$. Note that for the first job of τ_{ab} activated after t_0, φ_{abk} is exactly equal to the length of the interval $\hat{t}-t_0$ used in Section 6.3.3.

6.3.5 Exponential Analysis

Now we are ready to present the formulation for the exponential response time analysis. Let us use the term critical instant for the beginning of the busy chain, t_0, of a critical

activation pattern. We introduce a numbering scheme to track the number of job instances that we need to consider in a busy chain. We use $p = -1$ to refer to the instance activated in the interval $[t_0 - 2 * T_a, t_0 - T_a)$, $p = 0$ in the interval $[t_0 - T_a, t_0)$, $p = 1$ in the interval $[t_0, t_0 + T_a)$, and so on. Note that the numbering scheme corresponds with the indices of $\hat{t}_p$ in Figure 6.4; furthermore, we have $\hat{t}_p - t_0 = \varphi_{abv(a)} + (p-1) * T_a$. The first job instance that we consider is one with the least index that has enough jitter to be part of the busy chain. Hence, the first job instance $p^v_{0,ab} = -\lfloor \frac{J^R_{ab} + \varphi_{abv(a)}}{T_a} \rfloor + 1$.

Lemma 11. *The worst-case contribution of an application A_i to the busy chain of τ_{ab} on stage s_l when the activation of task τ_{ik} coincides with the critical instant is given by:*

$$W_{ik}(\tau_{ab}, l, t) = \sum_{\forall j \in \mathbb{S}^D_{l\,i}(\tau_{ab})} \left(\left\lfloor \frac{J^R_{ij} + J^I_{ij}(\tau_{ab}) + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil \right) * L_{ij}$$

Proof. By definition, a workload $\bar{W}_{\underset{l}{S^J}}(t, t')$ includes the transmission times of all jobs in the set $\underset{l}{S^J}$ that are released in the interval $[t, t')$. The set $\underset{l}{S^J}$ includes all jobs of higher priority tasks of application A_i, $\mathbb{S}^D_{l\,i}(\tau_{ab})$, that contribute to the workload on stage s_l. Consider an arbitrary higher priority task of application A_i, τ_{ij}. According to the first rule of Lemma 10, all job instances of τ_{ij} that are activated before the critical instant and have enough jitter that allows them to contribute to the workload, are released at the critical instant. The first term of the summation accounts for these job instances, this is similar to the derivation in [108]. The second term simply applies the second rule of Lemma 10. The job instances belonging to the second rule are a series of periodic activations starting φ_{ijk} time units after the critical instant. Note, that by definition of a workload, activations released at time t' are not part of the workload. □

We further explain Lemma 11 using Figure 6.4b applied to task τ_{ak}, where $v(a) = k$. The first term $\lfloor \frac{J^R_{ak} + J^I_{ak}(\tau_{ab}) + \varphi_{akk}}{T_a} \rfloor$ accounts for the second and third jobs that are activated before the critical instant. The second term $\lceil \frac{t - \varphi_{akk}}{T_a} \rceil$ accounts for the two jobs that are activated after the critical instant. Note that according to the definition of the workload, if the numerator $t - \varphi_{akk}$ is such that it exactly matches the period, then we will not consider the last job as part of the worst-case contribution $W_{ak}(\tau_{ab}, l, t)$.

For clarity of presentation, we define $W'_{ik}(\tau_{ab}, l, t)$ as the worst-case contribution of an application A_i on the response time of task τ_{ab} on stage s_{l-1} solely due to tasks that are common on stages s_{l-1} and s_l, i.e., in the set $\underset{l-1}{\mathbb{S}}{}^D_i(\tau_{ab}) \cap \mathbb{S}^D_{l\,i}(\tau_{ab})$. We also use $\underset{l}{p}{}^v_{B,ab}$ to denote the largest-numbered job instance in the interval w^l.

Lemma 12. *For each activation pattern v, the worst-case length of the busy chain for each job p of task τ_{ab} up to stage s_l is determined by:*

$$\underset{l}{w}{}^v_{ab}(p) = \underset{l}{I}{}^v_{ab}(p) + (p - p^v_{0,ab} + 1) * L_{ab}$$

$$\underset{l}{I}{}^v_{ab}(p) = \underset{l-1}{I}{}^v_{ab}(p') + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, \underset{l}{w}{}^v_{ab}(p)) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}{}^v_{ab}(p'))$$

$$\text{where } p' = \begin{cases} p & \text{if } p \leq {}_{l-1}p^{v}_{B,ab} \\ {}_{l-1}p^{v}_{B,ab} & \text{otherwise} \end{cases}$$

Proof. This proof directly descends from Lemma 9. In the above equation, if we remove the restriction to a specific job instance p, then we compute ${}_{l}w^{v}_{ab}$ which is the length of the busy chain up to stage s_l where the largest-numbered job instance is ${}_{l}p^{v}_{B,ab}$. We also replace p by $\lceil \frac{{}_{l}w^{v}_{ab} - \varphi_{abv(a)}}{T_a} \rceil$ to find the largest-numbered job instance in ${}_{l}w^{v}_{ab}$. The equation then becomes:

$$\begin{aligned} {}_{l}w^{v}_{ab} &= {}_{l}I^{v}_{ab} + (\left\lceil \frac{{}_{l}w^{v}_{ab} - \varphi_{abv(a)}}{T_a} \right\rceil - p^{v}_{0,ab} + 1) * L_{ab} \\ {}_{l}I^{v}_{ab} &= {}_{l-1}I^{v}_{ab} + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, {}_{l}w^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, {}_{l-1}w^{v}_{ab}) \end{aligned}$$

Thus, computing w^l as introduced in Lemma 9. The only difference is that ${}_{l}w^{v}_{ab}$ separates interference from higher priority tasks and jobs from the same task into different terms.

For the first stage ($k = 1$), the terms ${}_{l-1}I^{v}_{ab}$ and $W'_{ik}(\tau_{ab}, l, t)$ are zeros thus yielding:

$${}_{l}w^{v}_{ab} = \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, {}_{l}w^{v}_{ab}) + (\left\lceil \frac{{}_{l}w^{v}_{ab} - \varphi_{abv(a)}}{T_a} \right\rceil - p^{v}_{0,ab} + 1) * L_{ab} \tag{6.5}$$

This is equivalent to Equation 6.3. The first term is the interference from higher priority tasks (excluding jobs of τ_{ab}). The second term represents all jobs of τ_{ab} including the job under analysis.

In order to compute ${}_{l}w^{v}_{ab}(p)$ for a specific job instance then we only consider interference from higher priority tasks while excluding any job instances that are activated after p. Similar to Lemma 9, and using Lemma 11, to compute ${}_{l}w^{v}_{ab}(p)$, we add interference from higher priority tasks ${}_{l}I^{v}_{ab}(p)$ to the jobs of τ_{ab} up till p. To compute ${}_{l}I^{v}_{ab}(p)$, we use the interference on the previous stage ${}_{l-1}I^{v}_{ab}(p')$ plus any interference on stage s_l while subtracting interferences that are common on stages s_{l-1} and s_l.

Note that we use p' instead of p in ${}_{l-1}w^{v}_{ab}(p')$ and ${}_{l-1}I^{v}_{ab}(p')$. This is due to the fact that a certain p might only exist on stage s_l but not s_{l-1}. In such case, the worst-case length of the busy chain that can be considered from the previous stage is only up till the maximum p existing on it, i.e., ${}_{l-1}p^{v}_{B,ab}$.

□

Theorem 6. *The worst-case response time of task τ_{ab} is obtained by:*

$$R_{ab} = \max_{\forall v} \left(\max_{p=p^{v}_{0,ab} \cdots {}_{|\delta_{ab}|}p^{v}_{B,ab}} ({}_{|\delta_{ab}|}R^{v}_{ab}(p)) \right) + |\delta_{ab}| - 1$$

where $\underset{|\delta_{ab}|}{R}{}^{v}_{ab}(p) = \underset{|\delta_{ab}|}{w}{}^{v}_{ab}(p) - \varphi_{abv(a)} - (p-1) * T_a + \phi_{ab}$.

Proof. Using Lemma 12, we can find the worst-case length of the busy chain for each job p on any stage along the path δ_{ab} of τ_{ab} for a particular activation pattern v. We also showed that the response time for a job of τ_{ab} is $\bar{R} = w^{|\delta_{ab}|} - (\hat{t} - t_0)$, i.e, $\bar{R} = \underset{|\delta_{ab}|}{w}{}^{v}_{ab}(p) - \varphi_{abv(a)} - (p-1) * T_a$ for job p. To obtain the response time $\underset{|\delta_{ab}|}{R}{}^{v}_{ab}(p)$ measured from the activation of application A_a, rather than the activation of τ_{ab}, we also need to add the offset ϕ_{ab}. Next, we compute the maximum worst-case response time across all job instances, which gives us the worst-case response time of τ_{ab} for activation pattern v. From Lemma 10, we consider the maximum worst-case response time across all activation patterns to obtain the WCRT of task τ_{ab}, R_{ab}. □

6.3.6 Polynomial Analysis

The difficulty with the exponential analysis is that it is exponential in the number of critical activation patterns. Hence, we derive an upper bound on the interference caused by an application A_i as the maximum interference caused by considering each task in A_i to coincide with the critical instant. We use the notation:

$$W_i^*(\tau_{ab}, l, t) = \max_{\forall k \in \underset{l}{\mathbb{S}}{}_i^D(\tau_{ab})} W_{ik}(\tau_{ab}, l, t)$$

to compute this upper bound on a specific stage s_l. We use this approximation only for higher priority applications and not application A_a to which the task under analysis τ_{ab} belongs. The number of critical activation patterns that we must consider is thus reduced to $|\mathbb{S}_i^D(\tau_{ab})| + 1$. In what follows, we derive a polynomial WCRT analysis for our pipelined communication model. For clarity of presentation, we use:

$$W_i''^*(\tau_{ab}, l, t) = \max_{\forall k \in \underset{l}{\mathbb{S}}{}_i^D(\tau_{ab})} W''_{ik}(\tau_{ab}, l, t)$$

where $W''_{ik}(\tau_{ab}, l, t)$ is the worst-case contribution of an application A_i to the busy chain of task τ_{ab} on stage s_{l-1} solely due to tasks that are on stage s_{l-1} but not s_l, i.e., in the set $\underset{l-1}{\mathbb{S}}{}_i^D(\tau_{ab}) \backslash \underset{l}{\mathbb{S}}{}_i^D(\tau_{ab})$.

Lemma 13. *For a critical instant created with task τ_{ac}, the worst-case length of the busy chain for each job p of task τ_{ab} up to stage s_l is determined by:*

$$\begin{aligned}\underset{l}{w}{}_{abc}(p) = &\sum_{s=2\ldots l} \sum_{\forall i \neq a} W_i''^*(\tau_{ab}, s, \underset{s-1}{w}{}_{abc}(p')) + \sum_{\forall i} W_i^*(\tau_{ab}, l, \underset{l}{w}{}_{abc}(p)) \\ &+ \sum_{s=2\ldots l} W''_{ac}(\tau_{ab}, s, \underset{s-1}{w}{}_{abc}(p')) + W_{ac}(\tau_{ab}, l, \underset{l}{w}{}_{abc}(p)) + (p - p^v_{0,ab} + 1) * L_{ab}\end{aligned}$$

Proof. We first transform the interference from Lemma 12 into a more convenient form for the discussion. Consider using Lemma 12 to compute a busy chain (through dropping p as shown earlier). The interference from higher priority tasks is accounted for by:

$$I_{l}^{v}{}_{ab} = \underset{l-1}{I}^{v}{}_{ab} + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, \underset{l}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab})$$

The term $\underset{l-1}{I}^{v}{}_{ab}$ can in turn be expanded into:

$$\underset{l-1}{I}^{v}{}_{ab} = \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l-1, \underset{l-1}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l-1, \underset{l-2}{w}^{v}_{ab})$$

We expand all terms until reaching the first stage s_1. Hence, we get:

$$\begin{aligned}
I_{l}^{v}{}_{ab} = & \sum_{\forall i} W_{iv(i)}(\tau_{ab}, 1, \underset{1}{w}^{v}_{ab}) \\
& + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, 2, \underset{2}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, 2, \underset{1}{w}^{v}_{ab}) + \ldots \\
& + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l-1, \underset{l-1}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l-1, \underset{l-2}{w}^{v}_{ab}) \\
& + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, \underset{l}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab})
\end{aligned}$$

Recall that $\sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab})$ is the interference on stage s_{l-1} from tasks in the set $\underset{l-1}{\mathbb{S}}_{i}^{D}(\tau_{ab}) \cap \mathbb{S}_{l}^{D}(\tau_{ab})$. Hence the terms, $\sum_{\forall i} W_{iv(i)}(\tau_{ab}, l-1, \underset{l-1}{w}^{v}_{ab}) - \sum_{\forall i} W'_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab})$ can be rewritten as $\sum_{\forall i} W''_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab})$ which is the interference due to tasks in the set $\underset{l-1}{\mathbb{S}}_{i}^{D}(\tau_{ab}) \backslash \mathbb{S}_{l}^{D}(\tau_{ab})$. This is intuitive since the first term is interference from the set $\underset{l-1}{\mathbb{S}}_{i}^{D}(\tau_{ab})$ and the second term is the interference from the set $\underset{l-1}{\mathbb{S}}_{i}^{D}(\tau_{ab}) \cap \mathbb{S}_{l}^{D}(\tau_{ab})$. Therefore, we can write the interference in Lemma 12 in the form:

$$I_{l}^{v}{}_{ab} = \sum_{\forall i} W''_{iv(i)}(\tau_{ab}, 2, \underset{1}{w}^{v}_{ab}) + \ldots + \sum_{\forall i} W''_{iv(i)}(\tau_{ab}, l, \underset{l-1}{w}^{v}_{ab}) + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, \underset{l}{w}^{v}_{ab})$$

Or in a more compact form:

$$I_{l}^{v}{}_{ab} = \sum_{s=2\ldots l} \sum_{\forall i} W''_{iv(i)}(\tau_{ab}, s, \underset{s-1}{w}^{v}_{ab}) + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, l, \underset{l}{w}^{v}_{ab})$$

Next, we consider the first stage on path δ_{ab} of τ_{ab}. The interference on stage s_1 is equal to $\sum_{\forall i} W_{iv(i)}(\tau_{ab}, 1, \underset{1}{w}^{v}_{ab})$. Using the approximation introduced earlier, the interference will be equal to $\sum_{\forall i} W_{i}^{*}(\tau_{ab}, 1, \underset{1}{w}^{v}_{ab})$. Since the approximation considers the maximum interference that can be achieved by A_i through considering each task to coincide with the critical instant, then doing the summation over all higher priority tasks yields an interference that is greater than or equal to considering any individual activation pattern,

i.e., $W_i^*(\tau_{ab}, 1, \underset{1}{w}^v_{ab})$ is an upper bound of $\sum_{\forall i} W_{iv(i)}(\tau_{ab}, 1, \underset{1}{w}^v_{ab})$. This is similar to the derivation in [146].

Let us consider the second stage s_2. The interference on this stage is equal to:

$$\underset{2}{I}^v_{ab} = \sum_{\forall i} W''_{iv(i)}(\tau_{ab}, 2, \underset{1}{w}^v_{ab}) + \sum_{\forall i} W_{iv(i)}(\tau_{ab}, 2, \underset{2}{w}^v_{ab})$$

This is the sum of interferences from tasks that are only on stage 1 and tasks that exist on stage 2. Now consider the approximation:

$$\sum_{\forall i} W''^*_i(\tau_{ab}, 2, \underset{1}{w}^v_{ab}) + \sum_{\forall i} W^*_i(\tau_{ab}, 2, \underset{2}{w}^v_{ab})$$

Since $W_i^*(\tau_{ab}, 2, \underset{2}{w}_{abc}) \geq W_{iv(i)}(\tau_{ab}, 2, \underset{2}{w}^v_{ab})$ and the same holds for the interference that occurs only on stage 1, $W''^*_i(\tau_{ab}, 2, \underset{1}{w}^v_{ab}) \geq W''_{iv(i)}(\tau_{ab}, 2, \underset{1}{w}^v_{ab})$. Hence, the approximation on stage s_2 yields an upper bound to the interference on that stage. Similarly this can be extended to all stages until stage $|\delta_{ab}|$.

Lastly, in this Lemma, we only apply the approximation to higher priority applications, i.e., $\forall i \neq a$. Hence, we compute an upper bound for the interference from all higher priority applications. To find the polynomial worst-case response time for task τ_{ab}, we need to consider all possible critical instants from application A_a. □

Theorem 7. *The worst-case response time of task τ_{ab} is obtained by:*

$$R_{ab} = \max_{\forall c \in \mathbb{S}^D_i(\tau_{ab}) \cup b} \left(\max_{p=p^v_{0,ab} \cdots \underset{|\delta_{ab}|}{p}} (\underset{|\delta_{ab}|}{R}{}^v_{B,ab}(p)) \right) + |\delta_{ab}| - 1$$

where $\underset{|\delta_{ab}|}{R}{}^v_{abc}(p) = \underset{|\delta_{ab}|}{w}{}^v_{abc}(p) - \varphi_{abv(a)} - (p-1) * T_a + \phi_{ab}$.

Proof. The proof is similar to that of Theorem 6. Lemma 13 computes the worst-case length of the busy chain for each job p on any stage along the path δ_{ab} of τ_{ab} when a task τ_{ac} creates the critical instant. First, we obtain the worst-case response time of a job p measured from the activation of application A_a. Next, we compute the maximum worst-case response time across all job instances. Then, to find the worst-case response time, we consider the creation of the critical instant by all higher priority tasks of application A_a as well as by task τ_{ab}. □

6.4 Offset-based Flow-Level Analysis

Similar to OSLA, we also extend FLA [128, 126, 125] with offsets to indicate precedence relationships between computation and communication tasks.

6.4.1 Exponential Analysis

Recall that FLA does not take the pipelining of communication resources into consideration. Hence, the path δ_{ab} of the task under analysis τ_{ab} is viewed as a single resource. Similar to OSLA, we apply Lemma 10 to obtain a set of critical activation patterns. The number of critical patterns that we need to consider for an exponential analysis is also $(|\mathbb{S}_a^D(\tau_{ab})| + 1) \cdot \prod_{\forall i \neq a} |\mathbb{S}_i^D(\tau_{ab})|$. The indirect interference jitter J_{ik}^I suffered by a task τ_{ik} is defined in Section 6.3.4 with a slight modification. Since the whole path of the task τ_{ik} is viewed as a single resource, then the worst-case transmission latency on a stage, L_{ik}, is replaced by the worst-case transmission latency on the whole path, C_{ik}. The equation thus becomes:

$$J_{ik}^I(\tau_{ab}) = R_{ik}(\tau_{ab}) - C_{ik} - J_{ik}^R - \phi_{ik} \tag{6.6}$$

We can use the same reasoning from Lemma 11 to compute the worst-case contribution of an application A_i to the response time of task τ_{ab} when the critical instant coincides with the activation of task τ_{ik}. Again, since OFLA views the path δ_{ab} of task τ_{ab} as a single resource, we replace L_{ik} with C_{ik}. The equation then becomes:

$$W_{ik}(\tau_{ab}, t) = \sum_{\forall j \in \mathbb{S}_i^D(\tau_{ab})} \left(\left\lfloor \frac{J_{ij}^R + J_{ij}^I(\tau_{ab}) + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil \right) * C_{ij}$$

For each activation pattern v, the worst-case length of the busy interval for task τ_{ab} is determined by:

$$w_{ab}^v = \sum_{\forall i} W_{iv(i)}(\tau_{ab}, w_{ab}^v) + \left(\left\lceil \frac{w_{ab}^v - \varphi_{abv(a)}}{T_a} \right\rceil - p_{0,ab}^v + 1 \right) * C_{ab}$$

where $p_{0,ab}^v$ is the lowest-numbered job instance as defined in Section 6.3.5. Notice that this equation is equivalent to the worst-case length of the busy chain on the first stage as defined by OSLA in Equation 6.5. The first term accounts for interference from higher priority tasks, and the second term accounts for jobs of τ_{ab} including the job under analysis.

We now find the worst-case completion time of each job instance p in the busy interval of task τ_{ab}. To do this, we consider interference only up till job p and discard any job instances activated after p. The worst-case completion time $w_{ab}^v(p)$ of job p is given by:

$$w_{ab}^v(p) = \sum_{\forall i} W_{iv(i)}(\tau_{ab}, w_{ab}^v(p)) + (p - p_{0,ab}^v + 1) * C_{ab} \tag{6.7}$$

Theorem 8. *The worst-case response time of task τ_{ab} is given by:*

$$R_{ab} = \max_{\forall v} \left(\max_{p = p_{0,ab}^v \cdots p_{B,ab}^v} R_{ab}^v(p) \right)$$

where $R_{ab}^v(p) = w_{ab}^v(p) - \varphi_{abv(a)} - (p-1) * T_a + \phi_{ab}$

Proof. Using Equation 6.7, we can obtain the worst-case completion time of each job instance p in the busy interval of task τ_{ab}, for a given activation pattern v. We can now obtain the worst-case response time of job instance p, $R^v_{ab}(p)$. To get the worst-case response time of job p, we subtract from the worst-case completion time, the activation time of job p. The activation of job p occurs at time $\varphi_{abv(a)} + (p-1) * T_a$ measured from the critical instant. To obtain the worst-case response time measured from the activation of application A_a, we add the offset ϕ_{ab} of task τ_{ab}.

To obtain the worst-case response time of task τ_{ab}, first, we find the job instance with the highest worst-case response time for each activation pattern v. Then, we find highest worst-case response time amongst all activation patterns. □

6.4.2 Direct and Indirect Interferences

Equation 6.6 presents the worst-case contribution of application A_i on the response time of task τ_{ab}. The jitter, suffered by the higher priority task τ_{ij} causing interference, includes both release jitter and indirect interference jitter. The indirect interference jitter results from tasks in the indirect interference set of task τ_{ab} and through the intermediate task τ_{ij}. However, it might be the case that for a particular activation pattern, the offset between the indirectly interfering task and the critical instant is such that it does not contribute to the busy interval of τ_{ab}. In that case, a tighter worst-case response time of task τ_{ab}, for that activation pattern, can be achieved by not considering that indirectly interfering task.

Consider the task under analysis τ_{ab}. Assume the directly interfering task τ_{ij}, and an indirectly interfering task τ_{kl} which indirectly interferes with τ_{ab} through τ_{ij}. When computing the worst-case contribution of application A_i to the response time of task τ_{ab} for an activation pattern v, the indirect interference jitter of task τ_{ij}, $J^I_{ij}(\tau_{ab})$, is used. This indirect interference jitter is computed based on the worst-case response time of task τ_{ij} due to tasks in the indirect interference set of τ_{ab} including task τ_{kl}. Hence, the worst-case effect of task τ_{kl}, on τ_{ab}, is increasing the jitter of task τ_{ij} by an amount of time equivalent to the contribution of τ_{kl} to the worst-case response time of τ_{ij}. Note that this indirect interference jitter is computed through the worst-case response time of task τ_{ij}, which, according to Theorem 8, is the worst-case response time across all critical activation patterns for task τ_{ij}. This means that the indirect interference jitter of task τ_{ij} is computed based on the worst-case for task τ_{ij} and irrespective of the activation pattern v for which τ_{ab} is being analyzed. This leads to the conclusion, that for the activation pattern v, task τ_{kl} (which has already been accounted for in the indirect interference jitter of τ_{ij}) might have an offset from the critical instant such that it does not contribute to the busy interval of τ_{ab}.

