Integrating Security Mechanisms in Hard Real-Time Systems

by

Neda Paryab

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Statement of Contributions

The work presented in this thesis is partially based upon and extends the work presented in the following published paper:

[36]: Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B. Bobba. ”A generalized model for preventing information leakage in hard real-time systems.” In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, pp. 271-282.

Due to the relation with the published work, parts of this thesis contain significant material from [36], including Chapters 3-5 and Sections 2.2 and 8.1.

We would like to thank all the co-authors for their precious help in completing this research. In particular, we would like to thank Sibin Mohan and Rakesh B. Bobba for their work on the security model, Stanley Bak for contributing the exact FT bound and its implementation, Man-Ki Yoon for implementing the graph FT bound, and Rodolfo Pellizzoni for his help on the schedulability analysis.

Abstract

Traditionally Real-Time Systems (RTSs) and security have been considered as separate domains. This is mostly because traditional systems employed isolated customized components, while modern systems tend to be highly interconnected and rely on open components and protocols. A wave of recent attacks on real-time systems have forced both practitioners and researchers to consider security as an essential system requirement. To propose a first step towards integrating security mechanisms in real-time systems, we focus on the problem of information leakage through shared physical resources such as cache memory. Regular security mechanisms tend to be computationally intensive, and using them as a separate protection component in hard RTSs can affect the schedulability of the system. Hence, in this work we propose two mechanisms to prevent information leakage, analyze their impact on task schedulability and study how to optimize the system configuration to minimize overhead. A new generalized security model is introduced to model the relevant security requirements. We implemented all proposed techniques on an available real-time operating systems, and evaluated their performance based on both a realistic case study of a UAV system as well as synthetic applications.

Acknowledgements

I would like that thank all my co-authors who provided insight and expertise that greatly assisted this research; this work could not have been completed without their help.

I would like to deeply thank my supervisor, Prof. Pellizzoni, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions so promptly. I would also like to thank all the members of Real-Time Systems Lab, Jean-Christophe, Thomas, Yassir, and Peiyi who helped my technical issues. In particular I would like to thank Summit, the Lab Manager, who gave me effective suggestions.

I must express my gratitude to my mother and father who experienced all of the ups and downs of my research, and my very special and speechless thanks to my only sister, Nasim, who was always beside me in all circumstances and push me through finding the best positive experience out of them.

Dedication

This is dedicated to my beloved Parents, Nasrin and Khalil.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In the past, security was not considered a major concern in real-time embedded systems. The necessity of security in real-time systems was not recognized mainly because, until recently, real-time systems employed customized system components and were physically isolated from one another. On the other hand, new systems often rely on commercial off-the-shelf (COTS) components and are highly interconnected with each other, possibly through unsecured networks such as the Internet. Moreover, attackers are increasing in sophistication and are able to bridge air gaps in industrial control systems [18], perform malicious code injection into the telematics units of modern automobiles [15, 27], and demonstrate potential vulnerabilities in avionics systems [45] and attacks on UAVs [39].

Given the additional constraints on the operation of real-time systems, vulnerabilities and the impact of exploiting them differ from those of traditional enterprise systems. A successful attack could destabilize the system resulting in harm to humans, the environment and/or the system. Attacks could range from the leakage of critical data [41] to hostile actions such as an adversary taking control of the system, say due to the lack of authentication [15, 27, 45]. Many of the aforementioned attacks succeeded because such systems were not designed with security in mind. Hence, we argue that security requirements should be planned in the design phase of real-time systems.

One advantage of considering real-time and security requirements together at a system-level is to end up with extended flexibility and adaptability to add a new subsystem. But the discussed advantage might come with a significant cost. In fact, security mechanisms, such as encryption algorithms, are often computationally intensive, and adding them to an existing real-time system might compromise the ability of the system to meet its timing requirements (task deadline). In this sense, the real-time task scheduler has a crucial role to coordinate shared resources between various components of the system and also has the significant goal to guarantee that timing constraints and other real-time requirements are safely met. Hence, security mechanisms should be integrated with the real-time scheduler to guarantee that the system remains safe.

1.1 Problem Statement

Given the significant need for protecting real-time embedded systems against security vulnerabilities, this thesis provides a first step towards integrating security mechanisms in real-time scheduling by looking at a specific problem: avoiding information leakage through shared physical resources such as cache memory. It is fairly well understood that the use of shared resources can lead to information leakage without the need for explicit communication between tasks [26, 37, 53]. A new general security model is introduced to model the corresponding security requirements between tasks. Our research targets hard real-time tasks, where missing deadlines is not acceptable, since it can lead to catastrophic effects. Hence, the focus is on mechanisms that can be proven safe through schedulability analysis.

1.2 Contributions

In particular, we discuss two timing-analyzable mechanisms for preventing information leakage: flushing the state of the resource upon a context switch between protected tasks, and statically partitioning the resource among tasks. We introduce a model that allows us to combine the two mechanisms while still deriving predictable bounds on timing interference. Furthermore, we provide a heuristic optimization framework that allows the designer to trade-off the effects of the two mechanisms while searching for a near to optimal design. We demonstrate our solution using extensive simulations based on the case study of a UAV platform.

In more details, this research′s contributions are as follows:

- providing a general security model, which describes confidentiality relations between any pair of tasks. The relation determines whether a task requires to have confidential protection against another task;
- introducing two separate mechanisms to protect shared physical resources towards information leakage, and discussing their effects on real-time task scheduling;

- introducing an optimized solution, in terms of overhead for preserving confidentiality, that determines: (1) the lowest number of resource state resets, and (2) in addition to the previous item, the best allocation of shared resource partitions to tasks;
- implementing each of the introduced techniques within an existing real-time operating systems, and demonstrating them with a realistic fully-functional application;
- evaluating the demonstrated techniques in terms of their timing and performance using both the developed application and a large set of synthetic tasks sets.

1.3 Thesis Organization

Overall structure of the thesis can be described as following, Chapter 2 introduces readers with backgrounds on the concerned security problem and FreeRTOS real-time kernel, besides overviewing important related works. Chapter 3 describes the security model.Also, the chapter introduces the developed case study. Chapter 4 describes two main contributed security mechanisms to reduce the potential for security attacks to happen, as well as studying the impact of integrating those mechanisms with real-time task scheduler. Chapter 5 details the scheduling analysis beneath the experimentations and implementation. Chapter 6 describes the heuristic to finding optimized solutions for resource allocation. Chapter 7 comprehensively describes the implemented platform by introducing its components, and modifications on FreeRTOS task scheduler. Finally, evaluations and experimental results are described in Chapter 8, while Chapter 9 provides concluding remarks and discusses some required complementary efforts.

Chapter 2

Background and Literature Review

In this chapter, we introduce readers to relevant backgrounds on underlying concepts used throughout the thesis. Afterwards, we discuss significant related works in the area of security in real-time systems, discussing the similarities and differences in our approach.

2.1 Background

In the following three sections, we want to give basic background on: (1) some security problems in embedded systems, and more specifically on timing channel. In fact, it is worth explaining what is the applicable scope of our security model, (2) FreeRTOS in terms of its code architecture and task scheduling functionality. In order to demonstrate the security model within a case study (details in Chapters 3 and 7), we picked FreeRTOS as our beneath software platform. Apart from its light-wight kernel code, it gives us freedom to have our modifications in the kernel codes with the least struggling with dependency of codes on the system calls, etc. Therefore, it is very necessary to provide basic background on FreeRTOS before explaining how we applied the security model′s techniques on it (details in Chapter 7) (3) Genetic Algorithm (GA) by introducing its basic components and terminology, since we have used GA method afterwards to find a good performance solution. In addition, the terminology of GA will be referred frequently in the subsequent chapters. Thus it would be more clarifying for the readers to get introduced to the basic background beforehand.

2.1.1 Security Problem

Traditional embedded systems have not recognized security as a required matter. In fact they were less prone to having potential vulnerabilities because of (a) private protocols, (b) physically isolated subsystems, (c) executing on dedicated hardware.

Given the next-generation embedded Real-Time Systems (RTSs), using Commercial-Off-The-Shelf (COTS) technology, they reduce power consumption and are proceeded to be multi-functional systems. Simultaneously, modern RTSs established complicated vulnerabilities. Some of the security threats are against confidentiality, integrity and availability, that could be defended using encryption, message authentication and replication. Arguably, security has been mostly considered as a separate matter of fact from real-time requirements, resulted in having computationally intensive security techniques that swiftly reduce RTSs′ limited resources. On the other hand, the lack of design-time considerations for security issues, initiates RTSs to be unprotected against emergent and complicated security attacks such as timing channel.

Accordingly, real-time systems are threaten by various attacks, such as leakage of critical data [41], and lack of authentication [15], [27], [39]. Consequently, there are some documented examples of specific real-time industrial attacks, such as malicious code injection on modern automobile systems [15], [27]. In [15], authors expose the vulnerability of external attacks in contemporary automobiles. After synthesizing their threat model to demonstrate how possibly I/O channels can be used to convey malicious input, they have assessed the capabilities that can exploited by an attacker, and insightfully to have secured automotive platforms. Some other specific attacks exist on avionics systems [39], [45]. For example, [45] practically demonstrated that an attack on a aircraft can be done remotely, even with having no physical access to the target aircraft. The authors of [45] have studied an attack formation as a sequence of: discovery, information gathering, exploitation and post-exploitation. In order, discovery phase involves Automatic Dependent Surveillance-Broadcast (ADS-B), which is used for locating and plotting targets. Targets can be local data, including Software Defined Radio (SDR) data. Both information gathering and exploitation are fulfilled via Aircraft Communications Addressing and Reporting System (ACARS), which is about transmitting digital datalink between aircraft and ground stations in order to plot targets. The authors overviewed on some examples such as using vulnerabilities of ACARS to exploit Flight Management System (FMS).

Stuxnet worm is likely the first sophisticated malware attack that targeted some industrial nuclear embedded process control systems [18]. It exploited security vulnerabilities to get access to Siemens controllers and download itself, exploiting intimate knowledge of those embedded systems. By imposing an enormous cost on the victimized industrial system, Stuxnet took the fundamental message that there is a crucial need for improving security protection in embedded systems, especially in critical infrastructures.

There exists other complicated attacks such as covert timing channel, which is briefly about having information leakage even without having explicit communications (e.g., [26], [37]).

Since our focus of this research is more about avoiding covert channels, we want to introduce its basics and classifications in more details. In [41], types of covert channels, illustrated in Figure 2.1, have been well classified as: non-deducible, positive-deducible, negative-deducible, and partially-deducible.