Theorem 9. *The worst-case response time of task τ_{ab} is given by:*

$$R_{ab} = \max_{\forall v}(\min_{\forall \alpha}(\max_{p=p^v_{0,ab}\cdots p^v_{B,ab}} R^{v,\alpha}_{ab}(p)))$$

where α is a combination resulting from considering each task causing indirect interference as either in the direct or indirect interference set of τ_{ab}.

Proof. We can find out whether task τ_{kl} contributes to the busy interval of τ_{ab} by moving it to the direct interference set of τ_{ab}. In that case, it will not cause indirect interference jitter through τ_{ij}, and if it does not contribute to the busy interval of τ_{ab}, it will not affect the worst-case response time analysis. It is safe to consider τ_{kl} as a directly interfering task, because if τ_{kl} contributes to the busy interval of τ_{ab}, then its contribution will only increase by considering it as a directly interfering task. Therefore, in both cases (considering task τ_{kl} as either indirectly or directly interfering with τ_{ab}), the computed worst-case response time of τ_{ab} will be a correct upper bound to the response time except that one bound will be tighter than the other. A tighter worst-case response time can, thus, be computed by considering each task in the indirect interference set of task τ_{ab} as either indirectly or directly interfering with τ_{ab}. □

The implication of considering each task causing indirect interference as either in the direct or indirect interference sets results in an exponential number of combinations. If $|\mathbb{S}^I(\tau_{ab})|$ is the number of indirectly interfering tasks with task τ_{ab}, then the number of combinations that we need to check is equal to $2^{|\mathbb{S}^I(\tau_{ab})|}$ for each activation pattern v.

6.4.3 Polynomial Analysis

The worst-case response time computation using Theorem 9 is exponential in the number of activation patterns and the number of tasks indirectly interfering with τ_{ab}. The computational complexity is, thus, $(|\mathbb{S}^D_a(\tau_{ab})|+1)\cdot\prod_{\forall i\neq a}|\mathbb{S}^D_i(\tau_{ab})|\cdot 2^{|\mathbb{S}^I(\tau_{ab})|}$. For practical and tractability reasons, we discover and present an approximated polynomial analysis.

Approximation 1

This approximation is similar to the one presented in Section 6.3.6. We derive an upper bound on the interference caused by an application A_i to the busy interval of task τ_{ab} by computing the maximum interference caused by considering each task in A_i to coincide with the critical instant. We apply this to all higher priority applications interfering with τ_{ab} but not to application A_a to which τ_{ab} belongs. The number of activation patterns that we need to consider becomes $|\mathbb{S}^D_a(\tau_{ab})|+1$ and the upper bound on the interference is computed as:

$$W^*_i(\tau_{ab},t) = \max_{\forall k\in\mathbb{S}^D_i(\tau_{ab})} W_{ik}(\tau_{ab},t)$$

The completion time $w_{abc}(p)$ of job p of task τ_{ab} when the critical instant coincides with the activation of task τ_{ac} is given by:

$$w_{abc}(p) = \sum_{\forall i\neq a} W^*_i(\tau_{ab},w_{abc}(p)) + W_{ac}(\tau_{ab},w_{abc}(p)) + (p-p_{0,abc}+1)*C_{ab} \qquad (6.8)$$

This is similar to the computation in the exponential version of the analysis (Equation 6.7). The first term represents the upper bound on the interference on each application except A_a. The second term represents the interference suffered from tasks of application A_a when

task τ_{ac} coincides with the critical instant. The last term accounts for jobs of task τ_{ab} up to and including the job under analysis p.

The worst-case response time of job p of task τ_{ab} when the critical instant coincides with the activation of task τ_{ac} is given by:

$$R_{abc}(p) = w_{abc}(p) - \varphi_{abc} - (p-1) * T_a + \phi_{ab} \tag{6.9}$$

where $w_{abc}(p)$ is computed using Equation 6.8.

Approximation 2

The second approximation handles the combinations resulting from considering the tasks in the indirect interference set of τ_{ab}, $\mathbb{S}^I(\tau_{ab})$, as either indirectly or directly interfering with τ_{ab}. Instead of checking all $2^{|\mathbb{S}^I(\tau_{ab})|}$ combinations, we only check two:

1. α_1: All tasks in the indirect interference set of τ_{ab}, indirectly interfere with τ_{ab}.
2. α_2: All tasks in the indirect interference set of τ_{ab}, directly interfere with τ_{ab}.

This results in a polynomial analysis with $(|S_a^D(\tau_{ab})| + 1) \cdot 2$ computations.

Theorem 10. *The worst-case response time of task τ_{ab} is given by:*

$$R_{ab} = \max_{\forall c \in \mathbb{S}_a^D(\tau_{ab}) \cup b} \left(\min_{\alpha_1, \alpha_2} \left(\max_{p = p_{0,ab}^v \ldots p_{B,ab}^v} R_{abc}^{\alpha}(p) \right) \right)$$

where $R_{abc}^{\alpha}(p)$ is computed using Equation 6.9.

Proof. Using Equation 6.9, we can obtain the worst-case response time of each job instance p in the busy interval of task τ_{ab}, measured from the activation of application A_a. We consider a number of critical activation patterns created by each of the higher priority tasks in application A_a (including τ_{ab}) coinciding with the critical instant. For each of these critical patters, we take the minimum worst-case response time of the two combinations that we check, α_1 and α_2 The $WCRT$ of task τ_{ab} is then obtained by taking the maximum across all critical activation patterns. □

6.5 Experimental Evaluation

For the experimental evaluation of the proposed WCRT analysis, we present an application of a priority-aware NoC as presented by Shi and Burns [126, 128, 125]. This NoC supports wormhole switching with flit-level preemption. Details of the NoC architecture are available in [126]. In particular, we propose two instances of NoCs with sizes 4×4 and 8×8 for the deployment platform. Each node in the NoC is a processing resource in the resource model, and each link between two nodes represents a stage in the pipelined communication resources. A computation task executes on the node, and the communication task transmits

data across multiple links to its destination node. These links of the NoC correspond to the different stages of the communication resource. We experiment with exponential and polynomial versions of OSLA, OFLA, and Palencia and Gonzalez's analyses (PAL) [108]. We also add the optimization by Turja and Maki [89] to all polynomial versions.

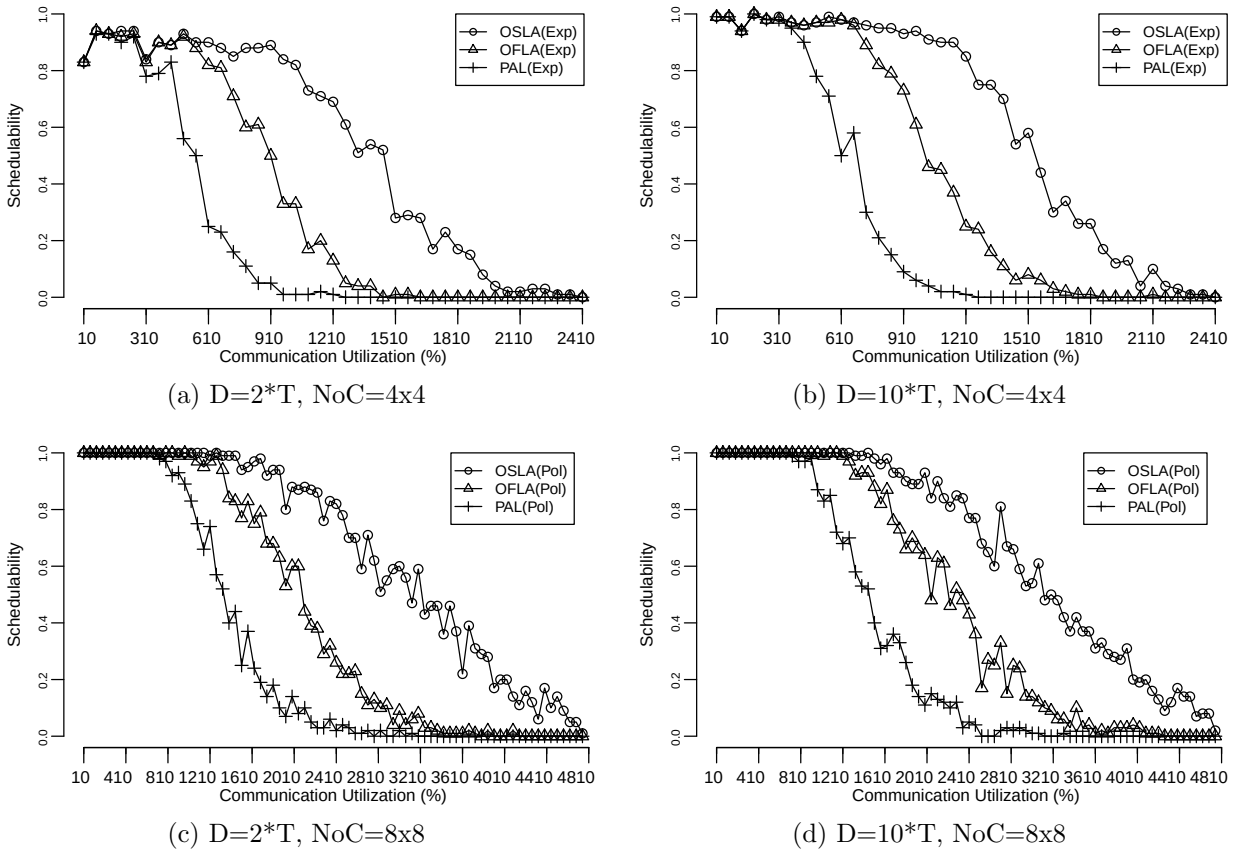
We setup the experiments with the following parameters:

1. DAGs are randomly generated to represent arbitrary applications.
2. The number of applications per test case is 10.
3. The number of tasks per application is varied in the range (3,10).
4. The application period is randomly chosen in the range $(1000, 1000000)$.
5. The deadline is chosen as a coefficient (e.g. 10x) of the period.
6. The applications are prioritized using rate-monotonic priority assignment.
7. An arbitrary priority assignment scheme is chosen for priority of tasks within each application.
8. The application release jitter is set to zero.
9. Task offsets and jitters are calculated based on the methods presented in Section 6.2.
10. The communication utilization is equally divided between applications ranging from 10% to 4800% in steps of 60.
11. The processing resource utilization is set to 500%.
12. Applications are randomly mapped.
13. The routes for the communication tasks are selected using a shortest path algorithm.
14. For each configuration, 100 random test cases are generated and executed.

We use the following metrics for evaluation:

- **Schedulability:** An application will be unschedulable if the response time of any of its tasks exceed the application's deadline. A test case will be unschedulable if one of its applications is unschedulable. The schedulability metric is a measure of the percentage of schedulable test cases for a particular configuration.
- **Analysis time:** This is the time taken to compute the worst-case response times for all applications in a test case. For any given configuration, we report the average analysis time over all test cases.

Schedulability: Figure 6.5 shows the average schedulability of applications, with 10 tasks per application, against communication utilization for the various WCRT analysis techniques. Figure 6.5a shows the average schedulability for a 4×4 NoC with a deadline of each application equal to twice its period. The schedulability of the exponential version of OSLA is higher than that of the exponential versions of both OFLA and PAL. Furthermore, as we increase the communication utilization of the NoC, we observe that OSLA has a higher schedulability than the other analysis techniques. OSLA continues to schedule application sets even after OFLA and PAL fail to schedule any application set. This is because OSLA is able to better analyze workloads with a large amount of interference per stage. Since such interferences do not exist when the utilization is low, all three techniques do equally well. This holds for all graphs in Figure 6.5. From Figure 6.5b, we make the same observations with the period set to 10 times that of the period. Notice that increasing the deadline results in higher schedulability for all analyses simply because the deadline is larger. Figures 6.5c and 6.5d show the schedulability of the polynomial versions of the analysis for an 8×8 NoC. Once again, the polynomial version of OSLA outperforms OFLA and PAL in terms of schedulability.



(a) D=2*T, NoC=4x4

(b) D=10*T, NoC=4x4

(c) D=2*T, NoC=8x8

(d) D=10*T, NoC=8x8

Figure 6.5: Schedulability of application sets with 10 tasks per application

Computation Time: Figure 6.6 displays the run time of the exponential and polynomial versions of OSLA, OFLA, and PAL. The results show that the run time of the exponen-

tial versions grow exponentially as we increase the number of tasks per application when compared to the polynomial. It also shows that the run time for the exponential OSLA is larger than others. Exponential OSLA performs the WCRT analysis for an exponential number of activation patterns on multiple stages, thus the large run time. Exponential PAL has a run time larger than that of exponential OFLA. Although, both OFLA and PAL view a communication task's route as a single resource, exponential PAL considers indirect interference as direct interference. This increases the number of activation patterns that have to be considered which is exponential in an exponential analysis. Thus, leading to a higher execution time compared to exponential OFLA.

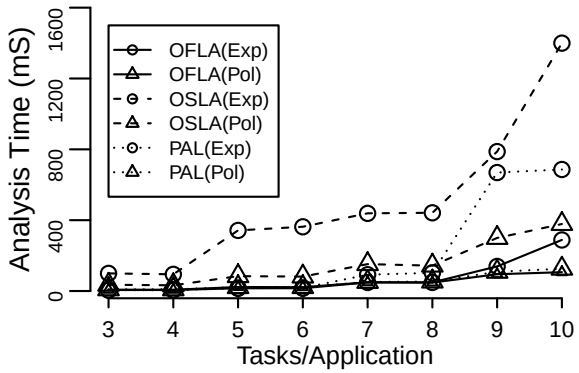


Figure 6.6: Run time comparison of OSLA, OFLA, and PAL

Summary: On average, OSLA improves schedulability by 48.3% and 66.7% over OFLA and PAL, respectively. OFLA improves schedulability, on average, by 38.1% compared to PAL.

6.6 Summary

This chapter presents a holistic analysis for computing offsets and jitters of tasks in a DAG task graph. We present OSLA and OFLA for computing the WCRT of applications in a pipelined communication resource model. A concrete application of this model is in estimating worst-case latencies across communication interconnects such as a priority-aware NoC. In developing these WCRT analysis techniques, we construct an exponential analysis, and its corresponding polynomial analysis. We provide proofs of correctness to ensure that the analyses provide upper bounds for the response times. To evaluate the analyses, we create two instances of a NoC, and deploy a large suite of synthetic benchmarks. These synthetic benchmarks are essential and necessary to stress test the analysis techniques. Our results show that OFLA improves schedulability by 38.1% compared to PAL for the two NoC sizes. The results also show that the schedulability of OSLA is 48.3% and 66.7% higher than the schedulability of OFLA and PAL for the two NoC sizes.

Chapter 7

Buffer Space Analysis for CMPs

Recent WCL analyses, including FLA, SLA, OFLA, and OSLA, have been developed for priority-aware networks with flit-level preemption. Priority-aware networks employ wormhole switching [106] and virtual channel resource allocation [31]. These techniques reduce the required buffer space, by handling packets at the flit level, and allow multiple flit buffers (virtual channels) to access the same physical channel. However, priority-aware networks are susceptible to chain-blocking (blocked flits spanning multiple routers). Chain-blocking creates back-pressure in the priority-aware network which eventually leads to blocking of the computation and communication tasks. Even though, it is clear that the blocking of tasks due to back-pressure affects the WCL of the communications tasks, recent analyses ignore blocking due to back-pressure and, hence, assume infinite buffer space in the network. This assumption is a serious impediment to implementing priority-aware networks because the buffer space required to guarantee the validity of the computed WCLs is unknown. This chapter presents the necessary buffer space bounds to ensure the validity of WCLs (FLA and SLA) and the compositionality of holistic WCL estimates (OFLA and OSLA). Reducing the buffer space beyond these necessary bounds, creates a back-pressure in the network that can affect the resultant WCL analysis. Later in this chapter, we also incorporate the effect of limited buffer space on the WCL analysis.

The main contribution in this chapter is the allocation of buffer space in priority-aware routers. NoC designs usually target specific applications, and, hence, customizing the design to limit cost, such as buffer space, is applicable [12, 18]. This chapter computes buffer space bounds to provide an ability to implement priority-aware networks and enable existing WCL analyses to provide timing guarantees for real-time applications. This enables us to design a priority-aware router with reconfigurable virtual channel buffers.

7.1 System Model

We use the same resource model as in Section 5.1. We also use the same communication task model presented in Section 5.2 with some minor extensions. We assume the assignment of distinct priorities to communication tasks. We also assume that a set of communication

tasks Γ is deployed on the communication resources, and that the communication tasks have fixed paths that are determined offline. The priority-aware network employs wormhole switching, and a credit-based task-control mechanism to prevent buffer overflow such as one used in the Æthereal NoC [44] and QNoC [18]. The credit feedback delay in the network is given by CF.

Each communication task τ_i has a basic stage latency L_i. The number of flits in one packet of the communication task τ_i is F_i. The basic stage latency L_i can, thus, be computed as $\frac{flit\_size * F_i}{bandwidth}$, which is the total packet size divided by the stage bandwidth. For clarity of presentation and without loss of generality, we assume that $\frac{flit\_size}{bandwidth} = 1$ to simplify the conversion between transmission latency and buffer space. The proportionality between transmission time and buffer space enables us to cleanly leverage WCL analyses for the buffer space computation.

7.2 Buffer Space Requirements

In this section, we introduce SLBA and FLBA to compute the buffer space required in the routers of the priority-aware NoC to guarantee a valid WCL analysis. Priority-aware routers implement a task-control mechanism to prevent buffer overflow. If a virtual channel in a receiving router is full, flits from the corresponding virtual channel in the sending router will be blocked until there is space at the downstream router buffer. This might lead to chain-blocking and the creation of back-pressure in the network which might invalidate the WCL analysis. Therefore, our goal is to compute the buffer space required for each virtual channel at each router to prevent back-pressure in the NoC. For each task τ_i, we compute the buffer space $\underset{s}{VC_i}$ (measured in flits) required at the virtual channels used by τ_i at each stage s along its path δ_i to avoid back-pressure in the network. First, we consider the simplest case.

Lemma 14. *Given a task τ_i that suffers no interference from higher priority tasks ($S_i^D = \emptyset$) and under the condition $D_i \leq T_i - J_i^R$ (no self-blocking), the buffer space required at each virtual channel along δ_i is $\underset{s}{\mathrm{VC}_i} = 1$.*

Proof. Since the routers implement the wormhole switching protocol, then each router will directly forward received flits to the next router in the path δ_i of τ_i. For example, if one router forwards a flit to another router at time t_1, the receiving router will forward the flit in the next cycle $t_1 + 1$. And since the virtual channels used by τ_i are used exclusively by τ_i along its path δ_i and no contention occurs for the network stages ($S_i^D = \emptyset$), then flits of τ_i are never blocked and are directly forwarded from one router to the other. Therefore, a buffer of one flit size suffices for the virtual channels of task τ_i in that case. □

Next, we consider two different interference scenarios:

S 1. *Interference from higher priority tasks with no self-blocking from packets of the same task under analysis.*

S 2. *Interference from higher priority tasks with self-blocking from packets of the same task under analysis.*

We introduce SLBA and FLBA to compute buffer space requirements for each interference scenario along the path of each task. If enough buffer space exists at each router to accommodate the blocked flits, there will be no back-pressure in the network.

7.2.1 Stage-Level Buffer-Space Analysis

SLBA computes the buffer space at each router on the path of the task under analysis. Using the interference on a specific stage on the path of task τ_i, we can compute the buffer space required in the virtual channel sending flits of τ_i on that specific stage. Theorems 11 and 12 compute the buffer space at each virtual channel for the interference scenarios S1 and S2, respectively.

Lemma 15. *Given a task under analysis τ_i suffering a worst-case interference $\underset{s}{I_i}$ from higher priority tasks in the set $\underset{s}{\mathbb{S}_i^D}$ on stage s on its path δ_i, an upper bound to the buffer space required at the corresponding virtual channel is given by $\underset{s}{\mathrm{VC}_i} = \underset{s}{I_i} + 1$.*

Proof. From Lemma 14, the buffer space required at each virtual channel along δ_i when no interference exists is one flit. Hence, if $\underset{s}{I_i} = 0$ on stage s, a buffer space of one flit will be needed at the corresponding virtual channel, i.e., $\underset{s}{VC_i} = 1$. However, if $\underset{s}{I_i} > 0$, more buffer space will be required to prevent back-pressure.

A priority-aware router forwards a flit to each of its output channels every cycle. Consider a flit of τ_i attempting to access stage s in a certain cycle. This flit can only be blocked by higher priority flits in the set $\underset{s}{\mathbb{S}_i^D}$. In the worst-case, this flit of τ_i will be blocked for a number of cycles equal to $\underset{s}{I_i}$. This will lead to other flits of τ_i accumulating in a FIFO order. To prevent back-pressure, enough buffer space is needed to buffer these flits of τ_i in the same virtual channel. Since a priority-aware router forwards a flit every cycle, then each cycle for which τ_i is blocked causes the accumulation of one more flit in the same virtual channel. Hence, in the worst-case, an extra buffer space of $\underset{s}{I_i}$ flits is needed to buffer flits accumulating from a maximum blocking time of $\underset{s}{I_i}$ cycles. Therefore, an upper bound to the buffer space required at the virtual channel buffering flits of task τ_i to access stage s is equal to $\underset{s}{I_i} + 1$. Note that, by definition, the worst-case latency of τ_i is a contiguous time interval during which flits of τ_i and higher priority flits access stage s, i.e., including the transmission of flits of tasks τ_i. Hence, no interference other than $\underset{s}{I_i}$ can occur before all blocked flits of τ_i can access stage s and the virtual channel becomes empty. □

We use Lemma 15 to derive the buffer space requirements for each of the different interference scenarios.

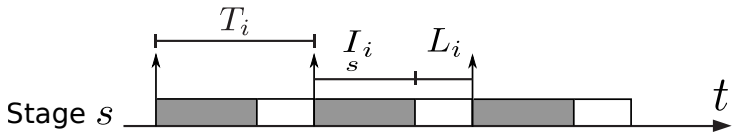


Figure 7.1: Interference scenario S1 on stage s

Theorem 11. *Given a task under analysis τ_i with WCL $\underset{s}{R_i}$ on stage s of its route δ_i suffering interference only from higher priority tasks in the set $\underset{s}{\mathbb{S}_i^D}$ under the condition $D_i \leq T_i - J_i^R$, the buffer space required at the corresponding virtual channel is given by:*

$$\underset{s}{\mathrm{VC}_i} = \min(L_i, \sum_{\forall \tau_j \in \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{R_i} + J_j^R + J_j^I}{T_j} \right\rceil * L_j + 1)$$

Proof. Since $\underset{s}{R_i}$ is the WCL of task τ_i on stage s, then the time window during which a higher priority task τ_j can interrupt τ_i is $R_i + J_j^R + J_j^I$. The number of times that task τ_j interrupts τ_i is equal to $\lceil \frac{R_i + J_j^R + J_j^I}{T_j} \rceil$. And the interference that τ_i suffers from the task τ_j is then equal to $\lceil \frac{R_i + J_j^R + J_j^I}{T_j} \rceil * L_j$. The maximum blocking $\underset{s}{I_i}$ that τ_i can suffer from all higher priority tasks is equal to the sum of the interference suffered from all higher priority tasks. Therefore, using Lemma 15, an upper bound to the buffer space $\underset{s}{VC_i}$ is equal to $\sum_{\forall \tau_j \in \underset{s}{\mathbb{S}_i^D}} \lceil \frac{R_i + J_j^R + J_j^I}{T_j} \rceil * L_j + 1$.

The required buffer size is, however, also bounded by the number of flits in one packet of the task, i.e., $F_i = L_i$. We use Figure 7.1 to prove that both bounds prevent back-pressure in the network. The figure shows flits of task τ_i (white) accessing stage s while suffering interference from higher priority flits (grey). Assume that the first flit of τ_i attempts to access stage s at a time t_1 (upward arrow in the figure). This flit will be blocked for an amount of time $\underset{s}{I_i} = \underset{s}{R_i} - L_i$.

Case $\underset{s}{I_i} + 1 < L_i$: At time $t_2 = t_1 + \underset{s}{I_i}$, the buffer will have $\underset{s}{I_i} + 1$ flits and the first flit will be accessing stage s, creating an empty slot for a new incoming flit. The last flit in the packet of task τ_i accesses the stage s at time $t_3 = t_1 + \underset{s}{I_i} + L_i$.

Case $L_i < \underset{s}{I_i} + 1$: At time $t_2 = t_1 + L_i$, the buffer will have all flits of the packet under analysis of task τ_i while the first flit is still blocked. After a blocking time $\underset{s}{I_i}$, the last flit in the buffered packet of task τ_i will access the stage s at time $t_3 = t_1 + \underset{s}{I_i} + L_i$.

In both cases, buffers sizes of $\underset{s}{I_i} + 1$ and L_i flits, respectively, will be sufficient under two conditions: (1) no new interference occurs before time t_3 and (2) no flits of another packet of τ_i arrive to the buffer before time t_3. By definition, the WCL $\underset{s}{R_i}$ is measured from the time the first flit of a packet of τ_i attempts accessing stage s (at time t_1) until the time when the last flit of the packet accesses the stage (at time t_3). Hence, no interference

other than $\underset{s}{I_i}$ (used to compute $\underset{s}{R_i}$) can occur before t_3 or else it would have been part of $\underset{s}{I_i}$. Thus, satisfying the first condition. The earliest time at which the first flit of a new packet attempts to access stage s is $t_4 = t_1 + T_i - J_i^R$. The flit of the new packet will be blocked if $t_4 < t_3$, i.e., $t_1 + T_i - J_i^R < t_1 + L_i + \underset{s}{I_i}$. Since $\underset{s}{I_i} = \underset{s}{R_i} - L_i$, then the blocking occurs when $T_i - J_i^R < \underset{s}{R_i}$. And for schedulability to be satisfied $\underset{s}{R_i} \leq D_i$. So the blocking occurs when $T_i - J_i^R < D_i$ which contradicts the no self-blocking condition $D_i \leq T_i - J_i^R$. Hence, the second condition is satisfied as well. Therefore, $\min(L_i, \underset{s}{I_i} + 1)$ is a safe upper bound to the buffer space $\underset{s}{VC_i}$ to prevent back-pressure. □

To compute the buffer space in the presence of self-blocking, we consider the busy period of task τ_i on stage s. The busy period is the longest contiguous time interval of flits of priority equal to or higher than that of τ_i accessing stage s before the stage becomes idle.

Theorem 12. *Given a task under analysis τ_i with a busy period $\underset{s}{B_i}$ suffering interference from higher priority tasks in the set $\underset{s}{\mathbb{S}_i^D}$ on stage s along the path δ_i, the buffer space required at the corresponding virtual channel is given by:*

$$\underset{s}{\mathrm{VC}_i} = \min(\underset{s}{p_{B,i}} * L_i, \sum_{\forall \tau_j \in \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{B_i} + J_j^R + J_j^I}{T_j} \right\rceil * L_j + 1)$$

Proof. The proof is similar to the proof of Theorem 11. The busy period $\underset{s}{B_i}$ is the time window during which interference from higher priority tasks can occur. Using Lemma 15, an upper bound to the buffer space $\underset{s}{VC_i}$ is equal to $\sum_{\forall \tau_j \in \underset{s}{\mathbb{S}_i^D}} \lceil \frac{\underset{s}{B_i} + J_j^R + J_j^I}{T_j} \rceil * L_j + 1$. The buffer space is also bounded by the number of flits of τ_i in the busy period which is equal to the number of packets $\underset{s}{p_{B,i}}$ multiplied by the latency (number of flits) in one packet L_i. The number of packets of τ_i in the busy period is equal to $\lceil \frac{\underset{s}{B_i} + J_i^R}{T_i} \rceil$.

If time t_1 is the time at which the first flit of τ_i attempts transmission in the busy period, then the time at which the last flit of τ_i accesses stage s is equal to $t_3 = t_1 + \underset{s}{I_i} + \underset{s}{p_{B,i}} * L_i$. We need to prove that the buffer space bound is sufficient under the two conditions mentioned in the proof of Theorem 11. The first condition (no further interference) is satisfied by the definition of a busy period. If further interference occurs before t_3, it would have already been accounted for by $\underset{s}{I_i}$. The first flit of a new packet (beyond the busy period) can attempt to access stage s at time $t_4 = t_1 + \underset{s}{p_{B,i}} * T_i - J_i^R$. This new flit can only be blocked if $t_4 < t_3$, i.e., when $\underset{s}{p_{B,i}} * T_i - J_i^R < \underset{s}{I_i} + \underset{s}{p_{B,i}} * L_i$. Since $\underset{s}{I_i} = \underset{s}{B_i} - \underset{s}{p_{B,i}} * L_i$, then blocking happens when $\underset{s}{p_{B,i}} * T_i - J_i^R < \underset{s}{B_i}$. This contradicts the definition of a busy period because if the length of the busy period exceeded the activation time of the new packet, it would

have been part of the busy period. Therefore, $\min(\underset{s}{p_{B,i}} * L_i, \underset{s}{I_i} + 1)$ is a safe upper bound to the buffer space $\underset{s}{VC_i}$ to prevent back-pressure. □

7.2.2 Flow-Level Buffer-Space Analysis

FLBA is applied for each task in the network while considering the whole path of the task as an indivisible unit during analysis. In this case, the buffer space computed using FLA will apply to each router along the path of the task under analysis. The theorems and proofs are similar to those used by SLBA but applied to the whole path of the task instead of each stage on its path. Hence, replacing L_j by C_j in the interference terms.

Theorem 13. *Given a task under analysis τ_i with WCL R_i suffering interference only from higher priority tasks in the set S_i^D along its path δ_i under the condition $D_i \leq T_i - J_i^R$, the buffer space required at each virtual channel used by τ_i along δ_i is given by:*

$$\underset{s}{\mathrm{VC}_i} = \min(L_i, \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{R_i + J_j^R + J_j^I}{T_j} \right\rceil * C_j + 1)$$

Theorem 14. *Given a task under analysis τ_i with a busy period B_i suffering interference from higher priority tasks in the set S_i^D along its path δ_i, the buffer space required at each virtual channel used by τ_i along δ_i is given by:*

$$\underset{s}{\mathrm{VC}_i} = \min(p_{B,i} * L_i, \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{B_i + J_j^R + J_j^I}{T_j} \right\rceil * C_j + 1)$$

7.2.3 Experimentation

We quantitatively evaluate the proposed buffer space computation techniques: SLBA and FLBA. We compare the proposed techniques to PAL [108] which was used for buffer space computation in [90]. We perform the evaluation on a set of synthetic benchmarks as in [90]. Synthetic benchmarks allow us to assess the effect of different factors (as well as their extreme values) on the buffer space computation. We perform our experiments on 4×4 and 8×8 instances of the priority-aware NoC. Our goals from these experiments are to:

1. Demonstrate the feasibility of computing buffer spaces for priority-aware networks
2. Quantify the reduction in the number of unfeasible implementations
3. Quantify the reduction in buffer space bounds computed by the proposed analyses
4. Compare the computation times of buffer spaces using PAL, SLBA, and FLBA

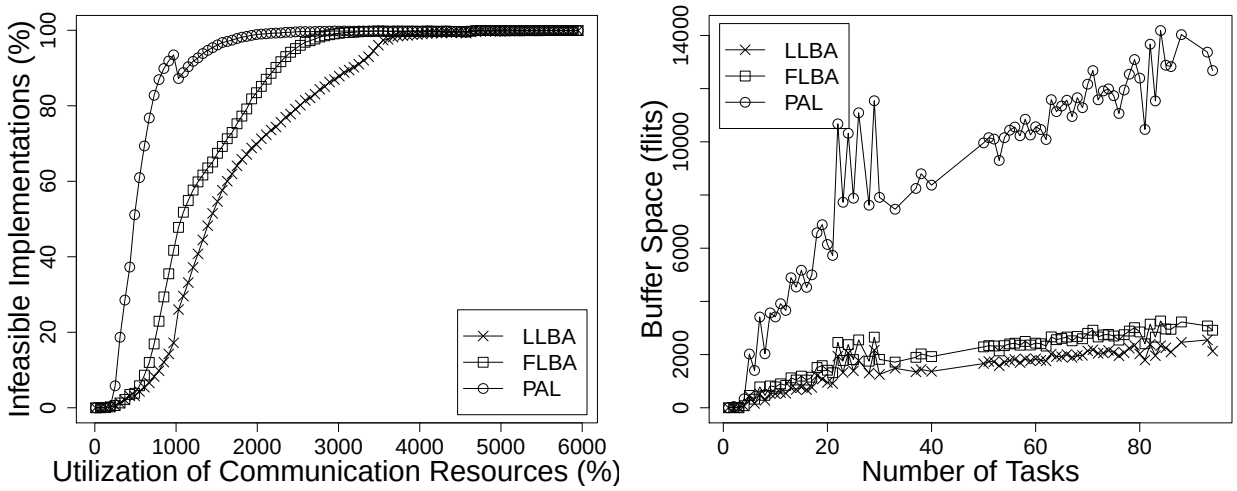
Our experiments involve changing multiple factors to evaluate their affect on buffer space computation:

1. The number of communications tasks varies from 1 task to 100 tasks (in steps of 1).
2. The source and destination pairs of the tasks are randomly mapped to the NoC.
3. The routes for the tasks are computed using a shortest-path algorithm.
4. A uniform random distribution is used to assign periods (or minimum interarrival time for sporadic tasks) T_i to communication tasks in the range (1000,1000000).
5. The task's deadline D_i is an integer multiple of its period.
6. An arbitrary priority assignment scheme is used for selecting task priorities.
7. The utilization of the NoC's communication resources varies from 10% to 6000% (in steps of 60%).
8. We have 40000 possible configurations and we generate 100 different test cases for each configuration.

We use the following evaluation metrics:

- **Infeasible implementations:** Some of the tasks might require an unbounded buffer space. This is equivalent to accumulating flits in a virtual channel's buffer without the flits getting a chance to access the communication channel. If one task requires unbounded buffer space, the test case will be considered as an infeasible implementation.
- **Buffer space requirements:** The buffer space requirement for a particular task is the sum of the buffer space needed at all the virtual channels along the task's route. For each configuration, we report the average buffer space requirement across all test cases. This metric is only valid for feasible implementations.
- **Computation time:** This is the time taken to compute the buffer space requirements for all tasks in a test case. For any given configuration, we report the average computation time over all test cases.

Infeasible Implementations: Figure 7.2a shows the percentage of infeasible implementations against the utilization of the communication resources in the priority-aware NoC. Increasing the utilization of the network stages results in an increase in the interference suffered by the communication tasks in the network. As the interference increases, some of the tasks will require an unbounded (infinite) buffer space. The graph shows that for any utilization, SLBA has the least percentage of infeasible implementations followed by FLBA followed by PAL. SLBA and FLBA reduce infeasible implementations by 42% and 27%, respectively compared to PAL.



(a) Percentage of infeasible implementations

(b) Buffer space against number of tasks

Figure 7.2: Infeasible implementations and buffer space requirements

Buffer Space Requirements: Figure 7.2b shows the required buffer space against the number of tasks at a network utilization of 910%. As the number of tasks increases, the required buffer space increases due to the increase of virtual channels and increase of interference in the network. SLBA has the least buffer space requirements followed by FLBA then PAL On average, SLBA and FLBA reduce the buffer space by 79% and 67%, respectively, compared to PAL.

Computation Time: Figure 7.3 shows the computation time of the buffer space analysis techniques against the number of tasks. To guarantee fairness between the different techniques, the computation time includes the time required to run the corresponding WCL analyses needed to apply the buffer space analysis techniques. PAL has the least computation time followed by FLBA then SLBA. On average, SLBA, FLBA, and PAL have computation times of 49 ms, 23 ms, and 16 ms, respectively.

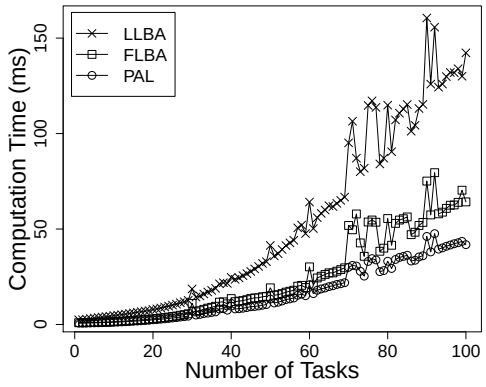


Figure 7.3: Computation time against number of tasks

7.3 Buffer Space Allocation

In the previous section, we computed the buffer space required for each virtual channel at each router to prevent back-pressure in the NoC. However, usually, there is a restriction on the buffer space available per router in the network. If the buffer space requirements, computed using SLBA or FLBA, exceed the buffer space restrictions at any router in the network, we cannot avoid back-pressure due to the restricted buffer space. In that case, the WCLs computed using SLA will be invalid because SLA does not consider this back-pressure. In this section, we extend SLA to include the effect of buffer space restrictions. We also present an algorithm to allocate the buffer space, available at each router, to the router's VCs. This algorithm operates by incrementally increasing the buffer space available for communication tasks, within the given buffer space restrictions, until the tasks meet their deadlines.

7.3.1 Stage-Level Analysis

First, we need to establish a separation between two types of interferences: (1) the interference that a task τ_i suffers from higher priority tasks on a stage s, and (2) the interference/blockage that τ_i suffers on stage s due to limited buffer space at the downstream router of the same stage. The first type of interference, $\underset{s}{I_i}$, results from the contention on stage s from higher priority tasks. The second type of interference results from back-pressure. It is the amount of time for which flits of τ_i are blocked from accessing stage s because the corresponding buffer in the downstream router is full. For clarity, we refer to the second type of interference by the term *blockage*. For a stage s, the upstream router is symbolized by $\underset{s}{UR}$, and the buffer space available for task τ_i in $\underset{s}{UR}$ is $\underset{s}{VC_i}$. The downstream router is denoted by $\underset{s}{DR}$ (which is also the upstream router of stage s^+, $\underset{s^+}{UR}$), and the buffer space available at $\underset{s}{DR}$ for task τ_i is $\underset{s^+}{VC_i}$.

First, we will compute the worst-case blockage suffered by task τ_i while ignoring credit feedback delay. Then, we extend it to include credit feedback delay. Finally, we present theorems for incorporating the buffer space restrictions into SLA.

Worst-Case Blockage without Credit Feedback Delay

A task under analysis τ_i will suffer a worst-case blockage $\underset{s}{IB_i}$ on stage s of its route due to limited buffer space at the downstream router of stage s. The amount of blockage on stage s due to back-pressure depends on the interference suffered by τ_i from higher priority tasks on the next stage s^+.

Lemma 16. *Given a task under analysis τ_i, the worst-case blockage that τ_i suffers due to back-pressure on the last stage $s_{|\delta_i|}$ of its route δ_i is $\underset{s}{\mathrm{IB}_i} = 0$.*

Proof. We showed in Section 7.2 that the buffer space $\underset{s}{VC_i}$ required by task τ_i at $\underset{s}{UR}$ depends on the interference suffered by τ_i on stage s. We also proved in Lemma 14 that the buffer space required when there is no interference from higher priority tasks is $\underset{s}{VC_i} = 1$. Since no stage follows the last stage $s_{|\delta_i|}$, task τ_i does not suffer interference on any stages following $s_{|\delta_i|}$. Hence, according to Lemma 14, the buffer space needed at the destination router of task τ_i is only one flit. A flit of task τ_i, at the destination router, will not be blocked because it will not traverse a stage on which it can suffer interference (we consider only interference on the network's stages). Since there is no blockage at the downstream router of stage $s_{|\delta_i|}$, then no blockage can occur on that stage due to back-pressure. Therefore, the worst-case blockage that τ_i suffers due to back-pressure on the last stage $s_{|\delta_i|}$ is $\underset{s}{IB_i} = 0$. □

We first consider the scenario when there is no self-blocking for the task under analysis, i.e., under the condition $D_i \leq T_i - J_i^R$. We compute $\underset{s}{IB_i}$ for any stage s on the route of task τ_i. To compute $\underset{s}{IB_i}$, we consider the interference that happens on stages following stage s on the route δ_i. Using this interference, we can find the amount of blockage that τ_i suffers on stage s because of the limited buffer space at $\underset{s}{DR}$.

Lemma 17. *Given a task under analysis τ_i with a worst-case latency $\underset{s}{R_i}$ on stage s of its route δ_i, under the condition $D_i \leq T_i - J_i^R$, and given that $\underset{s^+}{\mathrm{VC}_i} \geq L_i$, the worst-case blockage that τ_i suffers due to back-pressure on stage s is $\underset{s}{\mathrm{IB}_i} = 0$.*

Proof. In the proof of Theorem 11, we demonstrated that under the condition $D_i \leq T_i - J_i^R$, L_i is an upper bound to the buffer space required at $\underset{s}{UR}$. We also showed that, in the interval $\underset{s}{R_i}$, the maximum number of flits of task τ_i that can be transmitted is L_i. Consider, the buffer space $\underset{s^+}{VC_i}$ available for task τ_i at $\underset{s}{DR}$. If $\underset{s^+}{VC_i} \geq L_i$, this means that the buffer space will be enough to hold all flits of τ_i that can be transmitted within $\underset{s}{R_i}$. Hence, in that case, flits of τ_i cannot be blocked on stage s, and $\underset{s}{IB_i} = 0$. □

Lemma 18. *Given a task under analysis τ_i with a worst-case latency $\underset{s}{R_i}$ on stage s of its route δ_i, under the condition $D_i \leq T_i - J_i^R$, and given that $\underset{s^+}{\mathrm{VC}_i} < L_i$, the worst-case blockage that τ_i suffers due to back-pressure on stage s is given by:*

$$\underset{s}{\mathrm{IB}_i}(\underset{s}{R_i}) = \max\left(0, \underset{s^+}{\mathrm{IB}_i}(\underset{s}{R_i}) + \sum_{\forall \tau_j \in \underset{s^+}{\mathbb{S}_i^D} \setminus \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \underset{s^+}{\mathrm{VC}_i}\right)$$

Proof. For the last stage $s_{|\delta_i|}$, the succeeding stage s^+ does not exist. Hence, the term $\underset{s^+}{IB_i}(\underset{s}{R_i}) = 0$. Also, the set $\underset{s^+}{\mathbb{S}_i^D} \setminus \underset{s}{\mathbb{S}_i^D}$ does not contain any tasks. Hence, for the last stage $s_{|\delta_i|}$, $\underset{s}{IB_i}(\underset{s}{R_i}) = \max(0, -\underset{s^+}{VC_i}) = 0$ which is the one derived from Lemma 16. This forms the base case of our proof.

Our proof proceeds by induction. Assume that the lemma holds for stage s^+, and we prove the lemma for stage s. Consider the blockage $\underset{s^+}{IB_i}(\underset{s}{R_i})$ that τ_i suffers on stage s^+. Using the assumption that $\frac{flit\_size}{bandwidth} = 1$ from Section 7.1, this means that $\underset{s^+}{IB_i}(\underset{s}{R_i})$ flits will be blocked from accessing stage s^+. Due to back-pressure, this blockage will be propagated to stage s. The buffer at the downstream router of stage s will hold $\underset{s^+}{VC_i}$ flits of these blocked flits. Hence, the number of blocked flits on stage s will be equal to $\underset{s^+}{IB_i}(\underset{s}{R_i}) - \underset{s^+}{VC_i}$. This does not take into account any interference that occurs on stage s^+.

Consider the interference that τ_i suffers from higher priority tasks on stage s^+. These higher priority tasks can be divided into two groups: (1) tasks that also interfere with τ_i on stage s, and (2) tasks that only interfere with τ_i on stage s^+. Consider the first group of tasks. We proved in Lemma 3 that tasks of this group cannot cause more interference on stage s^+ than the interference they caused on stage s. We also established that, given interference only from this group of tasks, if a flit of τ_i transmits at time t on stage s, then it will transmit at time $t+1$ on stage s^+. Since, in that case, flits of τ_i are not blocked, this means that no blockage can occur from the first group of tasks. Now, consider the second group of tasks. These tasks cause *new* interferences on stage s^+ that did not exist on stage s. These new interferences will block the flits of τ_i. The number of blocked flits corresponds to the amount of new interference suffered on stage s^+. The set $\underset{s^+}{\mathbb{S}_i^D} \backslash \underset{s}{\mathbb{S}_i^D}$ contains the tasks causing new interferences. Hence, the term:

$$\sum_{\forall \tau_j \in \underset{s^+}{\mathbb{S}_i^D} \backslash \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j$$

computes the new interference suffered on stage s^+.

The total blockage that task τ_i can suffer on stage s, therefore, equals the blockage suffered on stage s^+ in addition to the new interference caused on stage s^+. However, the number of blocked flits on stage s are the ones that cannot be buffered at $\underset{s}{DR}$. Therefore, the blockage suffered on stage s now becomes:

$$\underset{s^+}{IB_i}(\underset{s}{R_i}) + \sum_{\forall \tau_j \in \underset{s^+}{\mathbb{S}_i^D} \backslash \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \underset{s^+}{VC_i}$$

Clearly, this blockage will only occur if the the buffer at the downstream router of stage s cannot hold all the blocked flits. This means that if the blockage on stage s is less than or equal to the number of flits $\underset{s^+}{VC_i}$, no blockage will occur. Hence, the usage of the max operator to set the blockage to zero in that case. □

We illustrate Lemma 18 using Figure 7.4. Task τ_i sends flits on stage s and suffers blockage from task τ_j that causes interference on stage s^+. The buffer space available for τ_i at $\underset{s}{DR}$, $\underset{s^+}{VC_i} = 5$. Hence, τ_i, while being blocked, can send five flits before $\underset{s}{DR}$ is full.

Task τ_j blocks τ_i from $t+1$ to $t+7$, i.e., a blockage of six flits. Task τ_i can send five flits between t and $t+5$, after which any blockage creates a gap during the transmission of τ_i on stage s. Task τ_j sends 12 flits on stage s^+. The blockage suffered by τ_i is equal to $12 - \underset{s^+}{VC_i} = 12 - 5 = 7$. Next, we compute $\underset{s}{IB_i}$ when self-blocking is possible.