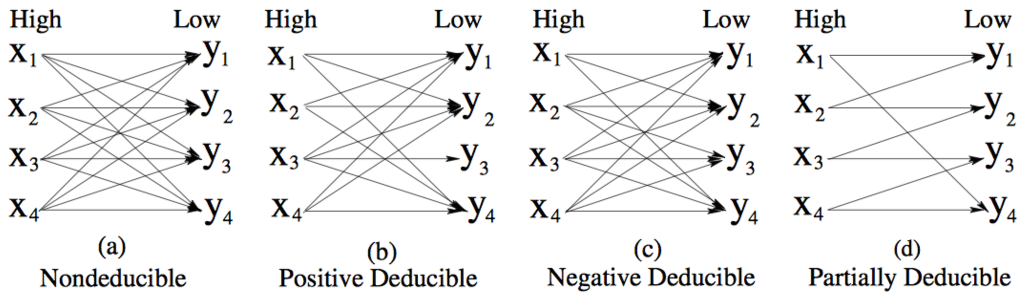


Figure 2.1: Cover channel types.

In their context, *High* and *Low* identify two real-time tasks from different levels of confidentiality, and as their names might clarify, *Low* task should not deduce information (with certainty) from *High* task. The authors defined X to be a set of all possible available symbols for *High* task to be transmitted, and Y be a set of all possible symbols that *Low* task receives through a noisy channel.

Let channel=$\{(x, y)|x \in X, y \in Y\}$ where a noisy channel, for instance, can be defined as: channel=$\{(x_1, y_1), (x_1, y_3), (x_2, y_2)\}$, in which an input symbol can be mapped to more than one output symbol.

Their classified covert channels are briefly explained as follows:

- In a *non-deducible* noisy channel, Low task receives all the symbols from *High* task, but after receiving any symbol from *High* task, *Low* task cannot reliably deduce which symbol has been sent by *High* task, in other word, here we have an ideal protected channel against the attack;
- In a *positive-deducible* noisy channel, *Low* task can definitely say which symbol is sent by *High* task. For example, in Figure 2.1 (b), after receiving y_3 by *Low* task, Low can reliably deduce that x_3 has been sent by *High* task;
- In a *negative-deducible* noisy channel, *Low* task can definitely say which symbol has not been sent by *High* task. For example, in Figure 2.1 (c), after receiving y_3 by *Low task*, *Low* can reliably deduce that x_3 has not been sent by High task;

- In a *partially-deducible* noisy channel, the occurrence cannot be classified as any of the other three types.

According to [41], we can assume that *High* is a Trojan horse and Low is an adversary. Trojan horse, while hiding in a computer system, has the intention of obtaining adversary′s information. The following is a possible scenario of the Trojan and the adversary for a positive-deducible covert channel; Trojan tries to transmit either of x_1, x_2 or x_4 as the normal mode operation, as Trojan found information from adversary, it changes its mode to only sends x_3, simultaneously, adversary ignores all the symbols but only y_3. By observing received y_3 symbols, it collected classified information from Trojan, and accordingly an instance of positive-deducible channel has happened.

2.1.2 FreeRTOS

FreeRTOS is a free and open source Real-Time Operating System (RTOS) for embedded systems. FreeRTOS can be used as a hard real-time operating system with either Preemptive or Collaborative/non-Preemptive fixed priority scheduler, capable of both Inter-Process Communication (IPC) and Shared Memory functionalities.

Code Architecture

In abstract, an application relying on FreeRTOS can architecturally consist of separate layers of codes as llustrated in the Figure 2.2 [1]:

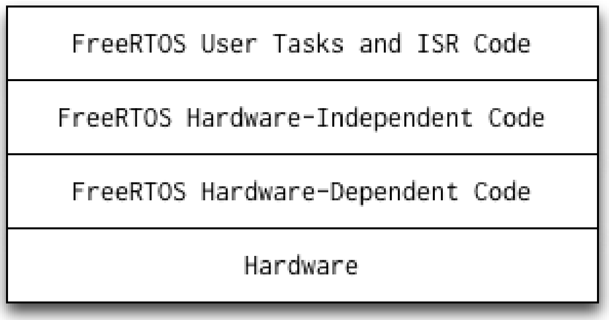

FreeRTOS User Tasks and ISR Code
FreeRTOS Hardware-Independent Code
FreeRTOS Hardware-Dependent Code
Hardware

Figure 2.2: FreeRTOS code layers.

Hardware independent code, which includes all of the FreeRTOS kernel, ISRs, and user-defined tasks codes. FreeRTOS has a minimal light-weight kernel consisting of tasks.c, list.c, queue.c and timers.c with their corresponding header files. ISRs are implemented within FreeRTOS kernel codes. As an important example, the system tick the heartbeat of FreeRTOS system, the tick interrupt frequency is user-configurable by setting the value of configTICK_RATE_HZ in the FreeRTOSConfig.h file. In each tick interrupt, vTaskSwitchContext kernel function will be called- the core functionality of FreeRTOS task scheduler, which initializes pxCurrentTCB variable with the highest priority ready task [5, 19].

Hardware dependent code provides a Hardware Abstract Layer (HAL) or a port for a specific integration of compilers (GCC, IAR, Keil, MemMang, etc.) with processor families (ARM7, ARM Cortex-M3, PICs, x86, etc.). The core functionalities are mostly implemented in the port.c and partmarco.c. Many official/contributed ports are available as demonstrated demos at the FreeRTOS website.

Task Scheduling

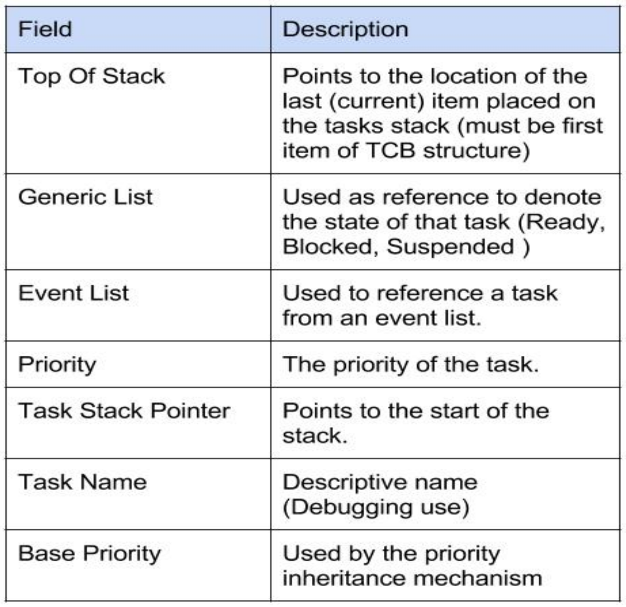
FreeRTOS as any other typical operating system, manages tasks by their Task Control Blocks (TCB). The field of each TCB data structure is illustrated in Figure 2.3 [5].

By default, the stack grows decreasingly, then avoiding a stack overflow requires to compare the top of stack value against the task stack pointer.

Task states within FreeRTOS are switching illustrated in Figure 2.4 [5].

To create a new task, FreeRTOS instantiate a TCB and add it to a *Ready* list based on the task′s priority since there exists a dedicated *Ready* list for each priority levels - multiple Ready lists equal to the maximum priority. No list exists for running task, in fact the running task is still in the *Ready* list, but is recognized using pxCurrentTCB which points to the currently running task′s TCB. The lists are implemented as the circular double linked list data structure. The functionality of *Ready* list is shown in details in Figure 2.5 [1].

Block state happens due to the lack of accessing to a required resource. In FreeRTOS, a resource means: using a queue, or obtaining semaphore as a specific case of queues. Thus, writing to a full queue or reading from an empty queue will cause a *Block* state. However, It can be chosen to be a *Block* state or not, since user can choose not having the task blocked by setting xTicksToWait as 0 (means that blocking time is zero). Otherwise the task will be blocked equal to tick value defined in xTicksToWait, as long as an event such as freeing up a full queue or filling in an empty queue did not happen, this type of event is called an external event. In a blocking scenario, the scheduler will transform the task into *Delayed* list. A task can also voluntarily be



Field	Description
Top Of Stack	Points to the location of the last (current) item placed on the tasks stack (must be first item of TCB structure)
Generic List	Used as reference to denote the state of that task (Ready, Blocked, Suspended)
Event List	Used to reference a task from an event list.
Priority	The priority of the task.
Task Stack Pointer	Points to the start of the stack.
Task Name	Descriptive name (Debugging use)
Base Priority	Used by the priority inheritance mechanism

Figure 2.3: FreeRTOS task TCB components.

in Block state waiting for a temporal event. In this case, a task can call vTaskDelay function to be held in Block state for a period of time, in this scenario, the task itself (not scheduler) puts itself into *Delayed* list. The last mentioned mechanism can also be exploited for implementing a periodic task. As we can see, blocked tasks will be unblocked after a *timeout* period, within blocking time, these tasks will not be scheduled. A *Delayed* task list is used for this purpose. The scheduler checks for any *timeout* delayed task at every point of scheduling decision, then the scheduler will transform unblocked tasks from *Delayed* list into their corresponding *Ready* list. It is important to mention that *timeout* counter should be as small as 8-bits, to deal with overflow waking times, the task will be placed in the OverflowDelayedTaskList instead. Then, at each vTaskIncrementTick execution, if the check for the counter overflow was true, then two lists of DelayedTaskList and OverflowDelayTaskList will be swapped. In addition, any task can be suspended if it is running or servicing ISRs. Suspended tasks are not considered for scheduling decisions and they do not have any *timeout* periods, and should be explicitly taken out of Suspended state and put into a Ready state. [19]

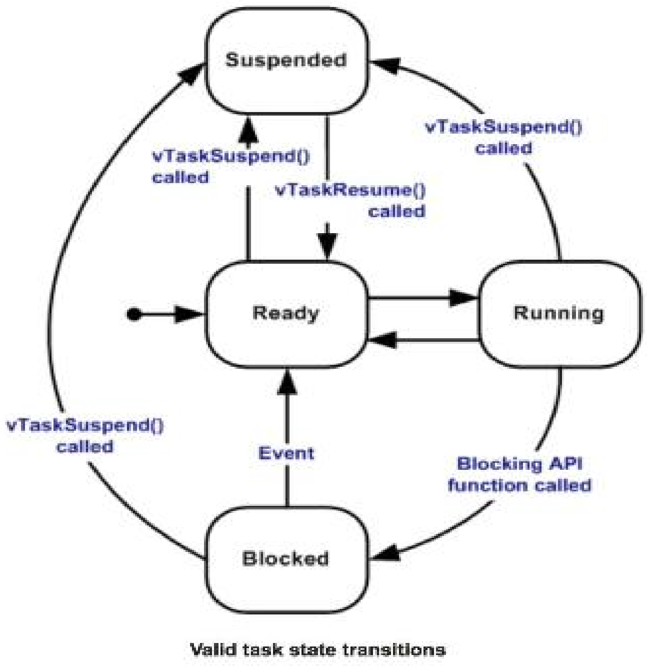


Figure 2.4: FreeRTOS task states.

2.1.3 Genetic Algorithm

In this research, we developed a linear un-constraint genetic algorithm, combined with a problem specific heuristic. The purpose is to minimize utilization and maximize schedulability.

Genetic Algorithms (GAs) are stochastic search algorithms to find global optimum solutions for optimization problems. GAs have underlying bio-inspirational mechanisms based on natural genetics, and try to find near-to-optimal solution starting from an initial population of chromosomes (or individuals), using basic operators called: selection, crossover and mutation. Chromosomes are typically binary strings, called bitstrings which evolve through progressive iterations, called generations. Within each generation, some of the chromosomes will be newly generated, using GA operators, and then will be evaluated by the fitness function. Finally, after checking against defined stopping criteria, the algorithm finishes resulting with a binary representation of the best solution.

An overview on GA development is illustrated in Figure 2.6. Let P(g) be the current population and C(g) be offstring chromosomes which are split up to be selected for different operators or be picked as elites. Then, the computed chromosomes are passed to be evaluated with Fitness function. At the end of each cycle of GA generation, the new population is tested against stop

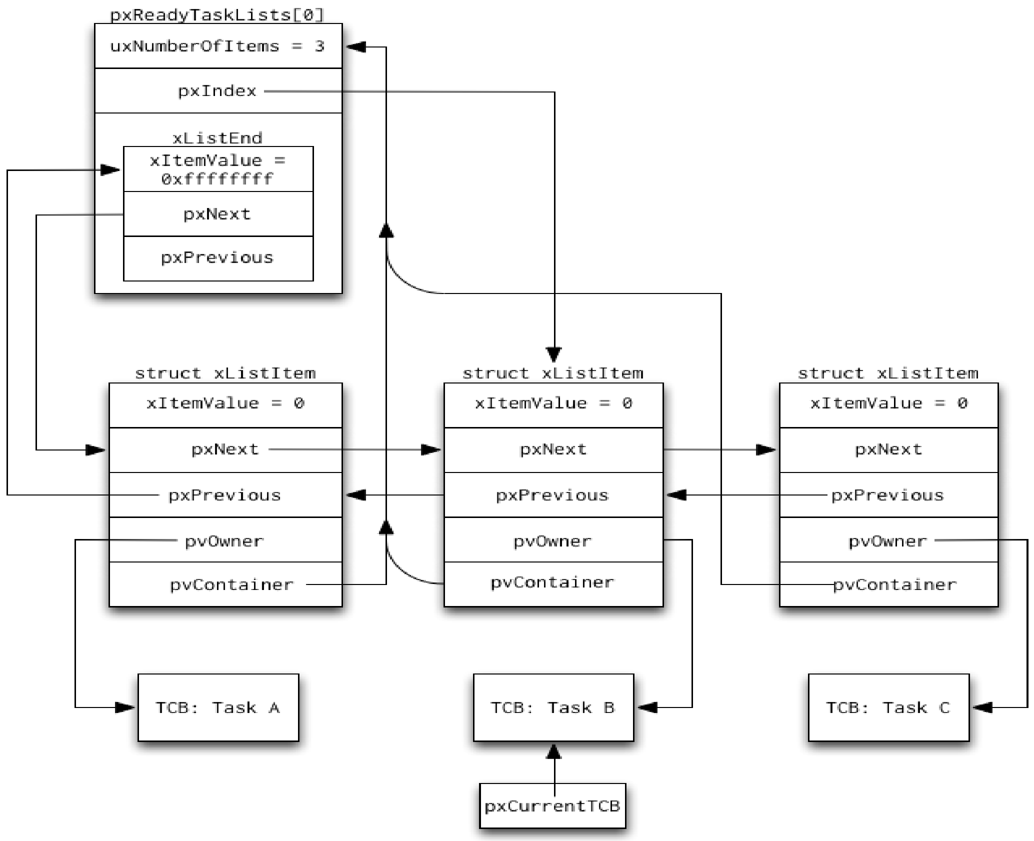


Figure 2.5: FreeRTOS′s Ready queue functionalities.

criteria that can be, for example, the maximum number of generations. If the stop criteria is met, then the GA will be finished and the best chromosome is the output; otherwise, the next generation will be processed.

In the following, we describe some of the GA basics: Population, Crossover and Mutation. GAs start working with initial population of individuals. In each generation of GA, some of the individuals will be chosen to feed Crossover and Mutation operators, afterwards, the newly generated individuals (or offsprings) will be collected to create a modified population. This process will be repeated until there is no need to work upon a new generation, because the best solution has already been found, or because the stop criteria is already met.

As mentioned, the individuals in the population are supposed to be evaluated with a fitness function that the developer has provided. The fitness function evaluates each of the given individ-

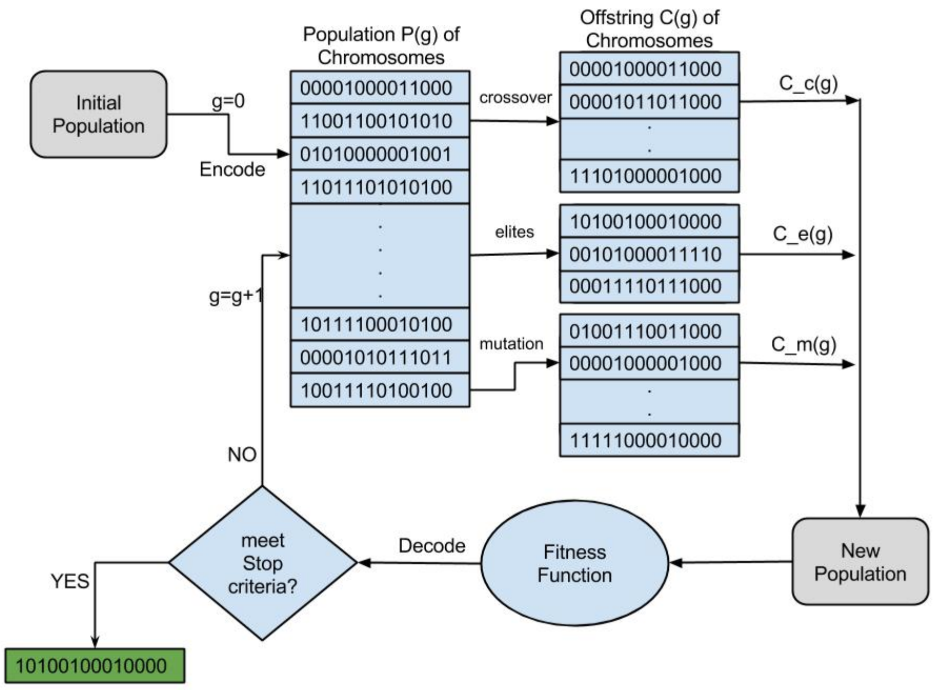


Figure 2.6: Overview of Genetic algorithm.

uals and will assign a score to each of them, which approximately describes the eligibility of each solution (individual) in relation with the other found solutions. Selection between individuals are so dependent on the developer′s strategy, however, it would be generally based on the scores of each individual. The basic idea is that the qualified individuals are chosen to generate some other individuals which are expected to approximately be close to the best optimized solution than the previous individuals.

Different Crossover operators might be defined for different problems. As a simple example, Figure 2.7 shows two parents with bitstring data type, and the child takes one part of its bitstring from Parent 1 and the other part from Parent 2, this method is called single point Crossover.

Finally, Mutation operator takes some of the chromosomes or individuals and replaces them with new individuals. To clarify the functionality, in Figure 2.8, a bitstring type chromosome will be evaluated for each of its compound bits, and for instance, with a 50% probability will switch each bit to 0, and 1 otherwise.

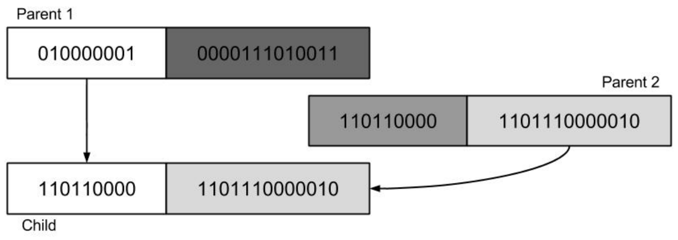


Figure 2.7: An example of single point crossover operator.

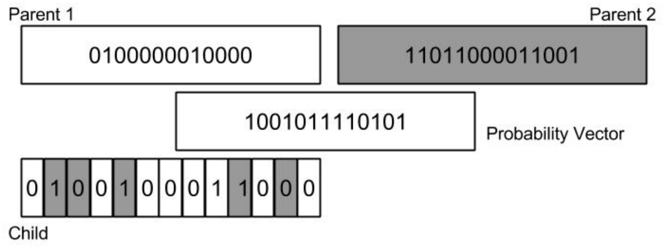


Figure 2.8: An example of mutation operator.

2.2 Related Work

Covert side channels are identified, analyzed, and mitigated in some works [21, 22, 24, 26, 37]. Here, we focus on cache memory resources. Timing attacks through highly "$stateful$" cache resources have been studied and documented in some research works [26, 37, 53]. In particular, Hu [53] assumes a similar cache flushing strategy to mitigate timing channel attacks, and discusses how scheduling algorithms can be modified to minimize the number of flushes. However, tasks do not have any real-time requirements and the scheduler does not support any service guarantee.

Focusing on relevant works alongside considering real-time requirements, Son et al [41], showed that a covert channel can be established in the rate-monotonic scheduler. Volp et al [48], studied unauthorized information flows obtained through altered scheduling behaviour, that is delayed preemption, then they discussed on modifications to a fixed-priority scheduler which reduces such vulnerabilities. Later in 2008 [47], they focused on the effect of timing channels introduced in by real-time resource locking protocols, and addressed them by transforming the relevant protocols.

Some recent works [40, 44, 52, 54], have focuses on both Hardware and Software as a combined architectural model to relieve security issues such as intrusion detection.

2.2.1 Security and Real-Time Task Scheduling

This research rather than specifically focusing on timing channel attacks, tries to mitigate security properties by defining the equivalent real-time task scheduling constraints. Some other works tracked timing channels in real-time systems between tasks of different levels of security; they also tried to integrate the security requirement with Rate Monotonic (RM) scheduler [41]. Unlike them, our collaborative research focus was on timing channels caused by shared resources. In addition, rather than having an underlying Multi-Level Security (MLS), we preferred to generalize the model to have any arbitrary security constraints (the model is discussed in more details in Section 3.2).

Similarly, Xin et al. [50] introduced a new scheduler, and Lin et al. [28] extended the existing Earliest Deadline First (EDF) scheduler to meet both security and real-time requirements, in such a way that overheads of priodic tasks with different security service times can be relaxed through task scheduling.

Some other research works [10,42,43] studied general information leakage issue in real-time database systems with MLS constraints. [42] has focus on transaction scheduling and concurrency control algorithms to meet both security and real-time requirements, which includes metrics to measure fulfilment of security requirements. In addition, in [10] authors discuss about the trade-off between security and real-time requirements. In [43], the conflict between real-time and security requirements is resolved by defining a notion of partial security.