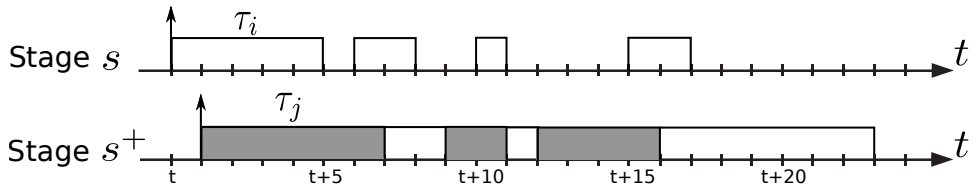


Figure 7.4: Illustration of Lemma 18

Lemma 19. *Given a task under analysis τ_i with a busy period $\underset{s}{B_i}$ on stage s of its route δ_i, and given that $\underset{s^+}{\mathrm{VC}_i} \geq \underset{s}{p_{B,i}} * L_i$, the worst-case blockage that τ_i suffers due to back-pressure on stage s is $\underset{s}{\mathrm{IB}_i} = 0$.*

Proof. In the proof of Theorem 12, we demonstrated that with self-blocking, $\underset{s}{p_{B,i}} * L_i$ is an upper bound to the buffer space required at $\underset{s}{UR}$. The rest of the proof is similar to the proof of Lemma 17. □

Lemma 20. *Given a task under analysis τ_i with a busy period $\underset{s}{B_i}$ on stage s of its route δ_i, and given that $\underset{s^+}{\mathrm{VC}_i} < \underset{s}{p_{B,i}} * L_i$, the worst-case blockage that τ_i suffers due to back-pressure on stage s is given by:*

$$\underset{s}{\mathrm{IB}_i}(\underset{s}{B_i}) = \max\left(0, \underset{s^+}{\mathrm{IB}_i}(\underset{s}{B_i}) + \sum_{\forall \tau_j \in \underset{s^+}{\mathbb{S}_i^D} \setminus \underset{s}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s}{B_i} + \underset{s}{J_j^R} + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \underset{s^+}{\mathrm{VC}_i}\right)$$

Proof. The proof is similar to that of Lemma 18. □

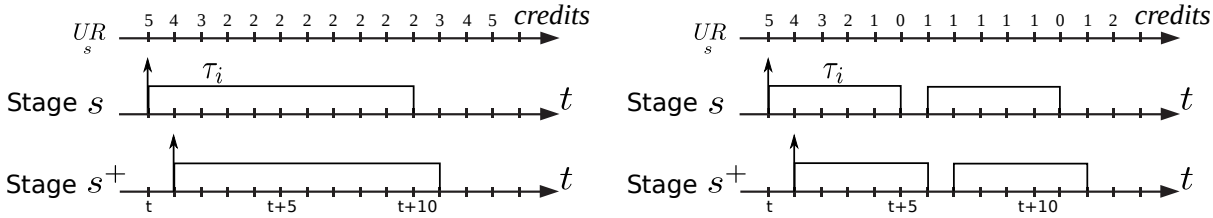
Credit Feedback Delay

So far, we have computed the blockage resulting from back-pressure in the network on any stage of the task's path. We, however, did not consider the credit feedback delay associated with the credit flow-control mechanism. Apart from the blockage that we computed in Lemmas 18 and 20, there is a credit feedback delay that is incurred every time the upstream buffer of a stage runs out of credit. We assume that once the buffer at the downstream router sends out a flit, it sends a credit back to the upstream router. The credit feedback delay, *CF*, is the time between the downstream router sending out the credit and its receipt by the upstream router (including any associated processing delays).

Lemma 21. *A task τ_i, without suffering any interference or blockage, can continuously send flits on stage s, only if $\underset{s^+}{\mathrm{VC}_i} \geq \mathrm{CF} + 1$.*

Proof. Assume that the first flit is sent at time t from the upstream buffer and reaches the downstream buffer at time $t+1$. Before time t, the upstream buffer has $\underset{s^+}{VC_i}$ credits equivalent to the size of the downstream buffer. The upstream buffer decrements a credit at time t as it sends a flit out. The credit corresponding to the first flit is sent upstream at time $t+1$, and is received by the upstream buffer after CF time units, i.e., at time $t+1+CF$. The upstream buffer will lose the last of its initial $\underset{s^+}{VC_i}$ credits at time $t + \underset{s^+}{VC_i} - 1$ while sending out the flit number $\underset{s^+}{VC_i}$. To be able to send a flit at time $t + \underset{s^+}{VC_i}$, the upstream buffer must receive the credit corresponding to the first flit it sent out by that time. Therefore, for τ_i to continuously send flits on stage s, the following condition must be satisfied: $t + 1 + CF \leq t + \underset{s^+}{VC_i}$, i.e., $1 + CF \leq \underset{s^+}{VC_i}$. □

We illustrate Lemma 21 using Figure 7.5. Task τ_i suffers no interference on stages s and s^+, and the buffer space available for τ_i at $\underset{s}{DR}$ is $\underset{s^+}{VC_i} = 5$. Hence, the upstream router of stage s, $\underset{s}{UR}$, starts with five credits at time t. In Figure 7.5a, the credit feedback delay is $CF = 3$, hence, the relation $\underset{s^+}{VC_i} \geq CF + 1$ is satisfied. When the downstream router of stage s, sends out the first flit at time $t+1$, it sends a credit upstream that is received at time $t+4$. Therefore, the credit being used by $\underset{s}{UR}$ at time $t+4$ gets replaced, and the number of available credits stays at two. This continues until task τ_i sends all of its flits on stage s. In Figure 7.5b, the credit feedback delay is $CF = 5$, hence, violating the relation $\underset{s^+}{VC_i} \geq CF + 1$. The first credit sent upstream by $\underset{s}{DR}$ at time $t+1$ will be received at time $t+6$. Hence, $\underset{s}{UR}$ will run out of credits at time $t+5$, and the flits of τ_5 will suffer a gap of one time unit during transmission.



(a) $CF = 3$ (b) $CF = 5$

Figure 7.5: Illustrative example of Lemma 21

Lemma 22. *The worst-case blockage suffered by a task τ_i on stage s, under the condition* $\underset{s^+}{\mathrm{VC}_i} \geq \mathrm{CF} + 1$, *increases by* $\mathrm{CF} + 1$ *time units when credit feedback delay is considered.*

Proof. Assume the first flit of τ_i is sent at time t from the upstream buffer of stage s, $\underset{s}{UR}$. The earliest time at which the credit for the first flit will be replaced is $t + 1 + CF$. At time $t + CF$, the number of credits will drop to $\underset{s^+}{VC_i} - CF$. From Lemma 21, we know that if there is no interference or blockage, flits will be continuously sent. In that case, at time $t + 1 + CF$, a credit will be sent and another will be received, fixing the number of credits

at $\underset{s^+}{VC_i} - CF$. This will continue as long as no interference or blockage is suffered on stage s.

Consider the time instant t_c which is the first time instant at which the number of credits available in $\underset{s}{UR}$ becomes 1. The least amount of blocking needed to reach the time instant t_c can be obtained by computing the drop required in the number of credits to reach 1 credit. This drop is equal to $\underset{s^+}{VC_i} - CF - 1$ credits. Since a flit is sent in one time unit, then a blockage of $\underset{s^+}{VC_i} - CF - 1$ flits is required to reach the time instant t_c. Note that if $\underset{s^+}{VC_i} = CF + 1$, the number of credits will drop to 1 after CF time units without any blockage ($\underset{s^+}{VC_i} - CF - 1 = 0$).

Any blockage suffered, beyond the $\underset{s^+}{VC_i} - CF - 1$ flits (required to reach t_c), will cause the number of credits to drop to zero. This creates a gap (empty time slot) on stage s. Hence, any blockage suffered, beyond a blockage of $\underset{s^+}{VC_i} - CF - 1$ flits, creates a gap on stage s. The size of this gap (in time units) is equivalent to the number of flits causing the blockage.

Consider the time $t_f > t_c$ at which the last flit sent by task τ_i on stage s is received by $\underset{s}{DR}$. Between t and t_f, task τ_i is either sending flits on stage s or suffering blockage (ignoring interferences on stage s). The blockage suffered in this time interval is only the blockage in excess of $\underset{s^+}{VC_i} - CF - 1$ flits. We showed in Lemma 18 that task τ_i suffers blockage in excess of $\underset{s^+}{VC_i}$. Therefore, the extra blockage suffered due to credit feedback delay is equal to $\underset{s^+}{VC_i} - (\underset{s^+}{VC_i} - CF - 1) = CF + 1$.

So far, we have ignored interference suffered on stage s. Any interference suffered on stage s will only stop $\underset{s}{UR}$ from sending flits. This means that $\underset{s}{UR}$ will stop using credits and can only gain credits. In that case, only more blockage might be needed until the number of credits drops to 1 and the blockage starts creating gaps on stage s. This extra blockage will always be less than or equal to the interference suffered. Therefore, $CF + 1$ is still an upper bound to the increase in blockage suffered by task τ_i on stage s due to credit feedback delay. □

We illustrate Lemma 22 using Figure 7.6, which is a rework of the example in Figure 7.4 while taking into account a credit feedback delay $CF = 2$. Task τ_i suffers blockage from τ_j and runs out of credit at $\underset{s}{UR}$ at time $t + 5$. After a blockage of six flits, the first credit is sent upstream at time $t + 7$ and is received at time $t + 9$. Between $t + 5$ and $t + 9$, τ_i suffers a blockage of four flits which is equivalent to $6 - \underset{s^+}{VC_i} + CF + 1 = 6 - 5 + 2 + 1 = 4$. Any further blockage by τ_j after time $t + 7$ creates an equivalent gap on stage s. The figure shows how any blockage beyond $\underset{s^+}{VC_i} - CF - 1 = 5 - 2 - 1 = 2$ flits, creates gaps in the transmission of the flits of τ_i on stage s. The total blockage suffered by τ_i is, therefore, $12 - \underset{s^+}{VC_i} + CF + 1 = 12 - 5 + 2 + 1 = 10$ flits.

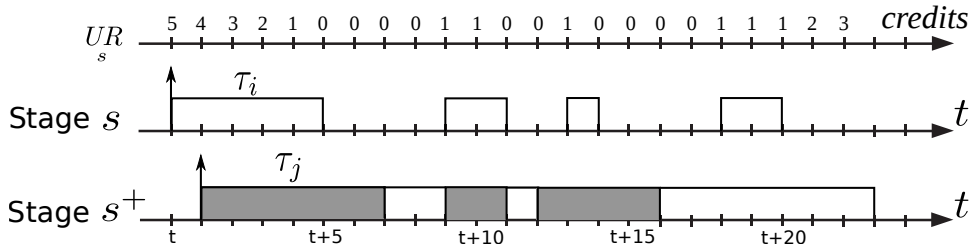


Figure 7.6: Illustration of Lemma 22

The Worst-Case Latency Analysis

Now, we can present the theorems for SLA while taking buffer space restrictions into account. Theorem 15 presents SLA when there is no self-blocking for the task under analysis, i.e., $D_i \leq T_i - J_i^R$. Theorem 16 presents SLA when self-blocking can occur.

Theorem 15. *The WCL R_i of a task τ_i along its path δ_i and under the conditions $D_i \leq (T_i - J_i^R)$ and $\underset{s}{\mathrm{VC}_i} \geq \mathrm{CF} + 1$, is given by:*

$$R_i = \underset{s}{R_i} + J_i^R + |\delta_i| - 1$$

where

$$\underset{s}{R_i} = \underset{s}{I_i} + \underset{s}{\mathrm{IB}_i}(\underset{s}{R_i}) + L_i$$

$$\underset{s}{I_i} = \underset{s'}{I_i} + \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \mathbb{S}_i^D} \left\lceil \frac{\underset{s'}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j$$

$$\underset{s}{\mathrm{IB}_i}(\underset{s}{R_i}) = \begin{cases} 0 & \text{if } \underset{s^+}{\mathrm{VC}_i} \geq L_i \\ \max\left(0, \underset{s^+}{\mathrm{IB}_i}(\underset{s}{R_i}) + \sum_{\forall \tau_j \in \mathbb{S}_{s^+}^D \setminus \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{R_i} + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \underset{s^+}{\mathrm{VC}_i} + \mathrm{CF} + 1\right) & \text{otherwise} \end{cases}$$

Proof. Taking buffer space restrictions into account, the WCL of task τ_i takes the buffer space blockage into account. From Lemma 17, the blockage $\underset{s}{IB_i} = 0$ when $\underset{s^+}{VC_i} \geq L_i$. Using Lemma 22, the blockage computed in Lemma 18 increases by $CF+1$ time units. Therefore, the WCL of task τ_i can be represented as shown above. □

Theorem 16. *The WCL R_i of a task τ_i along its path δ_i and under the condition $\underset{s}{\mathrm{VC}_i} \geq \mathrm{CF} + 1$, is given by:*

$$R_i = \max_{p=1 \ldots \underset{s}{p_{B,i}}} (\underset{s}{w_i}(p) - (p-1) * T_i + J_i^R) + |\delta_i| - 1$$

where

$$\underset{s}{w_i}(p) = \underset{s}{I_i}(p) + \underset{s}{\text{IB}_i}(\underset{s}{w_i}(p)) + p * L_i$$

$$\underset{s}{I_i}(p) = \underset{s'}{I_i}(p') + \sum_{\forall \tau_j \in \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \sum_{\forall \tau_j \in \mathbb{S}_i^D \cap \underset{s'}{\mathbb{S}_i^D}} \left\lceil \frac{\underset{s'}{w_i}(p') + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j$$

$$\underset{s}{\text{IB}_i}(\underset{s}{w_i}(p)) = \begin{cases} 0 & \text{if } \underset{s^+}{\text{VC}_i} \geq \underset{s}{p_{B,i}} * L_i \\ \max\left(0, \underset{s^+}{\text{IB}_i}(\underset{s}{w_i}(p)) + \sum_{\forall \tau_j \in \underset{s^+}{\mathbb{S}_i^D} \setminus \mathbb{S}_i^D} \left\lceil \frac{\underset{s}{w_i}(p) + J_j^R + \underset{s}{J_j^I}}{T_j} \right\rceil * L_j - \underset{s^+}{\text{VC}_i} + \text{CF} + 1\right) & \text{otherwise} \end{cases}$$

$$p' = \begin{cases} p & \text{if } p \leq \underset{s'}{p_{B,i}} \\ \underset{s'}{p_{B,i}} & \text{otherwise} \end{cases}$$

Proof. In Lemma 19, we showed that the blockage $\underset{s}{IB_i} = 0$ when $\underset{s^+}{VC_i} \geq \underset{s}{p_{B,i}} * L_i$. Using Lemma 22, the blockage computed in Lemma 20 increases by $CF + 1$ time units. To compute the worst-case completion time $\underset{s}{w_i}(p)$ of each job p in the busy period of τ_i, we compute the blockage suffered by p only during $\underset{s}{w_i}(p)$. Therefore, the WCL of task τ_i can be represented as shown above. □

Theorem 17. *The WCL computed using FLA is an upper bound to that computed using SLA while considering buffer space restrictions.*

Proof. The blockage computed using Theorems 15 and 16 is maximized when $-\underset{s^+}{VC_i} + CF + 1 = 0$. In that case, computing the blockage on one stage becomes equivalent to accounting for interferences that occur on subsequent stages. FLA treats the task's path as a single resource, and, hence, assumes that all interferences occur on this single resource. Therefore, in the worst-case, accounting for buffer space restrictions using SLA results in a WCL equivalent to the one computed using FLA. □

7.3.2 Allocation Algorithm

Normally, there is a limit on the buffer space available for each router in the NoC. This buffer space available to a router is distributed between the VCs of the tasks whose paths include this particular router. Increasing the buffer space for a task can help reduce the blockage that this task suffers. A buffer space allocation algorithm should achieve the following objectives:

Objective 1. *Schedule all tasks in a task set that is to be deployed on the NoC.*

Objective 2. *Minimize buffer space usage in the VCs of each router.*

Optimally, the algorithm would check every possible buffer configuration for the NoC to find the least buffer space usage while scheduling all tasks in the NoC. Such algorithm,

however, would be highly exponential. Therefore, we propose an algorithm that attempts to achieve these objectives using observations from the WCL analysis.

The new interferences suffered by the task under analysis on a stage s^+ contributes to the blockage suffered on the preceding stage s. The blockage contributing to the WCL is that in excess of $\underset{s^+}{VC_i} - CF - 1$. Hence, increasing the size of the buffer $\underset{s^+}{VC_i}$ in the router leading to the stage s^+ reduces the suffered blockage. Therefore, if a task is unschedulable, the proposed algorithm will primarily attempt to gradually increase the buffer space available in a VC preceding a stage on which interference occurs.

Algorithm 8 BUFFER SPACE ALLOCATION

Input: $\Gamma, B_{lim}, B_{step}$
Output: $\underset{s}{VC_i},\ \forall s \in \delta_i,\ \forall \tau_i \in \Gamma$

1: Set $\underset{s}{VC_i} = CF + 1,\ \forall s \in \delta_i,\ \forall \tau_i \in \Gamma$
2: **for all** $\tau_i \in \Gamma$ **do**
3: Compute R_i
4: **while** $R_i > D_i$ **do**
5: $s_{mod} = NULL$
6: **for all** $s \in \delta_i$ **do**
7: **if** $buffer(\underset{s}{UR}) < B_{lim}$ **and** $(\underset{s}{I_i} > \underset{s_{mod}}{I_i}$ **or** $(\underset{s}{I_i} = \underset{s_{mod}}{I_i}$ **and** $buffer(\underset{s}{UR}) < buffer(\underset{s_{mod}}{UR})))$ **then**
8: $s_{mod} = s$
9: **end if**
10: **end for**
11: **if** $s_{mod} \neq NULL$ **then**
12: $\underset{s_{mod}}{VC_i} = \underset{s_{mod}}{VC_i} + \min(B_{step}, B_{lim} - buffer(\underset{s}{UR}))$
13: **else**
14: Set Γ unschedulable and exit
15: **end if**
16: Compute R_i
17: **end while**
18: **end for**
19: **return** $\underset{s}{VC_i},\ \forall s \in \delta_i,\ \forall \tau_i \in \Gamma$

Algorithm 8 shows the proposed buffer space allocation algorithm. The inputs to the algorithm are the set of tasks Γ deployed on the NoC, the limit B_{lim} on the buffer space per router, and the step B_{step} by which the algorithm increments buffer space in VCs. A more efficient buffer space allocation can be obtained by decreasing B_{step} at the expense of the complexity and computation time of the algorithm. The output from the algorithm is a buffer space assignment to the VCs along the path of each task such that the tasks are schedulable. The algorithm uses the helper function, *buffer*, which returns the total buffer space usage of VCs in a specific router.

The algorithm starts by assigning the size of all VCs to $CF + 1$ flits. This is the minimum size of a VC for which Theorems 15 and 16 hold (line 1). The algorithm loops on all tasks in order of decreasing priority (line 2), and computes the WCL of each task using

Theorems 15 and 16 (line 3). The algorithm checks whether each task meet its deadline (line 4). If a task does not meet its deadline ($R_i > D_i$), the algorithm will start increasing the buffer space usage for this task. The algorithm modifies the buffer size of one VC at a time. The VC that the algorithm chooses to modify must have room for increasing its buffer space, i.e., $buffer(\underset{s}{UR}) < B_{lim}$. Amongst all VCs of a task, the algorithm chooses to modify the VC preceding the stage suffering most interference to reduce blockage. If multiple stages suffer the same interference, the algorithm will choose to modify the VC belonging to the router with the least buffer space usage (line 7). After selecting the VC to modify, the algorithm will increase its buffer size by the smaller of B_{step} and the buffer space available in the router before reaching B_{lim} (line 12). Note that if the algorithm cannot find a VC to modify, this means that all VCs of the task have allocated their maximum possible buffer space. In such case, the task and the task set are unschedulable, and the algorithm exits (line 14). After each VC modification, the algorithm re-computes R_i to check whether the task meets its deadline.

7.3.3 Experimentation

We quantitatively evaluate the modified SLA that takes buffer space restrictions into account. We also evaluate the proposed buffer space allocation algorithm. We perform the evaluation on a set of synthetic benchmarks as in [90]. We perform our experiments on 4×4 and 8×8 instances of the priority-aware NoC. Our goals from these experiments are to:

1. Demonstrate the feasibility of considering buffer space restrictions in the WCL analysis
2. Evaluate the performance of SLA compared to FLA when taking buffer space restrictions into account
3. Compare the buffer space bounds assigned using the proposed buffer space allocation algorithm against SLBA and FLBA
4. Compare the computation times of the proposed algorithm against SLBA and FLBA

We vary the following factors in our experiments:

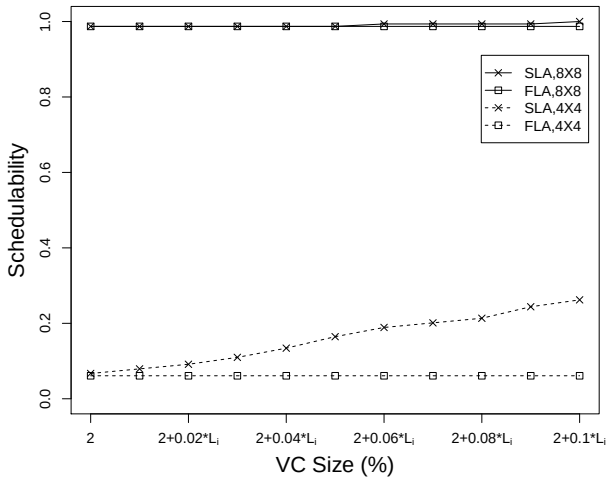
1. The number of communications tasks varies from 1 task to 100 tasks (in steps of 1).
2. The source and destination pairs of the tasks are randomly mapped to the NoC.
3. The routes for the tasks are computed using a shortest-path algorithm.
4. A uniform random distribution is used to assign periods (or minimum interarrival time for sporadic tasks) T_i to communication tasks in the range (1000,1000000).
5. The task's deadline D_i is an integer multiple of its period.

6. An arbitrary priority assignment scheme is used for selecting task priorities.
7. The utilization of the NoC's communication resources varies from 10% to 6000% (in steps of 60%).
8. The credit feedback delay is equal to the time taken to send one flit.
9. VC sizes are increased uniformly in 10 steps starting from two flits ($CF + 1$) with a step equal to $\frac{1}{100}$ from the size of a packet.
10. We have 400000 possible configurations and we generate 100 different test cases for each configuration.

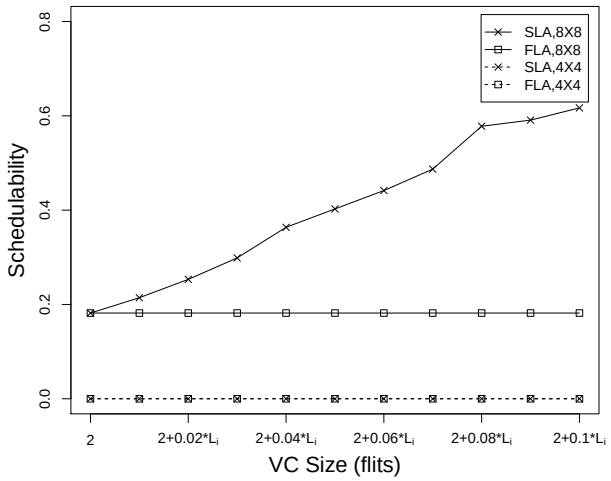
We use the following evaluation metrics:

- **Schedulability:** A test case will be unschedulable if one of the tasks in its task set is unschedulable. The schedulability metric is a measure of the percentage of schedulable test cases for a particular configuration.
- **Improvement in WCLs:** For each test case, we compute the WCL of each task using both SLA and FLA. Remember that FLA is still an upper bound to SLA even when buffer space restrictions are considered. We report the average improvement for a test configuration. This metric is only valid for schedulable tasks.
- **Analysis time:** This is the time taken to compute the latency bounds for all tasks in a test case using both SLA and FLA. For any given configuration, we report the average analysis time over all test cases.
- **Buffer space requirements:** The buffer space requirement for a particular task is the sum of the buffer space needed at all the virtual channels along the task's route. For each configuration, we report the average buffer space requirement across all test cases.
- **Algorithm Computation time:** This is the time taken to allocate buffer space using the proposed algorithm for all tasks in a test case. For any given configuration, we report the average computation time over all test cases.

Schedulability: Figures 7.7a and 7.7b show the schedulability against the buffer size of the VCs for task sets with 100 tasks and utilizations 1210% and 2410%, respectively. Note that FLA does not consider buffer sizes, and, hence, the schedulability result does not change for different buffer sizes. At a communication utilization of 1210%, SLA improves schedulability over FLA only slightly for 8×8 NoC instances as buffer space increases. For 4×4 NoC instances, the schedulability of SLA increases from 1.1 times that of FLA for a buffer size of two flits, up to 4.3 times for a buffer size equal to $2 + 0.1 * L_i$. The reason is that as more buffer space becomes available, blockages decreases and SLA computes tighter WCL bounds which increases schedulability. For a communication utilization of



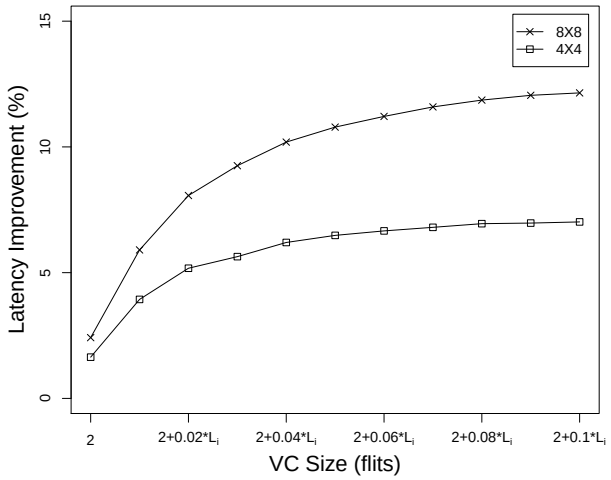
(a) U=1210%, 100 tasks



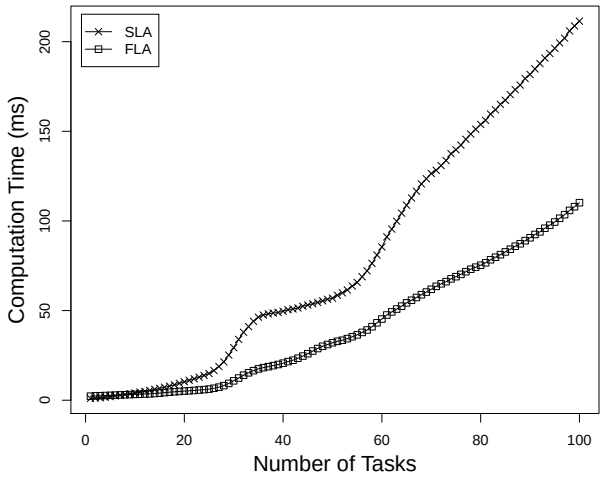
(b) U=2410%, 100 tasks

Figure 7.7: Schedulability results for SLA and FLA against VC size

2410%, both SLA and FLA cannot schedule any task sets with 100 tasks for 4×4 NoC instances due to high interferences. In 8×8 NoC instances, the schedulability of SLA is equal to that of FLA for a buffer size of two flits, and increases to 3.4 times that of FLA for a buffer size equal to $2 + 0.1 * L_i$. The schedulability of SLA increases over FLA as more buffer space is available for the tasks.



(a) U=3610%, 100 tasks



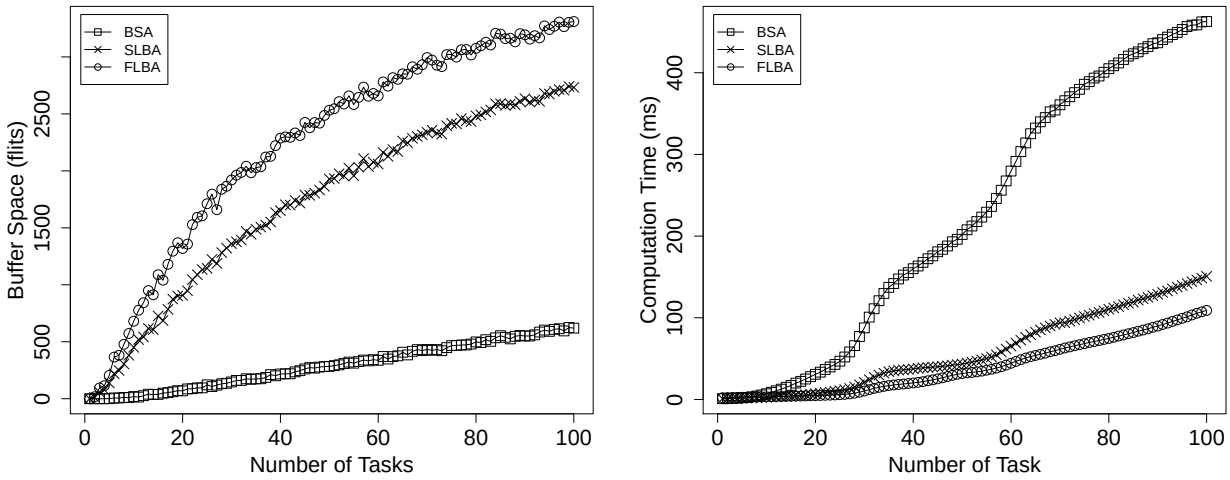
(b) Computation time

Figure 7.8: Latency and computation time results for SLA and FLA

Latency Improvement: Figure 7.8a shows the latency improvement against buffer sizes for task sets with 100 tasks and a communication utilization of 3610%. As more buffer space is available, the improvement in latencies computed by SLA over FLA increases from 1.6%

to about 7.0% for 4×4 NoC instances. For 8×8 NoC instances, this improvement increases from 2.4% to 12.1% as more buffer space is available for the tasks. The improvement in WCLs increases because when more buffer space is available, SLA computes tighter WCL bounds.

Analysis Time: Figure 7.8b compares the average computation times of both SLA and FLA. The analysis time for SLA is approximately double that of FLA, which is acceptable given the quality of the results of SLA.



(a) U=910%, 100 tasks (b) Computation time

Figure 7.9: Results for the buffer space allocation algorithm

Buffer Space Requirements: Figure 7.9a shows the average buffer spaces computed using the buffer space allocation algorithm, SLBA, and FLBA against the number of tasks at a network utilization of 910%. As the number of tasks increases, the required buffer space increases due to the increase of virtual channels and increase of interference in the network. The proposed buffer space allocation algorithm shows a large reduction in the buffer space requirements. On average, the buffer space allocation algorithm reduces the buffer space requirements by 85.1% and 88.9% compared to SLBA and FLBA, respectively.

Algorithm Computation Time: Figure 7.9b compares the average computation times of the buffer space allocation algorithm to SLBA and FLBA. The algorithm's computation time is about 3.8 times that of SLBA. This is a reasonable increase in computation time given the large reductions in the computed buffer space requirements.

Summary: The main conclusion from our experiments is that SLA is able to schedule task sets for buffers with a size of only two flits. The schedulability of SLA improves over FLA by only 0.1% for a buffer size of two flits and increases up to 11.9% for a buffer

size equal to $2 + 0.1 * L_i$. Compared to FLA, SLA reduces WCLs by 1.1% for two flit buffers, and by 4.7% for buffer sizes equal to $2 + 0.1 * L_i$. Over all test cases, the average analysis times using SLA and FLA are 81.3 mS and 40.5 mS, respectively. The buffer space allocation algorithm reduces the required buffer space to schedule task sets by 85.1% and 88.9% compared to SLBA and FLBA, respectively. Over all test cases, the average computation time of the buffer space allocation algorithm is 222.7 mS.

7.4 Summary

The increase in computational requirements of real-time software and the performance limitations of uniprocessors make CMPs with NoC interconnects a viable platform for real-time software. Although recent research focuses on WCL analysis techniques for priority-aware networks, buffer space requirements were not investigated. Limiting buffer space is also important to reduce silicon area and energy. Typically, NoCs are designed for a specific application or application-classes, hence, designers can customize buffer space based on application requirements. In this chapter, we extended the two most-recent analyses for WCL computation in priority-aware networks to compute buffer space requirements in priority-aware routers which guarantee the validity of the WCL analyses. Our experiments show that SLBA and FLBA reduce the number of infeasible implementations by 42% and 27% compared to PAL, respectively. SLBA and FLBA also reduce the required buffer space by 79% and 67%, respectively at the expense of an acceptable increase in computation time.

In this chapter, we also present theorems that incorporate buffer space limitations into the WCL bounds computed using SLA. SLA was able to schedule task sets with buffer spaces as small as two flits per VC. We also present a buffer space allocation algorithm that uses the extensions made to SLA. The proposed algorithm reduces the required buffer space to schedule task sets by 85.1% and 88.9% compared to SLBA and FLBA, respectively.

Chapter 8

Path Selection

We showed in Chapter 5, that at the expense of a detailed analysis, SLA results in significantly tighter bounds than FLA. These analyses model communication as periodic tasks on the priority-aware NoC, and they assume that the mapping of tasks, and the paths the tasks take are given. The set of tasks of an application are schedulable if the WCET or WCL of every task is less than or equal to its deadline. However, we notice that for an application with the same set of tasks and deadlines, the choice of paths can greatly influence the schedulability result of the entire application. Assuming a given mapping of the communication tasks onto the NoC, we contend that by judiciously selecting the paths the tasks take, we can increase the number of schedulable tasks; in turn, allowing more tasks to be schedulable.

In this chapter, we present a path selection algorithm assisted by SLA that aims to improve the number of schedulable tasks by selecting appropriate paths in the NoC [67]. We use SLA because it considers the pipelining effect of worm-hole switched NoCs, and it provides tight WCL bounds. This is unlike FLA, which treats the task as an indivisible unit across multiple network links. In particular, we propose a PSA that utilizes observations from SLA to efficiently select paths in the priority-aware NoC. PSA considers constraints on the number of virtual channels that can be present at router ports. To avoid the high complexity of an optimal algorithm, PSA uses heuristics to find least interference paths, and to consider lower priority tasks while selecting paths for the higher priority ones. We propose and compare six heuristics. For evaluation purposes, PSA operates using one of these heuristics.

8.1 System Model

We use the same resource model as the one presented in Section 5.1. We also use the same network model as in Section 5.2. For clarity, we assume a one cycle delay per stage. For an application with parallel tasks, we assume a given mapping of these tasks on the priority-aware NoC. We only consider priority-aware NoCs with mesh topologies. These tasks are marked as source and destination pairs based on the communication task between them.

All nodes of the priority-aware NoC contain both a processing element that executes tasks, and a router.

Recall that in the priority-aware NoC, the routers are priority-aware arbiters that implement worm-hole switching with flit-level preemption, and task control. The router architecture we employ was originally proposed in [126, 124], but for clarity we briefly describe its architecture. The router has a VC for every distinct communication task with a unique priority that passes through the router. Consequently, there exists a VC for each priority level. The VCs are designed as FIFO buffers at the input ports of the router. These FIFOs store the flits to be routed. The router selects the output port for a flit in the VCs based on its desired destination. When there are multiple flits waiting to be routed, the router selects and forwards the flit to the output port with the highest priority amongst all the waiting flits. Flow control guarantees that the router only sends data to the neighboring router if the neighbor has enough buffer space to store the data. If the highest priority flit is blocked in the network, the next highest priority flit can access the output link. Nodes are connected using bidirectional links with uniform bandwidth.

Since there is a VC for every distinct communication task, this guarantees that deadlocks due to cyclic dependencies never occur. The reason is that each task has its own buffers and thus never blocks another task for buffer space. If we, however, extend our model to allow sharing of VCs by multiple tasks, we must guarantee that our deterministic path selection algorithm is still deadlock-free. We can achieve this by ensuring that as the algorithm proceeds, we have an acyclic channel dependency graph [29]. Other algorithms have also been developed to ensure deadlock-freedom in wormhole NoCs [140].

8.2 Path Selection Algorithm

The path selection problem for deploying a hard real-time application on a priority-aware NoC is the following: discover possible paths on the NoC that tasks can take given their source and destination (V_s, V_d) pairs, and task requirements such that the tasks meet their respective deadlines. That is, given a graph $G = \langle V, E \rangle$, and a set of tasks $\Gamma = \{\tau_1, \ldots, \tau_k\}$, select a path δ_i for each task τ_i such that its worst-case latency is less than or equal to its deadline D_i.

8.2.1 Optimal Path Selection Algorithm

Objective. *Satisfy the deadline requirements of the tasks by searching all possible paths from source to the destination of each task.*

Assuming that a path visits a node only once, each task will have $4 * 3^{v-1}$ possible paths (assuming a NoC with v nodes) in a mesh topology because from each node descends three possible nodes to traverse (four for the source node). An optimal algorithm selects a path for the first task, then selects one for the second and so on. If at any point the worst-case latency is larger than the deadline then the algorithm backtracks one step, and selects

an alternative path. Hence, the decision tree has k levels corresponding to the number of tasks, and from each node descends $4 * 3^{v-1}$ choices that correspond to all possible paths yielding $O((3^v)^k)$. Due to the exponential complexity of the optimal path selection algorithm, it is necessary to find alternatives using heuristics.

8.2.2 Heuristic-based Path Selection Algorithm

The optimal path selection algorithm has exponential complexity that makes its applicability impractical. Therefore, we present a heuristic-based path selection algorithm that uses the stage-level analysis to guide the path selection process. Recall from Chapter 5 that the worst-case latency of a task on a link depends on its latency, and the interfering tasks on the preceding link. Our heuristics for path selection ensure that backtracking is not used, and all paths are not enumerated. In addition, the heuristic routes higher priority tasks while considering its impact on lower priority tasks. This is to avoid starving lower priority tasks by avoiding the assignment of critical links to higher priority tasks (if possible).

Objectives

Our overall goal with path selection is to select paths for communication tasks that improve their schedulability while incorporating the following objectives:

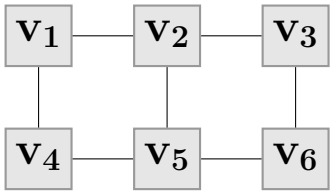
Objective 1. *Account for lower priority tasks.*

While selecting paths for higher priority tasks, expected paths for lower priority ones are taken into account to maximize schedulability. Otherwise, PSA may overload certain links that result in unschedulable lower priority tasks. By accounting for lower priority tasks, we prevent them from being starved.

Consider the example in Figure 8.1, in which we select paths for the tasks τ_0, τ_1, and τ_2 (order indicates decreasing priority). Task τ_1 has a tight deadline (close to its basic stage latency). And its total latency, if it suffers no interferences, will be $R_1 = L_1 + 2 - 1 = 3$. It is clear that τ_1 will miss its deadline if it suffers interference from the higher priority task τ_0. So if the path selection algorithm routes task τ_0 such that it does not interfere with task τ_1 (using the path $\langle V_4, V_1, V_2, V_3 \rangle$ for instance), then task τ_1 will meet its deadline. Hence, PSA attempts to account for lower priority tasks so that they can meet their deadlines. The different heuristics, that we propose, use different methods of identifying expected paths of lower priority tasks as discussed later in this section.

Objective 2. *Consider the availability of shortest paths.*

The criticality of links for lower priority tasks depends on their utilization as part of all available shortest paths. The intuition behind the criticality is that the more the number of available shortest paths, the more likely it is for a lower priority task to be schedulable. Similarly, the lower the number of available shortest paths, the less likely it is for the lower priority task to be schedulable. This concept is similar to that of critical links in [61, 38].



(a) Sample mesh NoC

Task	V_{s_i}	V_{d_i}	L	T	D
τ_0	V_4	V_3	2	10	10
τ_1	V_4	V_6	2	4	4
τ_2	V_5	V_3	2	5	5

(b) Data for communication tasks

Figure 8.1: An illustrative example for objectives 1 and 2 of PSA

Consider the example in Figure 8.1. Task τ_1 has a single shortest path: $\langle V_4, V_5, V_6 \rangle$. Task τ_2 has two shortest paths: $\langle V_5, V_6, V_3 \rangle$ and $\langle V_5, V_2, V_3 \rangle$. If both tasks have tight deadlines, the shortest paths' links of task τ_1 will be more critical than those of τ_2. This is because the more interference that τ_1 suffers on its shortest path, the less likely it is that τ_1 will be schedulable. Task τ_2, however, has higher chances of being schedulable because it has more shortest paths. Note that a task's shortest paths can share some links. For instance, task τ_0 has three shortest paths. Link (V_4, V_5), for instance, is common in two of them. Such link becomes more critical than a link that appears in only one shortest path. The different heuristics, that we propose, use more than one criterion to assign criticality to links. We discuss this in more detail later in this section.

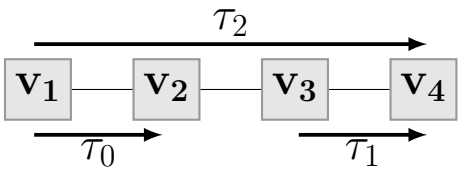
Objective 3. *Minimize the heterogeneity of interfering tasks.*

We promote the sharing of links between tasks that already interfered on previous links. Consider the example in Figure 8.2a. Using the data in Table 8.2b, we use SLA to compute the WCL of task τ_2. The latency of τ_2 on link (v_1, v_2) equals $R_2 = \lceil \frac{R_2}{4} \rceil * 2 + 5 = 11$. Task τ_2 suffers no interference on link (v_2, v_3) and, hence, it continues with the same latency of 11 time units. The latency on (v_3, v_4) is $R_2 = \lceil \frac{R_2}{4} \rceil * 2 + 11 = 23$. The total latency is then $R_2 = 23 + 3 - 1 = 25$.

Now assume, instead, that task τ_2 suffers interference only from τ_0 on all three links of the path δ_2. In this case, the WCL of τ_2 on the first link will be (as computed previously) $R_2 = \lceil \frac{R_2}{4} \rceil * 2 + 5 = 11$. On the second and third links, since no new interferences are introduced, the WCL remains the same $R_2 = 11$. The total WCL, including link delays, then becomes $R_2 = 11 + 3 - 1 = 13$. Hence, we identify from SLA that a task τ_i sharing multiple links with a task τ_j results in a lower worst-case latency than sharing fewer links with different tasks. Therefore, PSA should favor sharing links with tasks with which it had interfered with on previous links.

Objective 4. *Consider the maximum number of virtual channels per router port.*

Although PSA uses distinct priority assignment to communication tasks, the NoC architecture and available chip area might limit the number of virtual channels that can exist per router port. PSA attempts to find paths for communication tasks such that the virtual channel constraints are met.



(a) Deployment of communication tasks to NoC

Task	L	T	D
τ_0	2	4	4
τ_1	2	4	4
τ_2	5	30	30

(b) Data for communication tasks

Figure 8.2: An illustrative example for objective 3 of PSA

Algorithm

We use the interference that a task suffers on a link as the cost of that link. We construct the network graph G to capture the topology, and as the algorithm proceeds, it adds edges that represent sharing more than one successive link with the same task, but do not actually exist as links in the NoC. These edges hold the cost of interference over multiple links, and the intermediate nodes that represent actual NoC nodes. For example, if the algorithm selects the path $[v_1, v_2, v_3]$ for task τ_1, then when selecting the path for τ_2, the algorithm will set the interference for links (v_1, v_2) and (v_2, v_3), and creates a new edge (v_1, v_3) that has a cost of interference with τ_1 on both links and saves v_2 as an intermediate node. Although, this might not be an optimal solution with respect to space for representing all possible cases of multiple link interferences, this heuristic allows us to achieve Objective 3 efficiently and improve schedulability. The algorithm's complexity is discussed in more detail at the end of this section.

Since the algorithm does not enumerate all possible paths for a task, the order of assigning paths to tasks affects the latencies and the overall system schedulability. Accommodating multiple tasks in the network is known as the multi-commodity tasks problem, which is an NP-Complete problem [28, 139]. Notice that this makes the path selection algorithm intractable. Hence, we perform the path selection process according to the priority of the tasks: descending order of priorities (high to low). However, we still accommodate lower priority tasks while selecting a path for a higher priority task using six different heuristics. The algorithm operates using one of these six heuristics.

H 1. *Identify critical links as ones with least residual capacity. Assign weights to critical links based on the capacity required by each task being routed. Residual capacity is the difference between the capacity of a link and the capacity of the tasks being transmitted over it.*

H 2. *Identify critical links as the links constituting the paths of lower priority tasks with only a single shortest path.*

H 3. *Assign costs to all links in all shortest paths of each lower priority task. This cost depends on the number of available shortest paths to a task, and how critical the links are depends on how many shortest paths use the same link.*

H 4. *Identify critical links as ones with least residual capacity. Assign weights to critical links based on how close the best case latency of the task being routed to its deadline.*

H 5. *Identify critical links as the links constituting the paths of lower priority tasks with only a single shortest path. While routing a task, decrement the number of considered lower priority tasks until a selected path allows the schedulability of the task.*

H 6. *Assign costs to all links in all shortest paths of each lower priority task. While routing a task, decrement the number of considered lower priority tasks until a selected path allows the schedulability of the task.*

Algorithm 9 PATH-SELECTION H1

Input: $G\langle V, E\rangle$, $\Gamma = \{\tau_i : \forall i \in [1, k]\}$
Output: $\{\delta_i : \forall\ i \in [1, k]\}$

Let $LC \leftarrow \{\}$
for all $\tau_i \in \Gamma$ **do**
  Let $G'\langle V', E'\rangle$ s.t. $V' \leftarrow V$ and $E' \leftarrow E$
  $LC \leftarrow \mathsf{LowestCapacities}(G', v_{s_i}, v_{d_i})$
  $UE \leftarrow \{\}$
  for all $\delta_j \in LC$ **do**
    for all $e \in \delta_j$ **and** $e \notin UE$ **do**
      $UE = UE \cup \{e\}$
      $w \leftarrow \frac{L_i * bandwidth}{T_i * RC(e)}$
      $w(G', e) \leftarrow w(G', e) + w$
      $\mathsf{updateIntermediate}(G', e)$
    end for
  end for
  for all $e \in E$ **do**
    if $RC(e) < \frac{L_j * bandwidth}{T_j}$ **or** $\mathsf{tasks}(e) = \mathit{MAX\_VC}$ **then**
      $w(G', e) \leftarrow 0$
    end if
  end for
  $\delta_i \leftarrow \mathsf{Dijkstra}(G', v_{s_i}, v_{d_i})$
  $\delta_i \leftarrow \mathsf{expandIntermediate}(G', \delta_i)$
  $\mathsf{INTERFERENCE\text{-}COSTS}(G, \delta_i)$
end for
return $\{\delta_i : \forall\ i \in [1, k]\}$

Heuristics H1 and H4

Heuristics H1 and H4 are similar in the way they identify critical links. However, they assign weights differently to these links. When routing a particular task, H1 assigns weights to the critical links according to the capacity required by the task and the residual capacity on the links. If the task requires a high capacity, H1 will assign a high weight to the critical links. On the other hand, H4 assigns weights to critical links according to the slack the task being routed has. The proposed algorithm computes a task's slack as the difference between the task's deadline and its basic latency. The more the slack to the deadline,

a larger weight will be assigned by the algorithm. The intuition is that as more slack is available, the higher the chance is of meeting the deadline when avoiding critical links.