In [33], authors used information leakage as an example to illustrate their techniques in order to mitigate security requirements through task scheduling. They study enhancements to Fixed Priority (FP) scheduling to reduce information leakage through shared resources while meeting real-time requirements. They defined algorithms, called PF and CPF, and studied the issue of computing the number of cache flushings which are mostly related to number of preemptions suffered by a task or group of tasks. This research is a collaborative extension to theirs as follows: (a) the security model is generalized to capture the relationships between tasks; (b) our methods have been demonstrated by implementation of a realistic application and hardware platform. Existing work [33], has discussion on computing the exact preemption costs for tasks. In comparison, in this work [36], other algorithms are developed in which rather than cleaning up state (or do cache flush) in each preemption point, the confidentiality of the preempted task determines if a cache flush is required or not. In addition, in order to reduce the number of cache flushes and to increase schedulability, an optimal preemptivity assignment algorithm is developed to specify if a task should be preemptive or non-preemptive. [36] is a collaborative extension to [33], in the sense of generalization.

Chapter 3

Security Model

In this section, we introduce the developed security model which is based on confidentiality relations between tasks. Also, in order to demonstrate the security model, an example avionics case study is detailed in Section 3.1.

One of the main contributions of this research is to introduce a general security model that is an extension to [33]. That work proposed a security model that considers tasks with varying security levels and avoid information leakage between tasks. Here, we generalize that model by introducing security relations between tasks as an arbitrary matrix. Tasks in the system are subject to security constraints and the given matrix establishes security requirements of each task corresponding with each other task. The matrix is a true/false matrix, in which, each of true values can introduce a condition of information leakage that should be avoided and protected against. We model such security constraint by introducing a binary $noleak$ relation between any two different tasks τ_i and τ_j: if $noleak(\tau_i, \tau_j) = T$ (true), then we require that information leakage from τ_i to τ_j is prevented; otherwise if $noleak(\tau_i, \tau_j) = F$, no such constraint needs to be enforced between τ_i and τ_j. Note that we do not impose any other property on the $noleak$ relation; in particular, we do not require properties of symmetry (i.e., it might hold $noleak(\tau_i, \tau_j) = T$ but $noleak(\tau_j, \tau_i) = F$) or transitivity. We also do not assume any special relation between the real-time properties of a task and the $noleak$ requirements [36].

We argue that the described scenario (*i.e.,* tasks with different protection levels) can arise in various complex real-time systems where applications developed by different vendors are integrated together on the same computing platform. Examples include avionics systems designed according to the DO-178B standard [17], as well as emulation and integration systems designed for the porting of legacy applications, such as the "RePLACE" system from Northrup Grumman [20, 38].

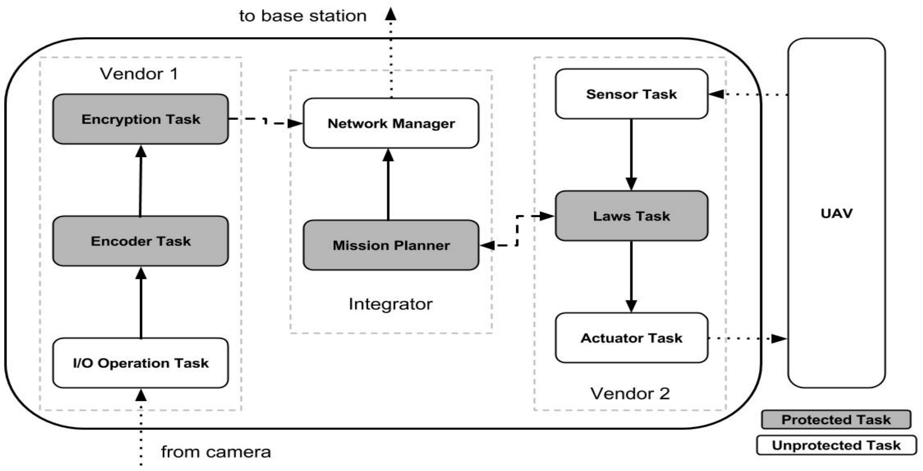


Figure 3.1: Demonstrator Overview

3.1 Case Study

As discussed in the previous section, $noleak$ relation have been defined to protect the confidentiality between tasks of different sub-systems. In this section, a case study is used to demonstrate how $noleak$ can work in a real application to preserve the security constraints.

The Real-Time Embedded Systems Lab (RESL) at the University of Waterloo (UW) [6], developed a tested avionics case study in two underlying implementations of QNX and Real-Time Linux (RTL). In this research, we took the fundamental control software of the case study, and extended it to have fundamental tasks and components of our research [4].

In order to motivate and evaluate the presented research, we use the example of the Electronic Control Unit (ECU) for an avionics platform system detailed in Figure 3.1. This demonstrative system, built and implemented at the University of Waterloo, runs many of the same types of tasks which could be expected to run on an Unmanned Arial Vehicle (UAV) surveillance system, as discussed in the last two previous sections. The ECU communicates locally with the inertial sensors, GPS localization system and actuators ("UAV" in the figure), as well as a camera subsystem. The ECU also uses off-board communication to exchange information with a base station. We assume that three parties are involved in building the ECU system, Vendor 1, Vendor 2, and the Integrator. Each party is responsible for a different ECU subsystem, which in turn comprises different real-time tasks. One or more tasks of each party have some degree of protected data. Each party trusts communication between tasks in its own subsystem, but wants to prevent information leakage from its protected task or tasks to other subsystems. For example, detected images can have a security classification, or control tasks can implement a proprietary algorithm.

Vendor 1 is responsible for the *image subsystem*. The I/O Operation Task mimics the behavior of a camera driver; to perform repeatable experiments, our system simply places a fixed set of images into a Memory File System (MFS) and extracts them in order. Since conceptually this task manages input state and not image data, it does not need to be protected. The Encoder task is realized as a JPEG compressor. Encryption task uses the AES cipher using a protected, secret key. The encrypted image is then passed to the network manager in the Integrator subsystem.

Vendor 2 is responsible for the *control subsystem*. The sensor task receives, parses examines incoming sensor data for the other tasks. The Laws task computes the control output to move the UAV towards a waypoint determined by the Mission Planner in the Integrator subsystem. Finally, the Actuator Task prepares the actual output commands and send them to physical actuators.

Finally, the *Integrator* is responsible for connecting the two previous subsystems together and performing mission control. The Mission Planner Task communicates with the Laws Task to determine the current position of the UAV and move it between a set of fixed waypoints. The Network Manager sends encrypted data coming from the Mission Planner and the Encryption Task to the base station.

The proposed binary $noleak$ relation between any two tasks can be generalized to implement a range of security constraints. In this work we focus on unintended information flow (or information leakage) between tasks of different vendors through the use of shared resources. In particular, we consider a vendor oriented security model where information leakage from a protected or sensitive task of one vendor to any task of a different vendor is considered undesirable while no such constraints are placed on tasks within a given vendor's subsystem. Such a model is useful in many scenarios. For example, in our avionics demonstrator scenario, images captured by the camera could have a higher security classification and it may be undesirable for Vendor 2 to gain unintended information about them even if the vendor is trusted with controlling the UAV. Similarly, the control laws from Vendor 2 may contain a proprietary algorithm and Vendor 2 may not want other vendors to gain unintended knowledge about the algorithm.

The $noleak$ relations between the tasks using the vendor oriented security model for the avionics demonstrator case study are listed in Table 3.1. While the proposed vendor oriented security model has the flavor of multi-level security models (e.g., [11]), it is quite different from such models. In fact the proposed model is quite relaxed relative to traditional multi-level security (MLS) security models such as the well-known Bell-LaPadula [11] model that aim to prevent information flow from a higher security level to a lower security level even within the same compartment (in this case vendor). In contrast, note that no constraints on leakage are placed between tasks from the same vendor in the proposed model. Similarly, no constraints

$noleak$		to							
		Sens.	Laws	Act.	MP	Net.	AES	JPEG	I/O
from	Sens.	-	F	F	F	F	F	F	F
	Laws	F	-	F	T	T	T	T	T
	Act.	F	F	-	F	F	F	F	F
	MP	T	T	T	-	F	T	T	T
	Net.	F	F	F	F	-	F	F	F
	AES	T	T	T	T	T	-	F	F
	JPEG	T	T	T	T	T	F	-	F
	I/O	F	F	F	F	F	F	F	-

Table 3.1: $noleak$ relations of the case study

on leakage are placed between unprotected tasks (not security sensitive) even if they are from different vendors. However, it is important to note that one could capture stricter security models using the proposed binary $noleak$ relation when the system at hand warrants such a stricter model.

Chapter 4

Security Mechanisms

As discussed in the previous chapters, the focus of this research is on protecting shared physical resources. Both the software and hardware execution platform influence tasks' ability to leak information. We assume that the employed platform already provides mechanisms such as virtual memory to prevent or control explicit communication between tasks.

Modern embedded systems overlay some highly "stateful" shared resources, such as caches, DRAM, Data Bus and I/O interconnections. Their "stateful" property is very considerable, because any state change caused by one task might have significant timing effect on the next executing tasks. Timing attacks based on cache state have been summarized in Section 2.2. Other resources could similarly be used as covert channels between tasks. For example, DRAM controllers typically implement an open row mechanism that behaves in a manner that is similar to a cache [49]. It has been shown that I/O buses can carry traffic belonging to one task even after a new task has been scheduled [34].

In general, there are multiple mechanisms that can be employed to mitigate or outright avoid information leakage due to a shared resource. In this thesis, we consider two such mechanisms. First, we propose to use a general "flushing" mechanism: whenever we detect that a task might undesirably leak information to another task, we execute a synthetic 'Flush Task' (FT) function that resets the state of all needed resources. Second, some resources can be divided into multiple partitions, and then tasks can be statically allocated to partitions. For example, cache can be partitioned into ways, and DRAM into banks. This prevents tasks allocated to one partition to leak information to tasks executing on a different partition.

The rest of the chapter is organized as follows. First, the task model is discussed in Section 4.1 to provide definitions for cache-aware execution time, and overheads which will be referred in the rest of the paper. Then, the two developed mechanisms for undertaking security protection on shared physical resources are described in Sections 4.2 and 4.3, including some examples to clarify the effectiveness of the proposed techniques.

4.1 Task Model

We consider the fixed priority scheduling of a set $\Gamma = \{\tau_1, \ldots, \tau_N\}$ of N sporadic real-time tasks on a uniprocessor. We consider information leakage through a single resource which can be partitioned into K partitions of equal size. Each task τ_i is characterized by a tuple $\{p_i, c_i(k_i), preempt_i\}$. p_i is the task's period or minimum inter-arrival time. $c_i(k_i)$ is the task's worst-case execution time when it runs non-preemptively with k_i partitions assigned to the task. Finally, $preempt_i$ determines the task preemptability: if $preempt_i = T$, then the task can be preempted by higher priority tasks; if instead $preempt_i = F$, a job of τ_i always run to completion once started.