Algorithm 9 shows how Heuristic H1 operates. The input to the Algorithm 9 is a priority-aware NoC with a mesh topology of size $n \times n$ represented as a graph $G = \langle V, E \rangle$, and Γ with k tasks ordered according to their priorities with 1 being the highest. The output is a set of paths for each of the tasks in Γ. When selecting a path for a task, the algorithm updates the cost of that path in the graph G. The cost on each edge accounts for both higher and lower priority tasks (using one of the six heuristics).

Each selected path will add interferences to the graph according to Function 10. The function call $edgeInterference(G, e)$ calculates the interference on edge e, and function $nodeInterference(G, [v_i, ..., v_j])$ computes interference on a sequence of adjacent nodes $[v_i, \ldots, v_j]$ forming a path. When the algorithm selects a path for a task, it adds edges between each node on the path and all of its successive nodes. The call *intermediate* saves the intermediate nodes for newly created edges. If the edge is an actual link, then the algorithm will add the interference on that link to the weight of the edge. However, if it is not a link, then the algorithm will set the weight of the edge to the interference on the path formed by the intermediate nodes of that edge in one of three cases: (1) the edge does not exist, or (2) the edge exists and has the same intermediate nodes as the one the algorithm is adding, or (3) the interference on the edge being added is less than the existing one.

Function 10 INTERFERENCE-COSTS

Input: G, $\delta = [v_s, \ldots, v_d]$
  for all $v_i \in \delta$ **do**
    for $v_j \in [v_i, \ldots, v_d]$ **do**
      $e \leftarrow (v_i, v_j)$
      if $j - i = 1$ **then**
        $w(G, e) \leftarrow$ edgeInterference(G, e)
      else if $w(G, e) = 0 \vee$ nodeInterference$(G, [v_i, \ldots, v_j]) < w(G, e) \vee$ intermediate$(G, e) = [v_i, \ldots, v_j]$ **then**
        $w(G, e) \leftarrow$ nodeInterference$(G, [v_i, \ldots, v_j])$
        intermediate$(G, e) = [v_i, \ldots, v_j]$
      end if
    end for
  end for

To account for lower priority tasks using H1, the algorithm finds the least capacity paths by calling the function *LowestCapacities*. The *LowestCapacities* algorithm is a variation of Dijkstra's algorithm that finds paths with lowest capacities (LC) [61, 38]. For each unique edge in the least capacity paths (edges in the set UE), the algorithm adds weights to the links as shown on line 10. The residual capacity on an edge e is represented as $RC(e)$. The function $updateIntermediate(G, e)$ updates the costs of all edges that do not belong to the topology if they have the edge e as an intermediate edge. The algorithm, then, eliminates links that have a residual capacity less than the required bandwidth. The function $tasks(e)$ finds the number of tasks traversing a particular edge. The algorithm also eliminates links

that have reached the maximum number of virtual channels, *MAX_VC*, at the receiving router port.

Dijkstra's algorithm is used to find the least cost path for the task being routed. The function $expandIntermediate(G, \delta)$ replaces edges in a path that do not belong to the actual topology with the equivalent intermediate nodes. This algorithm operates in a manner similar to that of minimum interference routing algorithms but with a few differences: (1) uses link-level interference for costs on links versus capacities, (2) keeps track of the cost of sharing multiple links with higher priority tasks, and (3) assigns weights differently. Heuristic H4 operates like H1 but assigns weights according to the equation $w \leftarrow D_k - L_k + \Delta x + \Delta y$ where Δx and Δy are the horizontal and vertical displacements, respectively, of the source and destination nodes.

Heuristics H2, H3, H5, and H6

Algorithm 11 PATH-SELECTION H3

Input: $G\langle V, E\rangle$, $\Gamma = \{\tau_i : \forall i \in [1, k]\}$
Output: $\{\delta_i : \forall\ i \in [1, k]\}$

Let $SPC[1, n-1][1, n-1] \leftarrow [\ ]$
Let $SPE[1, n-1][1, n-1] \leftarrow [\ ]$
$SPC[i, j] \leftarrow MAX\ \forall\ (i, j) \in M$
LOWER-PRIORITY$(SPC, SPE, n-1, n-1)$
for all $\tau_i \in \Gamma$ **do**
  Let $G'\langle V', E'\rangle$ s.t. $V' \leftarrow V$ and $E' \leftarrow E$
  for $\tau_j \in [\tau_{i+1}, \ldots, \tau_k]$ **do**
    $\Delta x \leftarrow |(v_{s_k} \bmod n) - (v_{d_k} \bmod n)|$
    $\Delta y \leftarrow |v_{s_k}/n - v_{d_k}/n|$
    for all $e \in SPE[\Delta x, \Delta y]$ **do**
      $w \leftarrow \frac{L_k}{D_k - C_k} \times \frac{\mathsf{count}(SPE[\Delta x, \Delta y], e)}{SPC[\Delta x, \Delta y]}$
      $w(G', e) \leftarrow w(G', e) + w$
      $\mathsf{updateIntermediate}(G', e)$
    end for
  end for
  for all $e \in E$ **do**
    if $\mathsf{tasks}(e) = MAX\_VC$ **then**
      $w(G', e) \leftarrow 0$
    end if
  end for
  $\delta_i \leftarrow \mathsf{Dijkstra}(G', v_{s_i}, v_{d_i})$
  $\delta_i \leftarrow \mathsf{expandIntermediate}(G', \delta_i)$
  INTERFERENCE-COSTS(G, δ_i)
end for
return $\{\delta_i : \forall\ i \in [1, k]\}$

Heuristic H2 assigns weights to the lower priority tasks with only a single shortest path. The idea is based on helping higher priority tasks avoid these links to give a chance for

lower priority tasks to meet their deadline. Heuristic H5 is similar to H2 but decrements the considered lower priority tasks until the task being routed meets its deadline. The algorithm for H5 starts by taking into account all lower priority tasks. If the task being routed is unschedulable, then the algorithm will remove the assigned weights for the least priority task. The algorithm keeps on decreasing the number of considered tasks until the task being routed is schedulable or all lower priority tasks have been dropped. Heuristic H3 assigns weights to all links of all shortest paths of lower priority tasks. The weights being assigned are based on how critical the links are in the shortest paths and the number of available shortest paths for each lower priority task. Heuristic H6 performs similarly, but decrements the considered lower priority tasks until the task being routed meets its deadline.

Algorithm 11 shows the operation of heuristic H3. The inputs and outputs of Algorithm 11 are similar to those of Algorithm 9. To account for lower priority tasks using H3, the algorithm finds the number of shortest paths available, and the number of times each edge is used amongst all the shortest paths for every lower priority task. The simplest method to obtain this information is by finding all shortest paths for a task, which for a mesh topology has a complexity of $2^{2*(n-1)}$, i.e. 2^n. However, notice that that the information we require depends only on the relative x and y positions of the source and destination nodes: Δx and Δy. Hence, we use memoization, which is a form of dynamic programming that reduces the complexity to n. The algorithm makes a single call to Function 12 that calculates the number of shortest paths, and the count of each edge on these paths for all possible combinations of Δx and Δy. Array SPC stores the number of shortest paths available for a given Δy and Δx, and array SPE stores, for every Δy and Δx, the number of times each edge appears on these shortest paths. SPC has n^2 entries while SPE has $n^2 * 2n * (n-1)$ entries. The function recursively uses the information from nodes with lower values of Δy and Δx.

Algorithm 11 adds costs to the edges based on a speculation of the paths that will be selected for lower priority tasks as described above. The function $count(SPE[i,j],e)$ retrieves the count of an edge e for a specific Δy and Δx. The algorithm calculates Δy and Δx for each lower priority task and adds a cost to the links involved. A weight is used to represent the criticality of the edge which is equal to the edge count in SPE divided by the number of shortest paths. This weight is multiplied by, L_k, the basic link latency of the lower priority task over the slack that it has to its deadline on the speculated path where D_k is the deadline and C_k is the basic latency. The algorithm eliminates links that have utilized the maximum number of virtual channels. Dijkstra's algorithm is used to find the least cost path for the task being routed.

Heuristic H2 only accounts for lower priority tasks with only a single shortest path. Hence, the criticality of the edges for H2 is the same for all considered shortest paths, i.e., it does not require knowledge of the edge count *SPE* or the shortest paths count *SPC*. Therefore, Algorithm 11 for H2 will not instantiate *SPE* and *SPC* arrays or call *LOWER-PRIORITY*. The algorithm will only update edge costs if a lower priority task has a single shortest path, i.e., $(\Delta x = 0 \vee \Delta y = 0)$. The weight assigned to all edges of the shortest path is equal to the basic link latency over the slack that it has to its deadline on the speculated path, i.e., line 11 in Algorithm 11 would be $w \leftarrow \frac{L_k}{D_k - C_k}$.

Function 12 LOWER-PRIORITY

Input: SPC, SPE, i, j
  if $SPC[i,j] < MAX$ **then**
    return $SPC[i,j]$
  end if
  if $i = 0$ & $j = 0$ **then**
    $SPC[i,j] \leftarrow 1$
    $SPE[i,j] \leftarrow [\,]$
  else if i $= 0$ **then**
    $SPC[i,j] \leftarrow$ LOWER-PRIORITY$(SPC, SPE, i, j-1)$
    $SPE[i,j] \leftarrow SPE[i,j-1] + edge(i,j,i,j-1)$
  else if j $= 0$ **then**
    $SPC[i,j] \leftarrow$LOWER-PRIORITY$(SPC, SPE, i-1, j)$
    $SPE[i,j] \leftarrow SPE[i-1,j] + edge(i,j,i-1,j)$
  else
    $SPC[i,j] \leftarrow$LOWER-PRIORITY$(SPC, SPE, i, j-1)+$
    LOWER-PRIORITY$(SPC, SPE, i-1, j)$
    $SPE[i,j] \leftarrow SPE[i,j-1] + SPE[i-1,j] + edge(i,j,i,j-1) + edge(i,j,i-1,j)$
  end if

Heuristics H5 and H6 are equivalent to H2 and 3, respectively, but with a slight modification to Algorithm 11. An extra loop will exist outside the loop at line 7. The new loop will decrement the set of considered lower priority tasks when assigning weights until the task for which the algorithm is selecting a path becomes schedulable.

Complexity Analysis

Recall that we have k tasks in a graph with v vertices. Function 12 has a complexity v and is called only once. At worst, each task will have a path with v nodes. The number of edges that the algorithm will create is given by: $k*((v-2)+(v-3)+\ldots+1) = k*\sum_{i=2}^{v-1}(v-i)$. Thus, in the worst case, the algorithm will create $k*v^2$ edges. The functions $Dijkstra$ and $expandIntermediate$ have linear complexity v. The function $updateIntermediate$ uses a structure that saves, for each edge e, newly created edges which e is a part of as an intermediate edge. The maximum number of edges that can be involved in all shortest paths between two nodes is $2(v-\sqrt{v})$. The overall complexity of Heuristic H3 is therefore given by: $k^2 * 2(v-\sqrt{v}) * k * v^2$ which is $O(k^3 * v^3)$. Although, this is a high complexity compared to minimum interference algorithms for example, PSA never elicits that upper bound computation time which assumes that each edge is part of all newly created edges.

Heuristic H2 only updates costs of edges on single shortest paths, i.e., line 10 in Algorithm 11 will only loop on edges in a single path. In a $\sqrt{v} \times \sqrt{v}$ mesh topology, a single shortest path means that $(\Delta x = 0 \vee \Delta y = 0)$. Thus, the number of edges in such a path is at worst $\sqrt{v} - 1$. The overall complexity of Heuristic H2 is therefore given by: $k^2 * (\sqrt{v}-1) * k * v^2$ which is $O(k^3 * v^{2.5})$.

Heuristics H5 and H6 are similar to H2 and H3, respectively, but have an extra loop to

decrease the number of considered tasks. This loop raises the order of k in the complexity by one. Thus, leading to a complexity of $O(k^4 * v^{2.5})$ and $O(k^4 * v^3)$ for H5 and H6, respectively.

Heuristics H1 and H4 identify links with least residual capacity when routing each task. The *LowestCapacities* algorithm has a complexity $O(v^2)$ [38]. The number of links in the mesh is equal to $2(v - \sqrt{v})$. Calling the function *updateIntermediate* for all critical links has a complexity of $O(2(v - \sqrt{v}) * 2(v - \sqrt{v})) = O(v^2)$. Hence, for k tasks, the complexity is equal to $O(k * v^2)$. Table 8.1 summarizes the complexity of all heuristics.

Table 8.1: Complexity of the different heuristics

Heuristic	Complexity
H1	$O(k * v^2)$
H2	$O(k^3 * v^{2.5})$
H3	$O(k^3 * v^3)$
H4	$O(k * v^2)$
H5	$O(k^4 * v^{2.5})$
H6	$O(k^4 * v^3)$

8.3 Experimentation

We quantitatively compare PSA against the WSP algorithm, and MIRA [61, 38]. We also evaluate the operation of PSA using each of the six proposed heuristics. In Section 8.3.1, we prove the statistical significance of our experimentation results. All tests were run on an AMD Opteron 6174 2.2 GHz processor with 8.0 GB of memory.

We vary several parameters during experimentation to assess their effect on the schedulability of tasks, and the execution time of the different algorithms. The proposed ranges for the parameters are either in accordance with or cover a wider range than the ones used in [126]. Note that for the utilization range, we start from an 0.4 link utilization to stress test the different algorithms at higher utilization values. Our experiments use 4×4 and 8×8 mesh topologies for the NoC. We vary the following factors in our experiments:

1. The basic link latency of a task is randomly chosen from a uniform distribution in the range $[16, 1024]$.
2. We vary the link utilization between $[0.4, 0.85]$ in step increments of 0.05. The link utilization of a task τ_i is its basic link latency divided by its period: $U_i = L_i/T_i$.
3. The deadline D_i takes values between $[0.7, 1.0]$ in increments of 0.1 as a ratio of period T_i.
4. The number of tasks in the network ranges between $[10, 100]$ in steps of 10.
5. The maximum allowed number of virtual channels per router port is 4.

6. The mapping of the source and destination nodes for the communication tasks is random.

7. Priorities are randomly assigned to the tasks.

8. With these parameters, we have a full factorial experiment with 800 different configurations. We generate 1000 test cases per configuration.

We use the following metrics to quantitatively evaluate the performance of the heuristics that we propose:

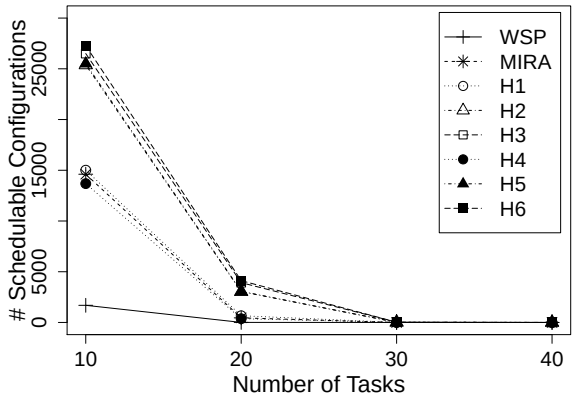
- **Number of schedulable configurations:** A task is unschedulable in one of two cases: 1) total worst-case latency of its route is larger than its deadline, or 2) no route is available that satisfies the bandwidth requirements of the task. If, for any given test case, any task is unschedulable, then the test case is unschedulable. Comparing the number of schedulable configurations across the different heuristics measures their ability to make the best use of the NoC resources to schedule an application. A higher number of schedulable configurations is better.

- **Ratio of unschedulable tasks:** For each test case, we measure the number of unschedulable tasks for each path selection algorithm. The ratio of unschedulable tasks for any given test case shows how one path selection algorithm improves schedulability over another. The average ratio of unschedulable tasks at a certain factor level is the geometric mean of all ratios of one path selection algorithm to the other at that factor level. Note that the ratio of unschedulable tasks can still be calculated for an unschedulable configuration. A lower ratio of unschedulable tasks is better.

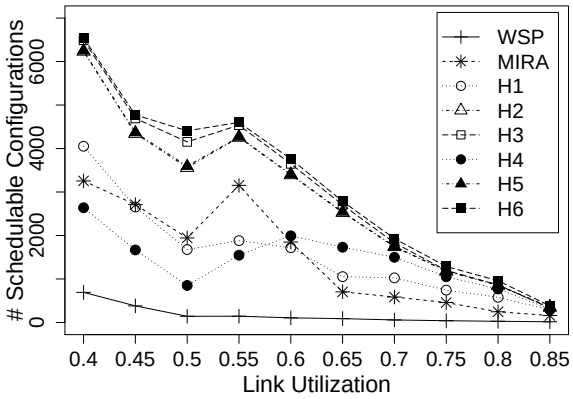
- **Computation time:** The computation time for a test case is the time the algorithm spends in selecting paths for all tasks in the test case. This metric enables us to reason about the feasibility of using a path selection algorithm in the deployment of applications to a NoC.

Schedulability of Configurations

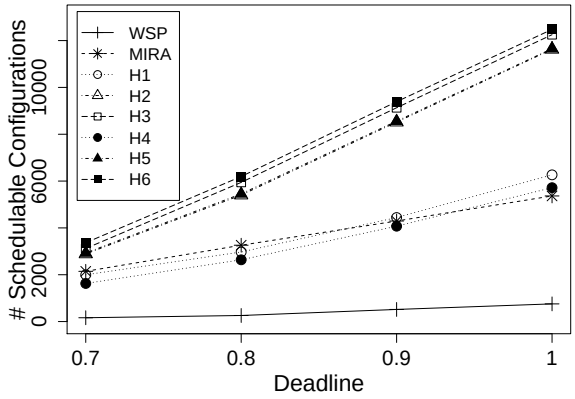
Figure 8.3a shows the number of schedulable configurations for each of the heuristics against the number of tasks. Each point is an average of 80 configurations with 1000 test cases per configuration. For 40 tasks and beyond, the number of fully schedulable configurations drops to zero for all path selection algorithms, because the number of tasks becomes too large to fit in a 4×4 or an 8×8 mesh. As the number of tasks increase, the number of schedulable configurations for any heuristic drops. WSP has the lowest number of schedulable configurations. H1, H4, and MIRA perform closely with H1 performing slightly better than MIRA which in turn performs slightly better than H4. These three algorithms identify critical links based on residual capacity and perform approximately alike. Heuristics H3 and H6 schedule more configurations than H2 and H5. H6 performs better than H3, and H5 performs better than H2. Using the shortest paths of lower priority tasks to identify



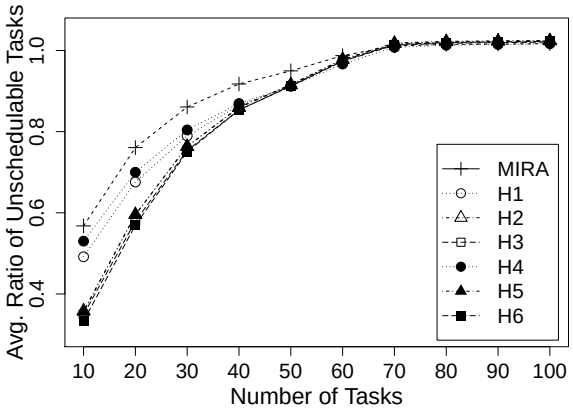
(a) Schedulable configurations vs number of tasks



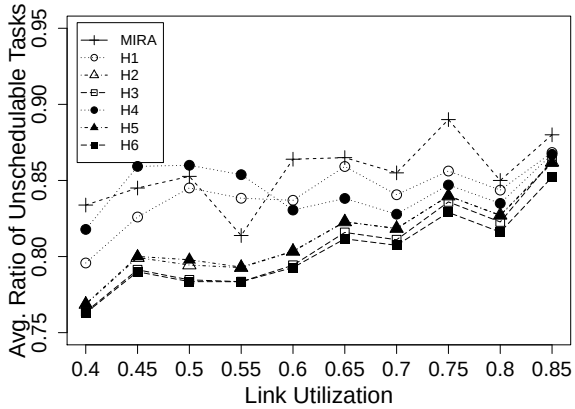
(b) Schedulable configurations vs link utilization



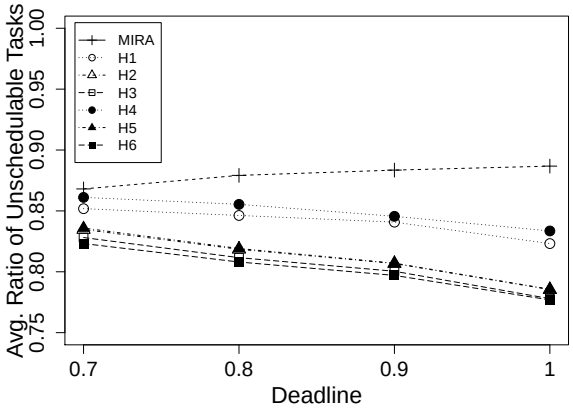
(c) Schedulable configurations vs deadline



(d) Unschedulable tasks ratio vs number of tasks



(e) Unschedulable tasks ratio vs link utilization



(f) Unschedulable tasks ratio vs deadline

Figure 8.3: Schedulability results

critical links considerably increases the number of schedulable configurations. Decrementing the number of considered lower priority tasks, using H5 and H6, only slightly increases schedulable configurations over Heuristics H2 and H3, respectively.

Figure 8.3b shows the number of schedulable configurations for each of the heuristics against link utilization. Generally, as the utilization increases, the number of schedulable configurations decreases. This is because the periodicity of the tasks gets closer to their basic link latencies, thus increasing the interference. As utilization increases from 0.5 to 0.55, however, there is a sudden increase in the schedulability. Note that this increase in utilization, decreases the period of a task to less than double its basic link latency. Hence, when weights are assigned to critical links, they are completely avoided by higher priority tasks instead of sharing them with lower priority ones. The reason is that each task requires more than half the bandwidth capacity of each link. Therefore, more lower priority tasks end up being scheduled, and in turn increasing the number of schedulable configurations. Heuristics H3 and H6 schedule more configurations than H2 and H5. Again, H1, H4, and MIRA perform closely with H4 performing better at higher utilizations. This indicates that at higher utilizations, when the deadlines are tighter, it is more beneficial to use the deadline for assigning weights versus the bandwidth capacity required by the tasks.