Tasks are assigned distinct priorities; without loss of generality, we assume that the priority of task τ_j is higher than the priority of task τ_i if and only if $j < i$. For ease of notation, let $hp_i = \{\tau_j | 1 \leq j < i\}$ be the set of all tasks with priorities higher than τ_i and let $hep_i = \{\tau_j | 1 \leq j \leq i\}$ be the set of higher priority tasks including τ_i; similarly, we define the set of lower priority tasks $lp_i = \{\tau_j | j > i\}$. Finally, we assume that time is an integral quantity measured in multiples of the system tick. For simplicity we do not discuss the effects of locking protocols, but the analysis could be extended to include confidentiality-preserving real-time locking schemes [47].

4.2 Resource Flushing

Resource flushing technique resets the state of the shared resource before transferring execution to an untrusted task. By resetting or flushing the whole cache, there is no possibility for an untrusted task to track timing information from a protected task based on the extra timing that it takes for cache line evictions which probably belong to the protected task. As another example, in order to prevent information leakage in DRAM, the state can be reset by closing and opening the bank.

In this thesis, we consider the "flushing" mechanism introduced in [36]; whenever we detect that a task might undesirably leak information to another task, we execute a synthetic 'Flush Task' (FT) function that resets the state of all needed resources. Based on the $noleak$ relation defined in Chapter 3, the following *No-Leak Flush (NLF)* mechanism is implemented:

	preempt	I_j
τ_1	T	3
τ_2	F	2
τ_3	T	1

$noleak$		to		
		τ_1	τ_2	τ_3
from	τ_1	-	T	F
	τ_2	T	-	T
	τ_3	T	F	-

Table 4.1: Example Task Set and "no-leak" Relationships Between Tasks

Definition 1 (NLF Mechanism)**.** *Let Γ' be the set of tasks executed since the last FT. Then if there $\exists\tau_j \in \Gamma', noleak(\tau_j, \tau_i) = T$, a FT must be executed before scheduling task τ_i.*

Note that our $noleak$ model does not make any assumption on which portion of a task reads/writes sensitive data; hence, in the situation stated by Definition 1, we need to flush even if the job of τ_j did not finish executing. Let c_{ft} be the worst-case execution time of the FT. Note that we assume that if a FT is required, it is executed as part of τ_i, i.e., the execution time of τ_i is effectively increased to $c_i + c_{ft}$. Once the NLF mechanism has been defined, the schedulability of task set Γ can be guaranteed by computing a safe upper bound N_{ft} to the number of FT required in the busy interval of any task $\tau_i \in \Gamma$; we show how to do so in Chapter 5.

4.2.1 Impact on Scheduling

An example schedule under NLF is shown in Figure 4.1 for the task set in Table 4.1, where I_j represents the number of instances of τ_j in the depicted busy interval[1] starting with the release of τ_3. Note that a FT is required before τ_3 is scheduled because τ_2 executed before the start of the busy interval and $noleak(\tau_2, \tau_3) = T$. Similarly, a FT is required before τ_1 can preempt τ_3 because $noleak(\tau_3, \tau_1) = T$; however, no FT is required when execution returns to τ_3 since $noleak(\tau_1, \tau_3) = F$. Finally, a FT is required before executing τ_2 even if $noleak(\tau_3, \tau_2) = F$ because $noleak(\tau_1, \tau_2) = T$ and no FT is run between τ_1 and τ_3.

It is interesting to note that task preemptivity can have a significant effect on the FT number. As an example, Figure 4.2 shows the schedule for the same task set as in Table 4.1, except that

[1]Note that since Definition 1 depends only on the sequence of task executions, for simplicity throughout all figures we do not report or draw execution times to scale.

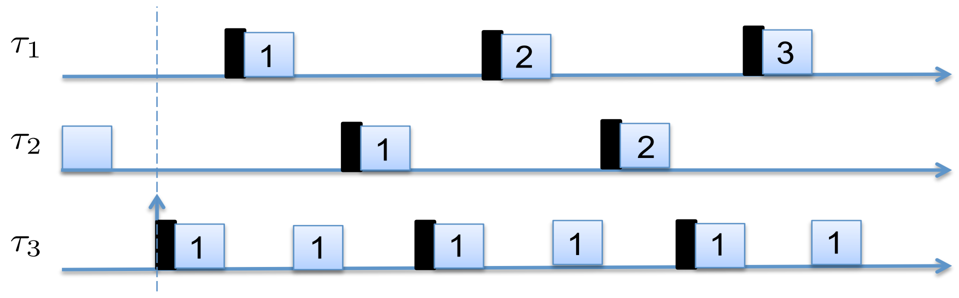


Figure 4.1: Example task set: worst-case schedule. Vertical bars represent FTs. Numbers represent job indexes (i.e., τ_3 executes a single job in the busy interval).

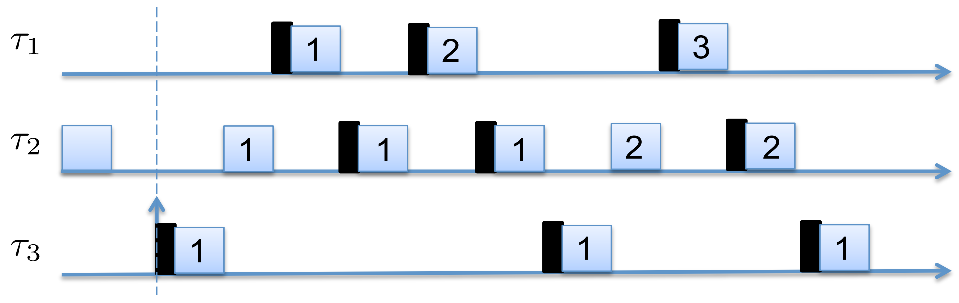


Figure 4.2: Example task set: preemptive worst-case schedule.

all tasks are preemptive. In this case, the number of FT is 9 rather than 8. Figure 4.3 shows the same example when all tasks are non-preemptive, resulting in 5 FT. As we show in Chapter 5, these are in fact exact bounds on the worst-case number of FT suffered by the example task set. In general, task preemption creates additional context switches that can increase the number of FT; on the other hand, executing a task non-preemptively creates blocking time for higher-priority tasks. In Chapter 5, we first provide a schedulability analysis for task set Γ assuming that the preemptability $preempt_i$ of each task τ_i is known; then, using the derived schedulability analysis, in Section 5.3 we show how to optimally assign $preempt_i$.

4.3 Resource Partitioning

Apart from the flushing technique discussed in the last section, we have studied a well-known technique called resource partitioning as a solution to the information leakage problem. Regarding cache memory resources, this technique in our single-core system model, can reduce cache evictions by assigning different partitions to various tasks. More in general, when applied

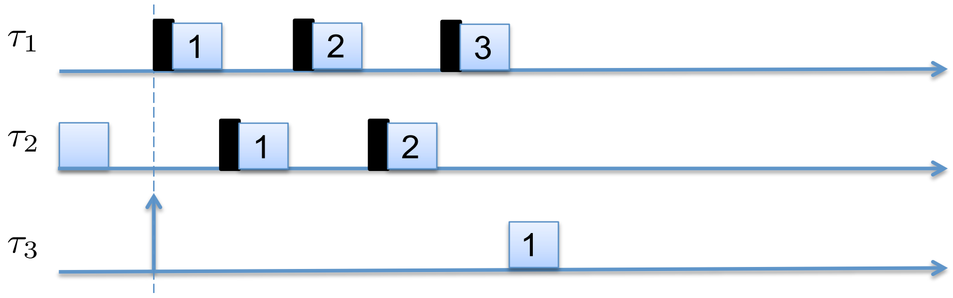


Figure 4.3: Example task set: non-preemptive worst-case schedule.

in conjunction with the NLF mechanism, the technique has two advantages: 1) it can reduce the number of required FT. This is because if we switch from a task τ_i to a task τ_j such that $noleak(\tau_i, \tau_j) = T$ but τ_i and τ_j do not share a partition, then no FT is needed. 2) It reduces the time required to perform a FT. In the situation described above, if τ_i and τ_j share a single partition, then only that partition would need to be flushed rather than the entire resource.

The disadvantage of using this technique is that it reduces the amount of resource available to a task. In the case of cache resource, the task execution time might be increased since the task is forced to use only a limited number of cache partitions rather than the whole cache. Therefore, we have a trade-off between having less and shorter FT, and speeding up computation by allowing each task to use the entirety of the resource.

In general, we will assume that a task can be assigned to any number k_i of partitions, from 0 to the maximum number K of partitions in the system. To keep track of the partition assignment, we define boolean values $a_{i,k}$, for all $i = 1 \dots N$ and $k = 1 \dots K$, where $a_{i,k} = T$ if task τ_i uses the k^{th} partition and $a_{i,k} = F$ otherwise. The problem is then how to assign the values of $a_{i,k}$ in such a way as to maximize system schedulability; we study how to do so in Chapter 6 based on the schedulability analysis detailed in Chapter 5. To keep track of the overhead of partition flushing, we further define a partition-specific no-leak relation as follows:

$$noleak_k(\tau_i, \tau_j) = noleak(\tau_i, \tau_j) \text{ and } a_{i,k} \text{ and } a_{j,k}. \tag{1}$$

In other words, for the k^{th} partition, $noleak_k(\tau_i, \tau_j)$ is true if $noleak$ relation for tasks τ_i, τ_j is true and both tasks are assigned to partition k; this implies that the partition must be flushed when τ_i is followed by τ_j. We can then define a *partition-aware* NLF mechanism as follows:

Definition 2 (Partition-Aware NLF Mechanism)**.** *Let Γ' be the set of tasks executed since the last FT. Then for each partition k, if there $\exists \tau_j \in \Gamma', noleak_k(\tau_j, \tau_i) = T$, a FT for partition k must be executed before scheduling task τ_i.*

Since when using the partition-aware NLF mechanism, a flush task invocation resets a single partition, c_{ft} shall represent the time required to flush one partition rather than the entire resource.

Finally, since the execution time of a task τ_i depends on the number k_i of partitions assigned to the tasks (i.e., such that $a_{i,k} = T$), we assume that the worst-case execution time $c_i(k_i)$ for τ_i is defined as a function of k_i. We further assume that $c_i(k_i)$ is measured with the task running non-preemptively. In practice, if $preempt_i = T$, then the task can be preempted while executing. To account for the preemption overhead, we introduce an additional term c_{rt} representing the time required to "reload" the state of a single resource partition; every time the task is preempted, it then suffers an extra overhead equal to $k_i \cdot c_{rt}$. For example, in the case of a cache resource, the reloading term represents the overhead caused by preempting tasks evicting cache lines belonging to a preempted task. In the case of a DRAM, it represents the overhead of closing and re-opening rows accessed by the preempted task.

While we argue that our methodology could be applied to a variety of shared resource, in this thesis we limit our implementation and evaluation to cache resources. Cache partitioning is performed using cache ways, and only last level cache is considered. Implementation details are discussed in Chapter 7.

Chapter 5

Schedulability Analysis

In this chapter, we detail how to determine schedulability for a task under analysis τ_i based on the system model and partition-aware NLF mechanism detailed in Chapter 4. The discussed analysis also works for the case of pure resource flushing (i.e., no partitioning) by setting the number of partitions $K = 1$.

Our schedulability analysis relies on determining an upper bound to the number N_{ft} of FT for τ_i on partition k. In the remaining of the paper, we shall consider bounds on N_{ft} based only on the $noleak_k$ relation and the number of higher priority jobs that interfere with τ_i, i.e., we make no assumption on the exact arrival time of interfering jobs. While this could lead to a potentially pessimistic bound on N_{ft}, it nevertheless allows us to decouple the determination of the worst-case number of FT, which depends on the job execution order, from the determination of the number of interfering jobs of higher priority tasks, which is based on the critical instant arrival pattern. Therefore, we shall write $N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\})$ to denote that N_{ft} is a function of $noleak_k$ and the number I_j of jobs of higher priority tasks τ_j that interfere with τ_i. Similarly, we determine the number of reloads $N_{rl}(\{a_{j,k}\}, \{I_j | \tau_j \in hp_i\})$ on partition k as a function of the partition assignments $\{a_{j,k}\}$ and number of higher priority jobs I_j only.

We use the analysis strategy in [12] for fixed-priority scheduling to determine the schedulability of τ_i, except that we add a term $\sum_{k=1}^{K} N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\}) \cdot c_{ft}$ to account for the overall FT time and a term $\sum_{k=1}^{K} N_{rl}(\{a_{j,k}\}, \{I_j | \tau_j \in hp_i\}) \cdot c_{rl}$ to account for the overall reload time. More in detail, τ_i is schedulable iff $\exists t, 0 < t \leq p_i$, such that:

$$B_i+\sum_{k=1}^{K} N_{ft}(noleak_k, \{I_j|\tau_j \in hp_i\})\cdot c_{ft}+\sum_{k=1}^{K} N_{rl}(\{a_{j,k}\}, \{I_j|\tau_j \in hp_i\})\cdot c_{rl}+\sum_{\forall \tau_j \in hp_i} (I_j \cdot c_j)+c_i \leq t, \tag{2}$$

where B_i represents the maximum blocking time induced by lower priority tasks and their FT. If τ_i is non-preemptive, then the number of interfering jobs I_j of τ_j is computed as:

$$I_j = \Big\lfloor \frac{t - c_i}{p_j} + 1 \Big\rfloor, \tag{3}$$

while if τ_i is preemptive it is computed as:

$$I_j = \Big\lceil \frac{t}{p_j} \Big\rceil. \tag{4}$$

Furthermore, the maximum blocking time B_i is:

$$B_i = \max_{\forall \tau_j \in lp_i \wedge preempt_j = F} \bar{c_j} - 1, \tag{5}$$

while $\bar{c_j}$ is computed as:

$$\bar{c_j} = c_j + \sum_{k=1 \ldots K, \exists \tau_l : noleak_k(\tau_l, \tau_j) = T} c_{ft}; \tag{6}$$

i.e., the execution time of lower priority non-preemptive task τ_j must be increased by c_{ft} for each partition k such that a FT might be required for τ_j on k. The -1 term in Equation 5 accounts for the fact that the lower priority blocking task must arrive at least one time unit before the activation of τ_i.

Note that in practice it is sufficient to test Equation 2 on all points before a discontinuity in I_j, which are $S_i = \{r \cdot p_j | \tau_j \in hp_i \wedge 1 \leq r \leq \lfloor p_i/p_j \rfloor\}$ if τ_i is preemptive, and $S_i = \{r \cdot p_j + c_i - 1 | \tau_j \in hp_i \wedge 1 \leq r \leq \lfloor p_i/p_j \rfloor\}$ otherwise; [12] shows how to further reduce the number of required testing points. Furthermore, from Equation 2 it follows immediately that we can compute the *slack* Δ_i for τ_i as:

$$\Delta_i = \max_{t \in S_i} \Big(t - \big(B_i + \sum_{k=1}^{K} N_{ft}(noleak_k, \{I_j|\tau_j \in hp_i\}) \cdot c_{ft} + \sum_{k=1}^{K} N_{rl}(\{a_{j,k}\}, \{I_j|\tau_j \in hp_i\}) \cdot c_{rl} + \sum_{\forall \tau_j \in hp_i} (I_j \cdot c_j) + c_i \big) \Big), \tag{7}$$

where τ_i is schedulable iff the slack is non-negative, in which case Δ_i represents the additional amount of time that τ_i can take to complete while remaining schedulable; we will use this value in Section 5.3.

A trivial bound on $N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\})$ can be obtained as the number of context-switches for tasks using partition k, including the one at the beginning of the busy interval if $a_{i,k} = T$. If a task τ_j can preempt another task τ_l in the busy interval, τ_j accounts for two context-switches (when τ_j starts by preempting τ_l, and when execution returns to τ_l); otherwise, τ_j accounts for only one context-switch (when it starts). Hence, let $\Gamma^2_{i,k} = \{\tau_j \in hp_i | \exists \tau_l \in hep_i : l > j \wedge preempt_k = T \wedge a_{j,k} = T \wedge a_{l,k} = T)\}$ be the set of all higher priority tasks using partition k that can preempt another task in the busy interval for τ_i (i.e., either τ_i or another task with priority lower than l but higher than τ_i must be preemptive), and let $\Gamma^1_{i,k} = \{\tau_j \in hp_i | a_{j,k} = T\} \setminus \Gamma^2_{i,k}$ be the set of all other tasks in hp_i that use partition k. Then $N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\})$ is upper bounded as follows:

$$N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\}) = \sum_{\forall \tau_j \in \Gamma^1_{i,k}} I_j + \sum_{\forall \tau_j \in \Gamma^2_{i,k}} 2 \cdot I_j + a_{i,k}. \tag{8}$$

Similarly, $N_{rl}(\{a_{j,k}\}, \{I_j | \tau_j \in hp_i\})$ can be computed as the number of preemptions in the busy interval by tasks using partition k:

$$N_{rl}(\{a_{j,k}\}, \{I_j | \tau_j \in hp_i\}) = \sum_{\forall \tau_j \in \Gamma^2_{i,k}} I_j. \tag{9}$$

Note that in the case of N_{ft}, the bound above does not take $noleak_k$ into account. Hence, in the following Section 5.1 we first show how to compute an exact bound (assuming no knowledge of tasks' arrival times) $N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\})$ based on a SMT formulation. Since the complexity of the algorithm is exponential, in Section 5.2 we then show how to derive a safe bound based on a min-cost flow graph problem that can be solved in polynomial time in the number of tasks N. We will then show through simulations in Section 8.1.2 that the graph bound is a good approximation of the exact bound.

5.1 Exact FT Bound

We created an SMT formulation of the exact FT bound problem using the Z3 SMT solver [16]. Note that since in this section and the next one we focus on computing the number of FT instances for a single partition, for simplicity we drop the index k and use $noleak$ rather than

$noleak_k$. The input to this problem is the $noleak$ matrix, the number of jobs for each higher priority task, and whether each higher priority task is preemptive or not. The problem is formulated by creating a set of variables for each step, where a step is a job start or a job end. The set of variables at each step includes (1) an integer for the number of flushes so far, (2) a vector of booleans whether each task is running, (3) a vector of booleans for whether each task is in memory, (4) a vector of integers for the number of jobs remaining to be started for each task, and (5) the transition type taken to get to the next step (encoded as an integer). At each step, a transition occurs, subject to precondition and postcondition constraints. Possible transitions include, for each task τ_i, the task can start, for each task τ_i and lower priority task τ_j (including an idle task), τ_i can end and τ_j resumes and an FT occurs, or τ_i can end and τ_j resumes and no FT occurs.

Conditions are then placed on both the precondition at each step and the postcondition at the subsequent step. For example, a task τ_i is allowed to start only if the jobs left to start for that task is greater than zero (precondition), and then in the subsequent step the number of jobs left to start will be one less than in the previous step (postcondition). Another example is that task τ_i can end and task τ_j resume with a FT only if τ_i is running, and no higher priority tasks are running, and τ_j is the next highest priority task, and τ_j is running, and there is some task in memory for which τ_j requires a flush and so on. Some of the postconditions in this case would be that only task τ_j is in memory (because of the FT that has occurred), and that the number of flushes so far is one greater than the in the previous step.

The SMT solver looks for a model where, at the last step, the number of flushes so far is above a threshold. This threshold is iteratively increased until the maximum is exceeded, after which the constraints are unsatisfiable. The last satisfiable model obtained gives an ordering of events that produces that number of flushes N_{ft}.

An upper bound on the number of schedules Z3 checks can be given by noticing that at each scheduling event (job start or end, *i.e.,* a context switch), at most a single event (such as a task starting) is possible for each task in the busy interval. This means that runtime of the exact bound approach for a specific task under analysis τ_i is upper-bounded by $\mathcal{O}\big((CS_i)^{|hep_i|}\big)$, where CS_i is the number of context-switches and $|hep_i|$ is the number of higher or equal priority tasks. While this bound is exponential, solutions for task sets consisting of six or seven higher priority tasks could typically be found in a few minutes. We used this ideal bound to compare with the graph bound in order to evaluate the pessimism of the faster method.

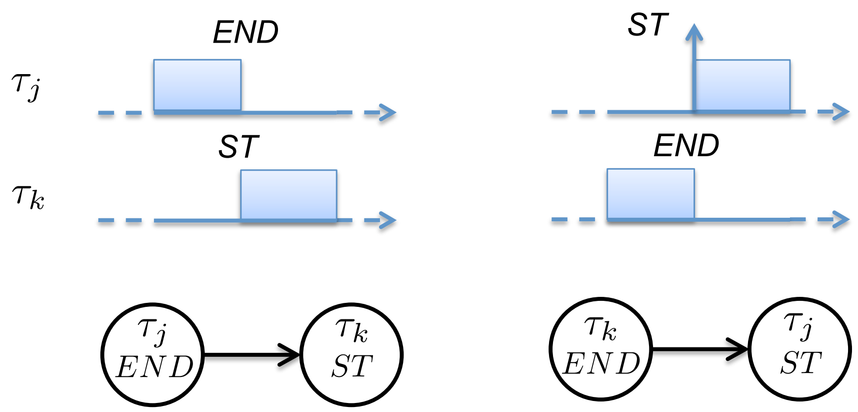


Figure 5.1: Flow Graph Intuition: Context-Switch at Job End. One unit of flow is exchanged between a job that finishes (END) and one that starts (ST) executing. Note any priority relationship is valid since a higher priority task τ_j could arrive at the same time τ_k finishes (right side of figure).