Figure 8.3c shows the number of schedulable configurations for each of the heuristics against the deadline. As the deadline (as a ratio of the period) increases, the number of schedulable configurations increases. This is because tasks can tolerate more interferences as the deadlines increase. Similar to the previous graphs, Heuristics H3 and H6 schedule more configurations than H2 and H5. H1, H4, and MIRA also perform closely. H1 and H4 schedule more configurations than MIRA at a higher deadline.

Ratios of Unschedulable Tasks

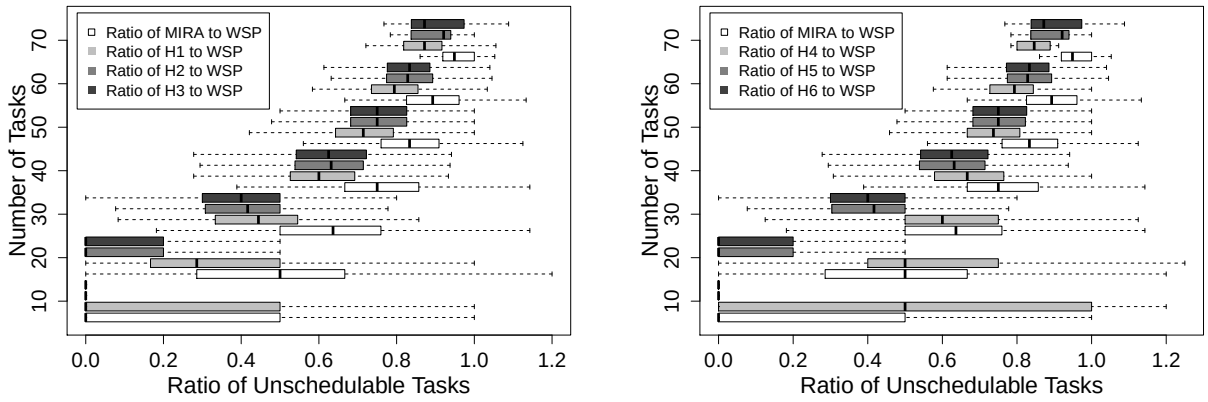
Figure 8.3d shows the average ratio of the number of unschedulable tasks of each heuristic to WSP against the number of tasks. A smaller ratio indicates less unschedulable tasks for a heuristic compared to WSP. As the number of tasks increases, the number of unschedulable tasks increases for all algorithms, hence, decreasing the ratio between them and WSP. Heuristics H3 and H6 have the least number of unschedulable tasks followed by Heuristics H2 and H5. These are followed by H1, H4, and MIRA, respectively.

Figure 8.3e shows the average ratio of unschedulable tasks against the link utilization. Generally, as the utilization increases, the number of unschedulable tasks increases due to more interference. Again, Heuristics H3 and H6 have the least number of unschedulable tasks followed by Heuristics H2 and H5. H1, H4, and MIRA perform closely with H4 performing better at higher utilizations followed by H1. This is the similar to the pattern in Figure 8.3b. This leads to the conclusion that at higher utilizations, storing the cost of interference over multiple links results in better schedulability. It also shows that using the deadline to assign weights to critical links results in more schedulable tasks as opposed to capacity requirements.

Figure 8.3f shows the ratio of unschedulable tasks against the deadline. In general, increasing the deadline gives a higher chance for tasks to meet their deadlines, thus de-

creasing the number of unschedulable tasks. The performance of the different heuristics is similar to their performance in the previous figures.

Figure 8.4a shows the ratio of the number of unschedulable tasks for MIRA and the Heuristics H1, H2, and H3 to WSP against the number of tasks for an 8×8 mesh with $U = 0.4$ and $D = 1.0$. Figure 8.4b shows the same data but for MIRA and the Heuristics H4, H5, and H6. For a given number of tasks, each box in the figure represents 1000 random test cases. The boxes represent the lower quartile, median, and upper quartile of the data, and the whiskers show the minimum and maximum observations. It is clear that H2, H3, H5, and H6 always perform better than WSP and MIRA except for some outliers. H1 has a comparable performance to MIRA for 10 tasks and performs better as the number of tasks increases. H4 performs worse than MIRA till 20 tasks and then starts to perform better as the number of tasks increases. H1 starts performing better than H2, H3, H5, and H6 as the number of tasks increases beyond 40 tasks. H4 performs better than H2, H3, H5, and H6 as the number of tasks increases beyond 50 tasks. For lower number of tasks, using shortest paths Heuristics (H2, H3, H5, and H6) can help leave these paths open for lower priority tasks thus increasing schedulability over H1 and H4. However, as number of tasks increase, these paths are occupied and avoiding links with low residual capacity (H1 and H4) becomes more important and improves schedulability. H1 performs better than H4. H5 and H6 perform slightly better than H2 and H3, respectively. The graph also shows that for a small number of cases, WSP has less unschedulable tasks compared to the other algorithms. The reason is that, in these cases, increasing the cost of critical links leads to selecting non-shortest paths. This leads in some cases to unschedulable tasks due to high interference on the chosen paths.



(a) Heuristics H1, H2, and H3 compared to WSP

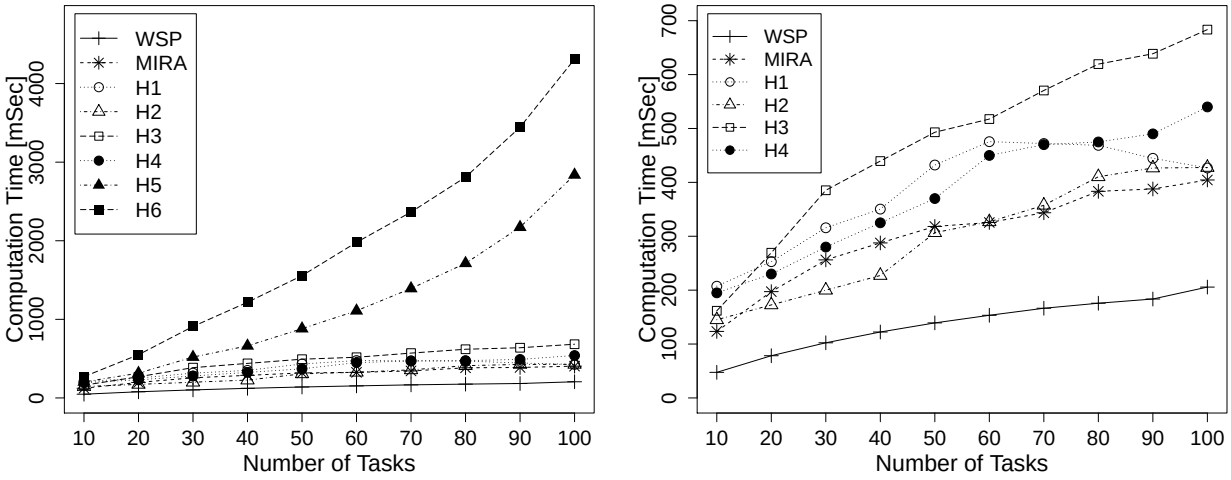
(b) Heuristics H4, H5, and H6 compared to WSP

Figure 8.4: Ratio of unschedulable tasks against number of tasks for 8×8 mesh, $U = 0.4$, $D = 1.0$

Computation Time

Figure 8.5a shows the average execution times of the algorithms against the number of tasks. Each point represents an average of 80000 test cases with 1000 for each of the

80 different configurations. It is clear how Heuristics H5 and H6 have higher execution times compared to the other algorithms. The reason is that these heuristics decrement the number of considered lower priority tasks until finding a path that allows the schedulability of a higher priority task. This leads to a higher execution time complexity as we indicated earlier.



(a) All heuristics

(b) WSP, MIRA, H1, H2, H3, and H4

Figure 8.5: Average computation times for the different algorithms against number of tasks.

Figure 8.5b shows a close-up on the average execution times of the Heuristics H1, H2, H3, and H4 against the number of tasks. WSP takes the least amount of execution time, and H2 closely follows MIRA. H1, H4, and H3, however, have higher execution times. The reason is that H2 only adds costs to lower priority tasks with one shortest path, thus doing less computations than H1, H4, and H3. H3 has the highest computation time because it updates the cost of all edges in all shortest paths of lower priority tasks. The maximum observed computation times are 1.8, 2.2, 2.8, 2.9, 2.0, 2.2, 11.8, and 17.1 seconds for WSP, MIRA, H1, H2, H3, H4, H5, and H6, respectively.

8.3.1 Summary of Experimental Results

Table 8.2 summarizes the schedulability results for all 800000 tests that were run. The table shows the geometric mean of the percentage improvement in schedulability of each heuristic over both WSP and MIRA. The 95% confidence intervals and the standard error of mean are shown as well. We also use the Wilcoxon matched pairs test to measure the significance of the improvement in schedulability of all heuristics. The p-value for the schedulability results of each heuristic is less than 2.2×10^{-16}, thus showing the statistical significance of the results.

In summary, for all $800,000$ tests, we observe an average improvement in schedulability of 10.7%, 11.5%, 12.7%, 11.0%, 11.7%, and 12.8% over WSP for H1, H2, H3, H4, H5, and

Table 8.2: Summary of experimentation results

Heuristic	Improvement over WSP			Improvement over MIRA		
	Mean	95% CI	SEM	Mean	95% CI	SEM
H1	10.7%	0.06211%	0.03619%	3.1%	0.05839%	0.02979%
H2	11.5%	0.07034%	0.03589%	3.8%	0.05792%	0.02955%
H3	12.7%	0.07132%	0.03639%	4.9%	0.06121%	0.03123%
H4	11.0%	0.06115%	0.03120%	3.3%	0.06860%	0.03500%
H5	11.7%	0.07051%	0.03598%	4.0%	0.05797%	0.02958%
H6	12.8%	0.07211%	0.03679%	5.0%	0.06095%	0.03110%

H6, respectively. The average improvement over MIRA is 3.1%, 3.8%, 4.9%, 3.3%, 4.0%, and 5.0% for H1, H2, H3, H4, H5, and H6, respectively.

On average, H1, H3, and H4 have higher computation times than MIRA by 27.1%, 57.8%, and 29.8%, respectively. H5 and H6 have higher average computation times than MIRA by 3.7 and 6.5 times, respectively. While, H2 uses, on average, less computation time compared to MIRA by 0.9%. The computation times of the first four heuristics are close to that of MIRA.

An interpretation of the experimentation results guides the choice of the heuristics to use for path selection. For smaller number of tasks in the NoC, Heuristics H2, H3, H5, and H6 schedule more tasks and configurations compared to H1 and H4. For larger number of tasks beyond 40 for an 8×8 mesh size and beyond 20 for a 4×4 mesh size, Heuristics H1 and H4 give better schedulability results.

The utilization is the factor that differentiates H1 and H4. H1 schedules more tasks at low utilizations. While H4 improves over H1 for higher utilizations beyond 0.6.

H5 and H6 have much higher computation times than their lower complexity counterparts: H2 and H3. They, however, give better schedulability results.

H3 has a slightly better performance than H2 but with almost double the execution time. The same relation holds between H6 and H5, respectively.

8.4 Set-top Box Application

We illustrate the usability of the proposed path selection algorithm on a dual input channel set-top box application. While this is a soft real-time application, we still use it mainly because it's a well-established application [77] with a significant amount of communication between its various tasks.

We use the proposed PSA to select paths for the different tasks in the dual input channel set-top box application. This application is an example of common digital video recorders that allow recording and watching two different video streams. It can record an input video stream to the storage device, and at the same time enable watching another input video stream which can either be pre-recorded or from another input channel. Hence,

in this case study, we implement the MPEG-2 encoding of the input video stream that is being recorded, and the MPEG-2 decoding of the input video stream being watched from disk on a multi-core real-time priority-aware NoC.

We represent the encoding and decoding throughput requirements as timing constraints (latency requirements). Note that the results of this case study (Table 8.4) still hold for a hard real-time application with the same task/communication characterization. We implement a cycle-accurate simulator of the priority-aware NoC in SystemC and use it to assess the applicability of PSA and its impact on interferences and schedulability. Our simulator models the cores, the interconnect, and the priority-aware routers.

Figure 8.6 shows the block diagram and the mapping of the set-top box application (assuming a 4×4 mesh priority-aware NoC). We indicate the mapping on the block as the number, e.g., the *Motion Estimation* block is mapped onto core 14. We annotate this diagram to show the encoder that writes to the hard disk, and the decoder that outputs to the display. The encoder consists of the tasks: *Source*, *Motion Estimation*, *DCT Estimation*, *Transform*, *Quantize*, *VLE*, *iQuantize*, *iTransform*, and *Disk*. Except for *Quantize* and *VLE* which communicate on the same core, all other communication takes place across the NoC as shown in Figure 8.6. The decoder is composed of: *Source*, *VLD*, *iQuantize*, *iDCT*, and *Motion Compensation*. The encoding and the decoding run in parallel. The first column of Table 8.3 names the communication tasks which require path selection in the NoC. We also add source and destination tasks for a graphical user interface visualization, represented as task *Source-Screen* in Table 8.3.

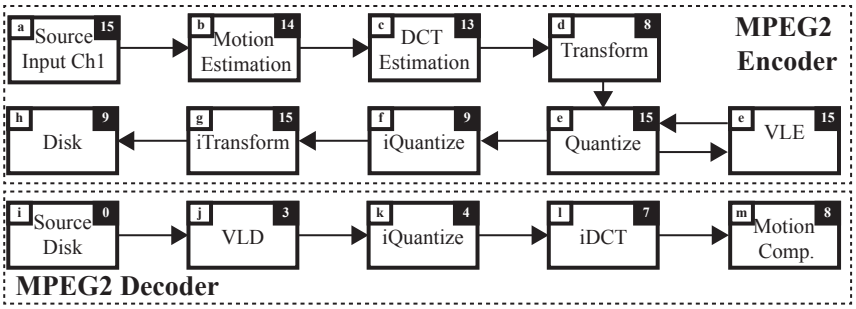


Figure 8.6: Set-top box block diagram and mapping

We apply WSP, MIRA, and PSA (with each of its six heuristics) to the set-top box application. The inputs to all algorithms are the same: (1) the mapping of the tasks to cores as shown in Figure 8.6 and (2) the characteristics of the communication tasks. The algorithms are required to select paths for the communication tasks. Table 8.3 shows the characteristics and the mapping of the different tasks in the application. Each communication task has a source node V_s and a destination node V_d which correspond to the mapping of the tasks in Figure 8.6. A task has a priority P (a smaller number represents a higher priority), a basic link latency L, and a period T. For simplicity we assume each task has zero release jitter J^R and has a deadline D equal to its period T. We also assume a rate monotonic priority assignment where tasks of equal periods are assigned priorities based on their precedence relations.

Table 8.3: Data for set-top box application

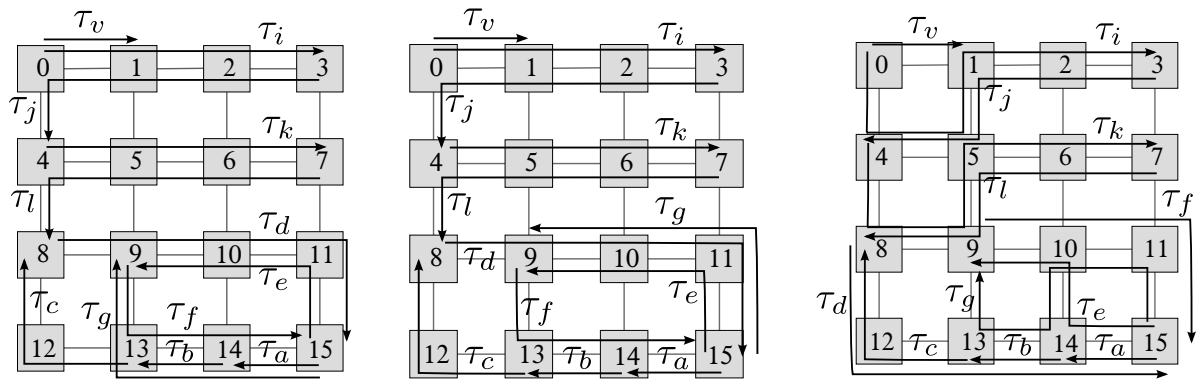
Task	Symbol	P	L	T	V_s	V_d
Source-VLD	τ_i	1	250	375	0	3
VLD-iQuant	τ_j	2	250	375	3	4
iQuant-iDCT	τ_k	3	250	375	4	7
iDCT-MC	τ_l	4	250	375	7	8
Source-ME	τ_a	5	200	550	15	14
ME-DCT Estim.	τ_b	6	200	550	14	13
DCT Estim.-Trans.	τ_c	7	200	550	13	8
Trans.-Quant/VLE	τ_d	8	250	550	8	15
Quant/VLE-iQuant	τ_e	9	250	550	15	9
iQuant-iTrans.	τ_f	10	250	550	9	15
iTrans.-Disk	τ_g	11	250	550	15	9
Source-Screen	τ_v	12	200	400	0	1

Table 8.4: Results for the set-top box case study

Task	WSP		MIRA		H1, H4		H2, H3, H5, H6	
	Path	WCL	Path	WCL	Path	WCL	Path	WCL
τ_i	[0,1,2,3]	253	[0,1,2,3]	253	[0,1,2,3]	253	[0,4,5,1,2,3]	255
τ_j	[3,7,6,5,4]	254	[3,2,1,0,4]	254	[3,2,1,0,4]	254	[3,2,1,5,4]	254
τ_k	[4,5,6,7]	253	[4,5,6,7]	253	[4,5,6,7]	253	[4,8,9,5,6,7]	255
τ_l	[7,11,10,9,8]	254	[7,6,5,4,8]	254	[7,6,5,4,8]	254	[7,6,5,9,8]	254
τ_a	[15,14]	201	[15,14]	201	[15,14]	201	[15,14]	201
τ_b	[14,13]	201	[14,13]	201	[14,13]	201	[14,13]	201
τ_c	[13,12,8]	202	[13,12,8]	202	[13,12,8]	202	[13,12,8]	202
τ_d	[8,12,13,14,15]	254	[8,9,10,11,15]	254	[8,9,10,11,15]	254	[8,12,13,14,15]	254
τ_e	[15,14,13,9]	1053	[15,11,10,9]	253	[15,11,10,9]	253	[15,14,10,9]	453
τ_f	[9,10,11,15]	253	[9,13,14,15]	253	[9,13,14,15]	253	[9,10,11,15]	253
τ_g	[15,11,10,9]	753	[15,14,13,9]	1053	[15,11,10,9]	503	[15,11,10,14,13,9]	455
τ_v	[0,1]	701	[0,1]	701	[0,1]	701	[0,1]	201

Table 8.4 shows the paths selected by the different algorithms and the worst-case latency of each task. The selected paths are also shown in Figure 8.7 where each task is given a symbol (Table 8.3). According to the discussion in Section 8.3.1, since we have less than 20 tasks for a 4×4 mesh, we expect Heuristics H2, H3, H5, and H6 to perform better than H1 and H4. Note that maybe because this is a small example, Heuristics H2, H3, H5, and H6 have the same results and Heuristics H1 and H4 choose similar paths as well. WSP has three unschedulable tasks, MIRA has two, H1 and H4 have one, and the other four heuristics schedule all the tasks.

WSP chooses the shortest path for each of the tasks which restricts the choice of paths. For task τ_e (*Quant/VLE-iQuant*), WSP chooses the path with the most residual capacity but still suffers high interference. The same happens for task τ_g (*iTrans.-Disk*). Task τ_v (*Source-Screen)* has a single shortest path but suffers interference from task τ_i (*Source-VLD*).



(a) MIRA (b) PSA: H1 and H4 (c) PSA: H2, H3, H5, and H6

Figure 8.7: Routing of the tasks for the different algorithms

MIRA makes smarter choices by avoiding critical links to satisfy lower priority tasks. The paths selected by MIRA are shown in Figure 8.7a. For task τ_g (*iTrans.-Disk*), MIRA chooses a path with the most residual capacity, however, this is not the best choice. Although the chosen path has the most residual capacity, it interferes with two other tasks: τ_a (*Source-ME*) and τ_b (*ME-DCT Estim*). The path [15,11,10,9] would have been a smarter choice as it causes less interference, this is however an artifact of PSA. The task τ_v (*Source-Screen*) does not have much options as it suffers interference on all possible paths making it unschedulable.

Heuristics H1 and H4 fail to schedule task τ_v (*Source-Screen*) like MIRA and WSP. They, however, improve on MIRA by scheduling the task τ_g (*iTrans.-Disk*). The main reason is that PSA tries to minimize the heterogeneity of interfering tasks (Objective 3) which is what makes the task schedulable.

Heuristics H2, H3, H5, and H6 schedule all tasks. The reason for this result is their ability to avoid the paths required by lower priority tasks. While routing the highest priority task τ_i (*Source-VLD*), they avoid the path [0,1] which is the path for task τ_v (*Source-Screen*) thus making it schedulable. For the other tasks, they minimize interference and the heterogeneity of interfering tasks to maximize schedulability.

In summary, for this application, PSA scheduled more tasks compared to MIRA and WSP, and only PSA managed to schedule the whole application. WSP and MIRA failed to schedule three and two tasks, respectively. Heuristcs H1 and H4 failed to schedule one task, while the other four heuristics scheduled all the tasks of the application.

8.5 Summary

This chapter presents a path selection algorithm for routing real-time communication tasks across a priority-aware NoC. Our algorithm accommodates the dependency of latencies on traversed links, enables PSA to minimize the heterogeneity of interfering tasks, and

consequently reduces the worst-case latency by appropriately selecting paths. We develop six heuristics to account for expected paths of low priority tasks. All heuristics improve schedulability compared to WSP by at least 10%. Compared to MIRA, H1 and H4 have only an average improvement of 3.1% and 3.3%, respectively. These two heuristics perform worse than MIRA for a number of configurations. H2 and H3 improve over MIRA in schedulability by 3.8% and 4.9%, respectively. H5 and H6 improve in schedulability over MIRA by 4.0% and 5.0%, respectively, with a small improvement over H2 and H3. The computation times of the first four heuristics are comparable to MIRA's and much less than that of the optimal algorithm. The results show that SLA-based PSA utilizing H2 is a good option in terms of complexity, computation time, and schedulability for doing the path selection of tasks.

Chapter 9

Conclusion

Information extraction and performance scalability are two challenges that exist in the field of real-time systems. Deploying real-time systems to chip-multiprocessors is a solution to the performance scalability problem. This thesis presents techniques for achieving this deployment. These CMP-specific analysis techniques compute tight worst-case latencies for the deployed tasks. This enables assigning more budget to the instrumentation process. Assigning more budget to the time-aware instrumentation techniques proposed in this thesis, allows extracting more information from programs.

Researchers develop custom NoC interconnects to use as a platform for real-time applications. Although priority-aware networks have advantages over TDM NoCs, there is no complete flow that facilitates deploying real-time applications on priority-aware networks. In this thesis, we try to complete this flow by presenting an analysis for deploying hard real-time applications on priority-aware networks. This includes:

1. A stage-level analysis for computing WCLs of communication tasks in priority-aware networks.
2. An offset-based WCRT analysis for computing end-to-end WCRTs of real-time applications in priority-aware networks.
3. A buffer-space analysis for computing the buffer space require for valid WCL analysis techniques for priority-aware networks.
4. An extension of stage-level analysis to incorporate limits on buffer space and a buffer space allocation algorithm.
5. A path-selection algorithm to select paths for communication tasks that reduce network interference and increase schedulability.