5.2 Approximated FT Bound

We now show how to compute a fast bound to $N_{ft}(noleak_k, \{I_j|\tau_j \in hp_i\})$ using a min-cost flow graph formulation, as defined below.

Definition 3 (Min-cost flow problem)**.** *Let (V, E) be a flow network, i.e., a directed graph where V is the set of vertices and E the set of directed edges of the form $e = (v \to v', u, a)$, where $v, v' \in V$, $u > 0$ is the capacity of the edge and a is its cost. Let $source \in V$ be the source vertex, producing an amount of flow $F > 0$, and let $sink \in V$ be the sink vertex, consuming an amount of flow equal to F. Finally, let $f(e)$ be the flow on edge e. Then the min-cost flow problem is to minimize the total cost $\bar{A}$ of the flow:*

$$\bar{A} = \sum_{\forall e=(v\to v',u,a)\in E} f(e) \cdot a, \tag{10}$$

subject to the constraints:

- *capacity constraint: $\forall e = (v \to v', u, a) \in E : f(e) \le u$;*
- *flow conservation: $\forall v \in V : \sum_{\forall e=(v\to v',u,a)\in E} f(e) - \sum_{\forall e=(v'\to v,u,a)\in E} f(e) = k$, where $k = F$ for $source$, $k = -F$ for $sink$, and $k = 0$ for all other vertices.*

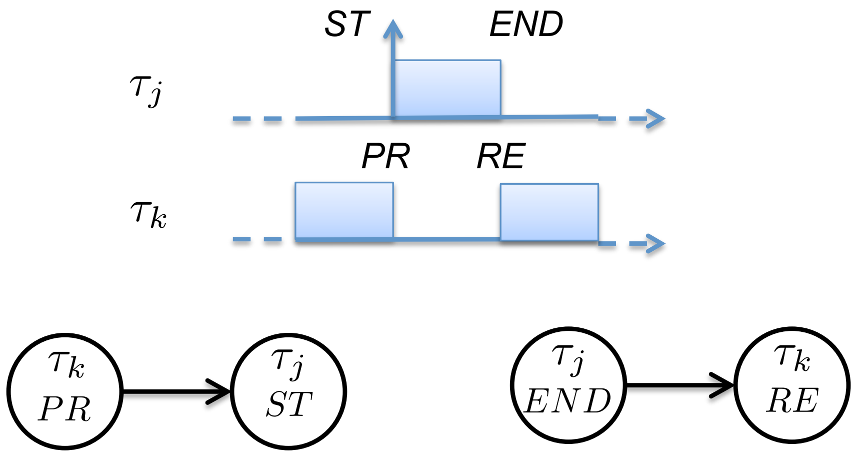


Figure 5.2: Flow Graph Intuition: Context-Switch at Preemption and Resumption. One unit of flow is exchanged between a preempted job (PR) of task τ_k and a starting job (ST) of higher priority task τ_j. In this example, when the job of τ_j ends (END), execution is returned to τ_k (RE).

We shall say that a flow assignment f for (V, E) is a *valid flow* if it satisfies all capacity and flow conservation constraints.

Before formally detailing how we construct the flow graph in Definition 4, we provide the intuition behind our method. The key idea is to encode the sequence of jobs executing during the busy interval of τ_i as a chain of vertices exchanging flows between them. The exchange of one unit of flow on an edge between a vertex and the next one in the chain represents a context-switch between jobs of the tasks represented by those vertices. We assign a cost of -1 to edges representing context-switches that result in the execution of a FT and we compute the minimum cost over any valid flow; we can show that if the resulting (negative) minimum cost is $\bar{A}$, then $-\bar{A}$ represents an upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$.

Note that a job could be context-switched out in two different situations: *(a)* the job has finished executing or *(b)* the job is preempted by a higher priority task. Similarly, a job could be context-switched into in two different situations: *(c)* the job starts executing or *(d)* the job resumes execution after having been preempted. To encode each situation, we associate four different vertices to each task τ_j: $\tau_j.END$ (job *end*ing); $\tau_j.PR$ (job *pr*eempted); $\tau_j.ST$ (job *st*arting); and $\tau_j.RE$ (job *re*suming). We can then recognize three types of context-switch and associated flow exchanges between vertices:

- A job of a task τ_j ends and a job of a task τ_k starts. Figure 5.1 shows this as an exchange of flow between $\tau_j.END$ and $\tau_k.ST$. Note: any priority relationship and any preemptivity is possible for tasks τ_j and τ_k.

- A job of a lower priority task τ_k is preempted by a job of a higher priority task τ_j that starts execution. The situation is depicted on the left side of Figure 5.2 as an exchange of flow between $\tau_k.PR$ and $\tau_j.ST$. Note that it must hold $j < k$, and furthermore $preempt_k = T$.
- A job of a higher priority task τ_j ends, and a preempted lower priority task τ_k resumes executing. This situation is depicted on the right side of Figure 5.2 as an exchange of flow between $\tau_j.END$ and $\tau_k.RE$. Similarly to the previous case, it must hold $j < k$ and $preempt_k = T$.

Note that each job must start and end once in the busy interval and, furthermore, the number of times the job is preempted must be equal to the number of times the job is resumed. Hence, for a task τ_j, the incoming flow to vertices $\tau_j.ST$ and $\tau_j.RE$ must be equal to the outgoing flow from vertices $\tau_j.END$ and $\tau_j.PR$. We can enforce such constraints by adding a vertex $\tau_j.B$ (*b*alance) that maintains the flow conservation. The resulting *vertex group* for a task $\tau_j \in hp_i$ is shown in Figures 5.3 and 5.5, where all edges between vertices in the group have a cost $a = 0$. Note that vertices $\tau_j.ST \to \tau_j.B$ and $\tau_j.B \to \tau_j.END$ have a capacity of I_j to enforce the fact that the task cannot execute more than I_j jobs in the busy interval. The vertex group for task τ_i is represented in Figures 5.4 and 5.6. In this case we omit $\tau_i.END$ since we do not need to consider any FT after the task under analysis ends; instead, the $sink$ drains $F = 1$ unit of flow from $\tau_i.B$ to represent the end of the busy interval. Similarly, we add a $source$ node to the graph that injects one unit of flow to represent the context-switch at the beginning of the busy interval.

Finally, we add edges between vertices of type END, ST, PR and RE of any two tasks $\tau_j, \tau_k \in hep_i$ based on the tasks' priorities and preemptivity, as in Figures 5.1, 5.2. We assign such edges a cost $a = -1$ if $noleak(\tau_j, \tau_k) = T$, and $a = 0$ otherwise. Note that as discussed in the examples in Section 4.2.1, before the busy interval starts, a job of any task could have executed last. Hence, when considering edges from the $source$ node to a task τ_j, we need to assign a cost $a = -1$ if there exists any task τ_k such that $noleak(\tau_k, \tau_j) = T$. Finally, we set the capacity constraint for all such vertices to $+\infty$ since the constraint on the number of jobs I_j executed in the busy interval is already enforced by capacities on the edges in the vertex group.

We can now define the graph. For ease of notation, we define $I_i = 1$ to represent the fact that only one job of the task under analysis appears in the busy interval.

Definition 4 (FT Graph). *The FT Graph for* $noleak, \{I_j | \tau_j \in hp_i\}$ *is a flow graph* (V, E) *with the following set of vertices* V*:*

1. *a* source *and a* sink *which produce/consume an amount of flow* $F = 1$*;*
2. *vertices* $\tau_i.B, \tau_i.ST$*;*

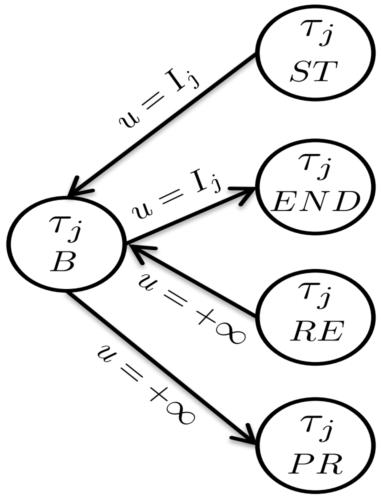


Figure 5.3: Vertex Group: Preemptive task $\tau_j \in hp_i$

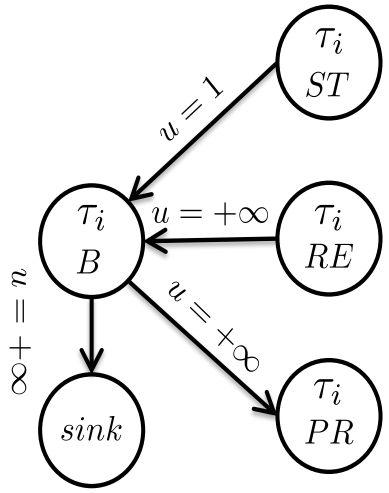


Figure 5.4: Vertex Group: Preemptive task under analysis τ_i. The sink consumes $F = 1$ units of flow.

3. *for each task $\tau_j \in hp_i$, vertices $\tau_j.B, \tau_j.ST, \tau_j.END$;*
4. *for each preemptive task $\tau_j \in hep_i$, vertices $\tau_j.RE, \tau_j.PR$;*

and the following set of directed edges E:

1. *for each task $\tau_j \in hep_i$, the following edges (if the corresponding vertices exist): $(\tau_j.ST \rightarrow \tau_j.B, I_j, 0), (\tau_j.B \rightarrow \tau_j.END, I_j, 0), (\tau_j.RE \rightarrow \tau_j.B, +\infty, 0), (\tau_j.B \rightarrow \tau_j.PR, +\infty, 0)$;*
2. *edge $(\tau_i.B \rightarrow sink, +\infty, 0)$;*
3. *for each task $\tau_j \in hep_i$, an edge $(source \rightarrow \tau_j.ST, +\infty, a)$, where $a = -1$ if there exists $\tau_k \in \Gamma, noleak(\tau_k, \tau_j) = T$, or $a = 0$ otherwise;*
4. *for each pair of tasks $\tau_j \in hp_i, \tau_k \in hep_i$, $j \neq k$, an edge $(\tau_j.END \rightarrow \tau_k.ST, +\infty, a)$, where $a = -1$ if $noleak(\tau_j, \tau_k) = T$, or $a = 0$ otherwise;*
5. *for each preemptive task $\tau_k \in hep_i$ and each task $\tau_j \in hp_i$ such that $j < k$, an edge $(\tau_k.PR \rightarrow \tau_j.ST, +\infty, a)$, where $a = -1$ if $noleak(\tau_k, \tau_j) = T$, or $a = 0$ otherwise;*
6. *for each task $\tau_j \in hp_i$ and each preemptive task $\tau_k \in hep_i$ such that $j < k$, an edge $(\tau_j.END \rightarrow \tau_k.RE, +\infty, a)$, where $a = -1$ if $noleak(\tau_j, \tau_k) = T$, or $a = 0$ otherwise.*

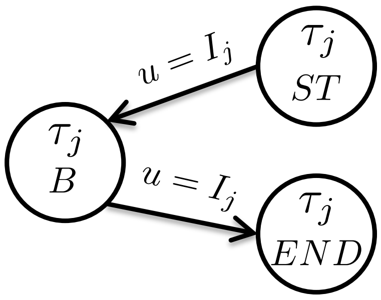


Figure 5.5: Vertex Group: Non-preemptive task $\tau_j \in hp_i$

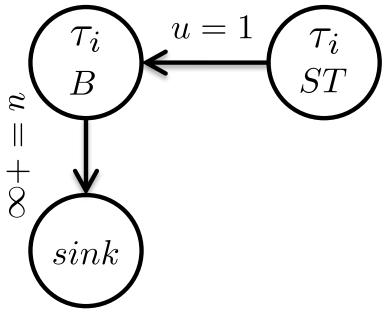


Figure 5.6: Vertex Group: Non-preemptive task under analysis τ_i. The sink consumes $F = 1$ units of flow.