Future work is required to complete the deployment flow for hard real-time applications on priority-aware NoCs. This includes:

1. A mapping algorithm to map computation tasks to the different NoC cores to reduce interference in the network and increase schedulability. It is common practice to combine both the mapping and path selection algorithms. Interference computations on the stage-level of the network using SLA might produce better results than existing algorithms.

2. A priority-assignment algorithm to assign priorities to the computation and communication tasks to increase schedulability. The proposed analyses can be extended to support priority sharing between tasks. An algorithm that assigns priorities to tasks, including shared priorities, should also take buffer space allocation into account. SLA can be used to compute WCLs on each stage based on allocated buffer space and priorities of tasks.

3. An implementation of the priority-aware network with reconfigurable routers to reallocate buffer space between virtual channels based on application requirements. The synchronization scheme used in the implementation might require modifications to the proposed analyses.

In this work, we address one of the prominent challenges in deploying hard real-time applications on priority-aware NoCs. That is, bounding the buffer space requirements in the virtual channels of the priority-aware routers. We also provide tighter worst-case latency analysis techniques compared to state-of-the-art in bounding communication latencies and response times in priority-aware networks. Tightening the worst-case latency bounds, increases the static time window between the application's worst-case response time and its deadline. This, in turn, increases the instrumentation budget available for the different computation tasks of the application. Increasing the instrumentation budget allows the time-aware instrumentation techniques, proposed in this work, to extract more information from the program.

The results from the work on deploying hard real-time applications on priority-aware NoCs are promising. This work, however, only focuses on analysis techniques for priority-aware NoCs. A key element to validating these results is experimenting on an implementation of the priority-aware network which is part of future work. The proposed analyses assume a fully-synchronous priority-aware NoC. Fully synchronous networks, however, have limitations such as processing cores operating at different and maybe adaptive clock frequencies. Also clock distribution amongst the cores is difficult, and the minimization of clock skew is complex and costly. These limitations can be addressed using new clock generation and distribution techniques, and by using asynchronous communication resources such as in globally asynchronous, locally synchronous (GALS)-type NoCs. Hence, some implementation-specific aspects might affect the applicability and practicality of the proposed deployment flow, and might require modifying the proposed techniques.

Information extraction techniques for real-time systems have to consider the time requirements of such applications. We present techniques that enable static source-code time-aware instrumentation for real-time applications. These techniques allow information extraction from real-time applications while meeting their timing constraints. These approaches to time-aware instrumentation include:

1. Program transformation techniques to increase the effectiveness of time-aware instrumentation.

2. Slack-based conditional time-aware instrumentation to enable instrumenting the worst-case path of a program.

3. A static instrumentation framework, INSTEP, for preserving the extra-functional properties of a program including time.

These time-aware instrumentation techniques make use of the instrumentation budget available to the program. Since, these techniques take the program's timing constraints into account, they cannot extract full traces from the instrumented program. However, the extracted partial traces can be composed to increase the instrumentation coverage. Apart from increasing the instrumentation coverage, partial traces are still useful. For instance, they can be used to build inductive debugging mechanisms for deployed resource and space constrained systems.

Worst-case execution time analysis is required for the proposed time-aware instrumentation techniques. It is required before the instrumentation process to compute the WCET of the different basic blocks of the program's CFG. WCET analysis is also required after the instrumentation process to ensure that the timing constraints are met. This is needed because the proposed techniques ignore certain side-effects of the instrumentation process such as changing the program's memory and cache layouts. Although, in certain cases, due to timing violations, the instrumentation process has to be repeated, the number of retries required are low.

The proposed static source-code instrumentation techniques are suited for hard real-time applications. These techniques can be applied to applications where WCET analysis is common and source-code is available for instrumentation. Other time-aware instrumentation techniques, such as time-aware dynamic binary instrumentation, can be used otherwise.

The time-aware instrumentation techniques proposed in this work respect the program's timing constraints. The information they extract is limited by the instrumentation budget available to the program. Knowledge of the underlying platform, on which the real-time system runs, can result in assigning higher instrumentation budgets to programs. In this work, we present techniques for using CMPs, in particular priority-aware NoCs, as a platform for deploying real-time systems. These techniques enable computing tight worst-case latencies and response times for the real-time application. Hence, they allow for assigning higher instrumentation budgets to applications, which enables extracting more information using the time-aware instrumentation of real-time applications.

References

[1] EasyWeb. http://www.keil.com/download/docs/295.asp.

[2] RapiTime. http://www.rapitasystems.com/products/RapiTime.

[3] SNU Real-Time Benchmarks. http://www.cprover.org/goto-cc/examples/snu.html.

[4] Mohammad Abdullah Al Faruque and Jörg Henkel. Minimizing Virtual Channel Buffer for Routers in On-Chip Communication Architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, New York, NY, USA, 2008. ACM.

[5] Pansy Arafa, Hany Kashif, and Sebastian Fischmeister. DIME: Time-aware Dynamic Binary Instrumentation Using Rate-based Resource Allocation. In *Proceedings of the 13th International Conference on Embedded Software (EMSOFT)*, Montreal, Canada, September 2013.

[6] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Comput. Archit. Lett.*, 6, 2007.

[7] David F. Bacon and Seth Copen Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *SIGPLAN Not.*, 26, 1991.

[8] J. Baginski and H. Seiffert. Interaktives Trace- und Debugging- System ALGOL KIEL X 8. In *3. Fachtagung über Programmiersprachen, Gesellschaft für Informatik*, London, UK, 1974. Springer-Verlag.

[9] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16, July 1994.

[10] Dan H. Barnes and Larry L. Wear. Instruction Tracing via Microprogramming. In *Conference record of the 7th Annual Workshop on Microprogramming (MICRO)*, New York, NY, USA, 1974. ACM.

[11] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, Berkeley, CA, USA, 2005. USENIX Association.

[12] L. Benini and G. De Micheli. Networks on Chip: A New Paradigm for Systems on Chip Design. In *Proceedings of the conference on Design, Automation and Test in Europe*, 2002.

[13] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1), January 2002.

[14] Thierry Benoist, Bertrand Estellon, FrEdEric Gardi, Romain Megel, and Karim Nouioua. LocalSolver 1.x: A Black-Box Local-Search Solver for 0-1 Programming. *4OR: A Quarterly Journal of Operations Research*, 9, 2011.

[15] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a Good Bug Report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, New York, NY, USA, 2008. ACM.

[16] Luca Bisti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Numerical analysis of worst-case end-to-end delay bounds in fifo tandem networks. *Real-Time Systems*, 48(5), 2012.

[17] Tobias Bjerregaard and Jens Sparso. A Scheduling Discipline for Latency and Bandwidth Guarantees in Asynchronous Network-on-Chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS Architecture and Design Process for Network on Chip. *Journal of System Architecture*, 50, 2004.

[19] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, 2003.

[20] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. *SIGPLAN Not.*, 47(7), Mar. 2012.

[21] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4), Nov. 2000.

[22] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric wcet calculation. *J. Syst. Archit.*, 57(6), June 2011.

[23] T. A. Cargill and B. N. Locanthi. Cheap Hardware Support for Software Debugging and Profiling. *SIGARCH Comput. Archit. News*, 15, 1987.

[24] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified wcet analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4s), April 2014.

[26] Alvin Cheung and Samuel Madden. Performance Profiling with EndoScope, an Acquisitional Software Monitoring Framework. *Proceedings VLDB Endowment*, 1, August 2008.

[27] Martijn Coenen, Srinivasan Murali, Andrei Ruadulescu, Kees Goossens, and Giovanni De Micheli. A Buffer-Sizing Algorithm for Networks on Chip Using TDMA and Credit-based End-to-End Flow Control. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New York, NY, USA, 2006. ACM.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[29] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36, 1987.

[30] William J. Dally and Brian Towles. Route packets, not wires: On-chip inteconnection networks. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, New York, NY, USA, 2001. ACM.

[31] W.J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), 1992.

[32] P.S. Dodd and C.V. Ravishankar. *Monitoring and Debugging Distributed Real-Time Programs*, chapter Monitoring and debugging distributed real-time programs. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[33] J. Duato, A. Robles, F. Silla, and R. Beivide. A comparison of router architectures for virtual cut-through and wormhole switching in a now environment. *J. Parallel Distrib. Comput.*, 61(2), February 2001.

[34] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[35] M. B. Dwyer, M. Diep, and S. Elbaum. Reducing the Cost of Path Property Monitoring Through Sampling. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. Vulcan: Binary Transformation in a Distributed Environment. Technical report, 2001.

[37] C Ferdinand and Reinhold Heckmann. aiT: Worst Case Execution Time Prediction by Static Program Analysis. *International Federation For Information Processing (IFIP)*, 156, 2004.

[38] Gustavo B. Figueiredo, Nelson L. S. da Fonseca, and José A. S. Monteiro. A Minimum Interference Routing Algorithm with Reduced Computational Complexity. *Comput. Netw.*, 50, 2006.

[39] S. Fischmeister and P. Lam. Time-Aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 2010.

[40] Sebastian Fischmeister and Yanmeng Ba. Sampling-based Program Execution Monitoring. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2010.

[41] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *SIGPLAN Not.*, 24, 1988.

[42] P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. Softw. Eng.*, 14, October 1988.

[43] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technology Planning Report, 2002.

[44] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test*, 22(5), 2005.

[45] Giovani Gracioli and Sebastian Fischmeister. Tracing Interrupts in Embedded Software. *SIGPLAN Not.*, 44, 2009.

[46] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. GProf: A Call Graph Execution Profiler. *SIGPLAN Not.*, 39, April 2004.

[47] Roch A. Guerin, Ariel Orda, and Douglas Williams. QoS Routing Mechanisms and OSPF Extensions. In *Proceedings of IEEE GLOBECOM*, 1996.

[48] Andreas Hansson, Mahesh Subburaman, and Kees Goossens. aelite: A flit-synchronous network on chip with composable and predictable services. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 250–255, April 2009.

[49] Damien Hardy and Isabelle Puaut. Wcet analysis of instruction cache hierarchies. *Journal of Systems Architecture - Embedded Systems Design*, 57(7), 2011.

[50] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. *SIGARCH Comput. Archit. News*, 36, 2008.

[51] Jingcao Hu and R. Marculescu. Application-Specific Buffer Space Allocation for Networks-on-Chip Router Design. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2004.

[52] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc. of the 3rd Conf. on USENIX Windows NT Symp. (WINSYM)*, 1999.

[53] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[54] Mathur Idika. A Survey of Malware Detection Techniques. February 2007.

[55] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *Proceedings of 19th IEEE Euromicro Conference on Real-Time Systems*, 2007.

[56] P. Jayachandran and T. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *Proceedings of 21st IEEE Euromicro Conference on Real-Time Systems*, 2009.

[57] Praveen Jayachandran and Tarek Abdelzaher. A Delay Composition Theorem for Real-Time Pipelines. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A.S. Ward. System Monitoring with Metric-Correlation Models: Problems and Solutions. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC)*, New York, NY, USA, 2009. ACM.

[59] Dilip D. Kandlur and Kang G. Shin. Traffic Routing for Multicomputer Networks with Virtual Cut-Through Capability. *IEEE Trans. Comput.*, 41, 1992.

[60] Dilip D. Kandlur, Kang G. Shin, and Domenico Ferrari. Real-Time Communication in Multihop Networks. *IEEE Trans. Parallel Distrib. Syst.*, 1994.

[61] Koushik Kar, Murali Kodialam, and T. V. Lakshman. Minimum Interference Routing of Bandwidth Guaranteed Tunnels with MPLS Traffic Engineering Application. *IEEE Journal on Selected Areas in Communications*, 2000.

[62] Hany Kashif, Pansy Arafa, and Sebastian Fischmeister. INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Taipei, Taiwan, August 2013.

[63] Hany Kashif and Sebastian Fischmeister. Program Transformation for Time-aware Instrumentation. In *Proceedings of the 17th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, Krakow, Poland, September 2012.

[64] Hany Kashif, Sina Gholamian, and Hiren Patel. SLA: A Stage-level Latency Analysis for Real-time Communication in a Pipelined Resource Model. *IEEE Transactions on Computers*, 64(4), April 2014.

[65] Hany Kashif, Sina Gholamian, Rodolfo Pellizzoni, Hiren D. Patel, and Sebastian Fischmeister. ORTAP:An Offset-based Response Time Analysis for a Pipelined Communication Resource Model. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, USA, April 2013.

[66] Hany Kashif and Hiren Patel. Bounding Buffer Space Requirements for Real-Time Priority-Aware Networks. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, SunTec, Singapore, January 2014.

[67] Hany Kashif, Hiren D. Patel, and Sebastian Fischmeister. Using Link-level Latency Analysis for Path Selection for Real-time Communication on NoCs. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, Sydney, Australia, February 2012.

[68] Hany Kashif, Johnson Thomas, Hiren Patel, and Sebastian Fischmeister. Static Slack-Based Instrumentation of Programs. In *Proceedings of the 20th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, Luxembourg, September 2015.

[69] Irvin R. Katz and John R. Anderson. Debugging: An Analysis of Bug-Location Strategies. *Hum.-Comput. Interact.*, 3, 1987.

[70] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Form. Methods Syst. Des.*, 24(2), 2004.

[71] A Ko and B Myers. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing*, 16(1-2), 2005.

[72] A.S. Kumar, M.P. Kumar, S. Murali, V. Kamakoti, L. Benini, and G. De Micheli. A Simulation Based Buffer Sizing Algorithm for Network on Chips. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2011.

[73] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '02, 2002.

[74] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN Not.*, 30, 1995.

[75] J.R. Larus. Efficient Program Tracing. *Computer*, 26(5), 1993.

[76] A.S. Lee and N.W. Bergmann. On-chip Communication Architectures for Reconfigurable System-on-Chip. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, 2003.

[77] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3), May 2008.

[78] Sheayun Lee, Insik Shin, Woonseok Kim, Insup Lee, and Sang Lyul Min. A Design Framework for Real-time Embedded Systems with Code Size and Energy Constraints. *ACM Trans. Embed. Comput. Syst.*, 7(2), Jan. 2008.

[79] Sunggu Lee and Jong Kim. Path Selection for Message Passing in a Circuit-Switched Multicomputer. *J. Parallel Distrib. Comput.*, 35, 1996.

[80] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of 11th IEEE Real-Time Systems Symposium*, 1990.

[81] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004.

[82] Y.T.-S. Li, S. Malik, and A Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, Dec 1995.

[83] Ben Liblit, Alex Aiken, Mayur Naik, and Alice X. Zheng. Scalable Statistical Bug Isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2005.

[84] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[85] Daniel Lohmann, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. Functional and Non-functional Properties in a Family of Embedded Operating Systems. In *Proceedings of the 10th IEEE International Workshop On Object-Oriented Real-Time Dependable Systems (WORDS)*. IEEE Computer Society, 2005.

[86] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, Washington, DC, USA, 2009. IEEE Computer Society.

[87] Zhonghai Lu and Axel Jantsch. TDM Virtual-Circuit Configuration for Network-on-Chip. *IEEE Transactions on Very Large Scale Integrated Systems*, 16, August 2008.

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[89] J. Maki-Turja and M. Nolin. Fast and Tight Response-Times for Tasks with Offsets. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[90] Sorin Manolache, Petru Eles, and Zebo Peng. Buffer Space Optimisation with Communication Synthesis and Traffic Shaping for NoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[91] McAfee. W32/mydoom@mm. http://vil-origin.nai.com/vil, 2004.

[92] J. M. Mellor-Crummey and T. J. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, New York, NY, USA, 1989. ACM.

[93] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Washington, DC, USA, 2004. IEEE Computer Society.

[94] MIRA Ltd. MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems, October 2004.

[95] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. *SIGARCH Comput. Archit. News*, 36, 2008.

[96] Linda J. Moore and Angelica R. Moya. Non-Intrusive Debug Technique for Embedded Programming. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, Washington, DC, USA, 2003. IEEE Computer Society.

[97] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. We have it Easy, but do we have it Right? *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[98] Mayur Naik and Jens Palsberg. Compiling with Code-size Constraints. *ACM Trans. Embed. Comput. Syst.*, 3(1), Feb. 2004.

[99] Kyungwan Nam, Sunggu Lee, and Jong Kim. Path Selection for Real-Time Communication in Wormhole Networks. *International Journal of High Speed Computing*, 1999.

[100] Samaneh Navabpour, Borzoo Bonakdarpour, and Sebastian Fischmeister. Optimal Instrumentation of Data-Flow in Concurrent Data Structures. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS)*, Berlin, Heidelberg, 2011. Springer-Verlag.

[101] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[102] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[103] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6), Jun. 2007.

[104] Robert H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. *SIGPLAN Not.*, 28, 1993.

[105] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2), February 1993.

[106] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 26(2), 1993.

[107] William Omre. Debug and Trace for Multicore SoCs. Technical report, ARM, 2008.

[108] J. C. Palencia and M. González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Washington, DC, USA, 1998. IEEE Computer Society.

[109] J. C. Palencia and M. González Harbour. Response Time Analysis of EDF Distributed Real-time Systems. *Journal of Embedded Computing*, 1(2), April 2005.

[110] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case. *IEEE/ACM Trans. Netw.*, 1993.

[111] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Trans. Netw.*, 1994.

[112] L.T.X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *Proceedings of IEEE Real-Time Systems Symposium*, 2008.

[113] Peter Phillips. Enhanced Debugging with Traces. *Queue*, 8, March 2010.

[114] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable Omniscient Debugging. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, New York, NY, USA, 2007. ACM.

[115] Antonio Pullini, Federico Angiolini, Davide Bertozzi, and Luca Benini. Fault tolerance overhead in network-on-chip flow control schemes. In *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design*, SBCCI '05, New York, NY, USA, 2005. ACM.

[116] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1), Jan 2005.

[117] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2), March 1999.

[118] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. *IEEE Proceedings of Computers and Digital Techniques*, 150(5), September 2003.

[119] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Berkeley, CA, USA, 1997. USENIX Association.

[120] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.*, 17, 1999.

[121] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/Replay for Non-Deterministic Program Executions. *Commun. ACM*, 46, September 2003.

[122] Raul Santelices and Mary Jean Harrold. Efficiently Monitoring Data-Flow Test Coverage. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, New York, NY, USA, 2007. ACM.

[123] Mauricio Serrano and Xiaotong Zhuang. Building Approximate Calling Context from Partial Call Traces. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, USA, 2009. IEEE Computer Society.

[124] Z. Shi. *Real-Time Communication Services for Networks on Chip*. PhD thesis, The University of York, UK, 2009.

[125] Zheng Shi and Alan Burns. Priority Assignment for Real-Time Wormhole Communication in On-Chip Networks. In *Proceedings of the 2008 Real-Time Systems Symposium (RTSS)*, Washington, DC, USA, 2008. IEEE Computer Society.

[126] Zheng Shi and Alan Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *Proceedings of the 2nd ACM/IEEE International Symposium on Networks-on-Chip*, Washington, USA, 2008.

[127] Zheng Shi and Alan Burns. Real-Time Communication Analysis with a Priority Share Policy in On-Chip Networks. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2009. IEEE Computer Society.

[128] Zheng Shi and Alan Burns. Schedulability Analysis and Task Mapping for Real-Time on-Chip Communication. *Real-Time Syst.*, 46, December 2010.

[129] Kang G. Shin, Chih-Che Chou, and Seok-Kyu Kweon. Distributed Route Selection for Establishing Real-Time Channels. *IEEE Trans. Parallel Distrib. Syst.*, 11, 2000.

[130] Shridhar B. Shukla and Dharma P. Agrawal. Scheduling Pipelined Communication in Distributed Memory Multiprocessors for Real-Time Applications. *SIGARCH Comput. Archit. News*, 19, 1991.

[131] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Common Sense Computing (Episode 4): Debugging. *Computer Science Education*, 18(2), 2008.

[132] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC)*, 2010.

[133] Jens Sparso. Design of networks-on-chip for real-time multi-processor systems-on-chip. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD)*, pages 1–5, June 2012.

[134] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.*, 39, 1994.

[135] M. Staats, M.W. Whalen, and M.P.E. Heimdahl. Better testing through oracle selection: (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, May 2011.

[136] Radu Stefan, Anca Molnos, Angelo Ambrose, and Kees Goossens. A tdm noc supporting qos, multicast, and fast connection set-up. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1283–1288, March 2012.

[137] D. Sundmark and H. Thane. Pinpointing Interrupts in Embedded Real-Time Systems Using Context Checksums. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2008.

[138] Daniel Sundmark, Henrik Thane, Joel Huselius, and Anders Pettersson. Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies. In *Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging (AADEBUG)*, 2003.

[139] Subhash Suri, Marcel Waldvogel, and Priyank Ramesh Warkhede. Profile-Based Routing: A New Framework for MPLS Traffic Engineering. In *Proceedings of International Workshop on Quality of Future Internet Services*, London, UK, 2001. Springer-Verlag.

[140] Sami Taktak, Jean-Lou Desbarbieux, and Emmanuelle Encrenaz. A tool for automatic detection of deadlock in wormhole networks on chip. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1), February 2008.

[141] Ariel Tamches and Barton P. Miller. Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.

[142] Xinan Tang, J. Wang, Kevin B. Theobald, and Guang R. Gao. Thread Partitioning and Scheduling based on Cost Model. In *Proceedings of the 9th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, New York, NY, USA, 1997. ACM.

[143] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Department of Computer Science and Electronics, Mälardalens University, 2000.

[144] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[145] Johnson Thomas, Sebastian Fischmeister, and Deepak Kumar. Lowering Overhead in Sampling-based Execution Monitoring and Tracing. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, USA, 2011.

[146] Ken Tindell and John Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming*, 1994.

[147] TrendMicro. Bkdr.surila.g (w32/ratos). http://www.trendmicro.com/vinfo/virusencyclo, 2004.

[148] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Non-Interference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Trans. Softw. Eng.*, 16, 1990.

[149] Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative Synthesis for Concurrent Programs. In *Proceedings*

of the 23rd International Conference on Computer Aided Verification (CAV), Berlin, Heidelberg, 2011. Springer-Verlag.

[150] Manel Velasco, Pau Martí, Josep M. Fuertes, Camilo Lozoya, and Scott A. Brandt. Experimental evaluation of slack management in real-time control systems: Coordinated vs. self-triggered approach. *J. Syst. Archit.*, 56(1), January 2010.

[151] Daniel Wiklund and Dake Liu. SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, Washington, DC, USA, 2003. IEEE Computer Society.

[152] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3), 2008.

[153] P.T. Wolkotte, G.J.M. Smit, G.K. Rauwerda, and L.T. Smit. An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[154] Jean yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[155] Cesar Albenes Zeferino and Altamiro Amadeu Susin. Socin: A parametric and scalable network-on-chip. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, SBCCI '03, Washington, DC, USA, 2003. IEEE Computer Society.

[156] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. *SIGOPS Oper. Syst. Rev.*, 31, 1997.

[157] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical Debugging: Simultaneous Identification of Multiple Bugs. In *Proceedings of the 23rd International Conference on Machine learning (ICML)*. ACM, 2006.