Note that in Definition 4, edges of Types 1-2 are edges in the vertex groups shown in Figures 5.3-5.6, while edges of Types 3-6 are edges between vertex groups representing context-switches.

Figure 5.7 shows a complete example graph for the task set in Table 4.1, where $\tau_i = \tau_3$ is the task under analysis (note that flow assignments $\tilde{f}$ and $\hat{f}$ are used in the upcoming Theorem 1 to illustrate the proof). We use dashed edges to represent context-switches where $noleak = T$ and dotted edges for context-switches where $noleak = F$. Based on the discussed context-switch rules, the edges between vertex groups can be constructed as follows:

- an edge from *source* to every ST (Type 3 in Definition 4);
- an edge from every END to every ST (Type 4), to represent context-switches between an ending and a starting job;
- an edge from every PR to every ST of a higher priority task (Type 5; τ_3 to τ_1 and τ_2; τ_2 has no PR since it is non-preemptive) to represent context-switches where a job is preempted;
- an edge from every END to every RE of a lower priority task (Type 6; in the example, both τ_1 and τ_2 to τ_3; again, τ_2 has no RE) to represent context-switches where a job resumes from preemption.

Note that since for each τ_j, there exists another task τ_k such that $noleak(\tau_k, \tau_j) = T$, all edges from the source are dashed; following Table 4.1, the only dotted edges are from τ_1 to τ_3 and from τ_3 to τ_2.

Table 5.1 summarizes the N_{ft} bounds computed by the exact SMT formulation, the min-cost flow algorithm, and the trivial bound on the example task set, for the three preemptivity

	FT number	Trivial Bound	Graph Algorithm
Original Example	8	11	8
All Tasks Preemptive	9	11	9
All Tasks Non-Preemptive	5	6	5

Table 5.1: Example task set: FT bounds

assignments of Figure 4.1, 4.2 and 4.3. Note that in this case, the bounds computed by the min-cost flow algorithms are exact, while the trivial bound always overestimates the value of N_{ft}. Finally, the computed bound corresponds to the number of FT for the schedule reported in the figures.

Theorem 1 states that the flow algorithm is indeed always correct; the main intuition is that we can algorithmically construct a flow to match any feasible job schedule.

Theorem 1. *Let $\bar{A}$ be the min-cost for the flow graph in Definition 4. Then $-\bar{A}$ is a valid upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$.*

Proof. Let ϕ be any valid sequence of job executions in the busy interval, *i.e.,* any sequence that respects the number of interfering jobs I_j and preemptivity for each task $\tau_i \in hep_i$. [1] The proof will show that we can construct a valid flow assignment $\hat{f}$ on the graph that results in a cost $\hat{A} = -N_{ft}(\phi)$, where $N_{ft}(\phi)$ is the number of FTs required in sequence ϕ (assuming that the first job in the sequence, say of a task τ_f, suffers a FT if it is possible, i.e., there exists any task τ_k such that $noleak(\tau_k, \tau_f) = T$). But since $\bar{A}$ is the min-cost for the flow graph, it must hold $-\bar{A} \geq -\hat{A} = N_{ft}(\phi)$; hence, $-\bar{A}$ is indeed an upper bound to $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$, since it bounds the number of FT required by any valid sequence of job executions in the busy interval of τ_i.

To ease exposition, we first summarize how the rest of the proof works. We first construct a valid, initial flow assignment $\tilde{f}$ that mirrors the valid sequence ϕ by exchanging one units of flow along a chain of vertices representing the jobs in the sequence. We then obtain the desired flow assignment $\hat{f}$ by modifying $\tilde{f}$; intuitively, this is performed by removing the flow through the vertices representing certain jobs in the sequence. The resulting flow $\hat{f}$ has a cost $\hat{A} = -N_{ft}(\phi)$ by construction; we finally show that $\hat{f}$ is a valid flow.

We construct the initial flow $\tilde{f}$ by starting with an assignment $\tilde{f}(e) = 0, \forall e \in E$ and then adding flow to edges in this way: 1) we send one flow unit on the edge from *source* to $\tau_f.ST$,

[1] As stressed in Section 5, note that the definition does not consider the exact arrival times of tasks in the busy interval.

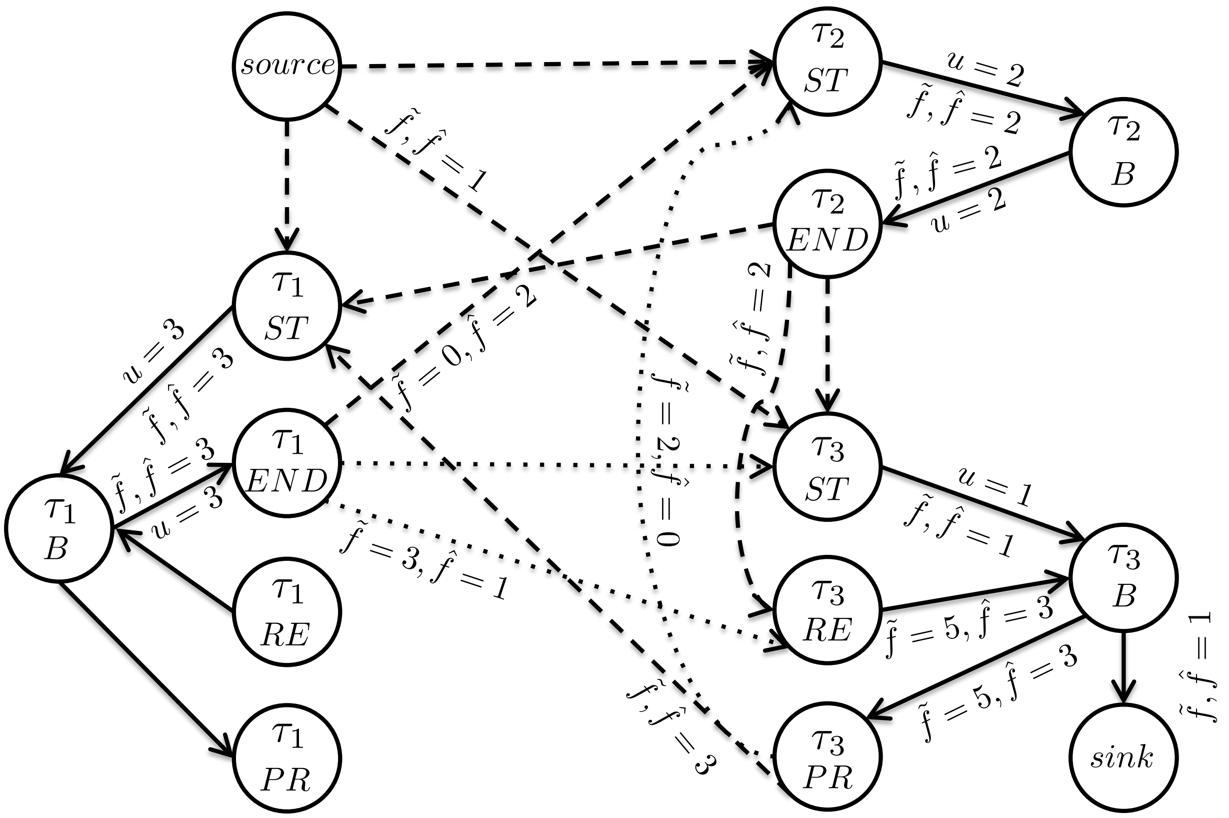


Figure 5.7: Example task set: flow graph. Where not explicitly labeled, an edge has $u = +\infty$ and $\tilde{f} = \bar{f} = 0$. Solid edges and dotted edges have cost $a = 0$. For dashed edges, $a = -1$.

the task of the first job in the sequence ϕ. 2) Next, we consider each context-switch between jobs in ϕ. Assume that execution switches from a job of a task τ_k to a job of a task τ_l. Then we add one unit of flow from either $\tau_k.END$ or $\tau_k.PR$, depending on whether the job of τ_k ends or is preempted, to either $\tau_l.ST$ or $\tau_l.RE$, depending on whether the job of τ_l starts or resumes execution after a preemption. 3) We send one unit of flow from $\tau_i.B$ to *sink*, where τ_i is the task under analysis. 4) Finally, for any task $\tau_j \in hep_i$, we send flow between vertices $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PR$ and vertex $\tau_j.B$, such that the flow conservation is met at vertices $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PR$. Example: consider Figure 5.7 and the related job sequence in Figure 4.1. Note $\tilde{f}(\tau_3.PR \to \tau_1.ST) = \tilde{f}(\tau_1.END \to \tau_3.RE) = 3$ since τ_3 is preempted three times by τ_1 and then immediately resumes. Since τ_3 is preempted two more times by τ_2, we also need to set $\tilde{f}(\tau_3.B \to \tau_3.PR) = \tilde{f}(\tau_3.RE \to \tau_3.B) = 5$ to meet flow conservation at vertices $\tau_3.PR$ and $\tau_3.RE$.

It is straightforward to see that $\tilde{f}$ is valid flow. Since at most I_j jobs of a task τ_j can be executed in ϕ, it follows that at most I_j units of flow can be sent/received by vertices $\tau_j.END$ and $\tau_j.ST$, respectively; hence, the capacity constraint on edges $\tau_j.ST \rightarrow \tau_j.B$ and $\tau_j.B \rightarrow \tau_j.END$ are respected; all other edge capacities are obviously respected since they are infinite. The flow conservation constraint at vertices *source*, *sink*, and $\tau_j.ST, \tau_j.END, \tau_j.RE, \tau_j.PS$ for all tasks is respected by construction. Finally, since the number of times a job of task τ_j starts executing in ϕ must be equal to the number of times it finishes executing, and furthermore the number of times a job of τ_j is preempted must be equal to the number of times that the job resumes from preemption, the flow conservation is also respected for $\tau_j.B$. Hence, flow $\tilde{f}$ is valid.

Unfortunately, the cost $\tilde{A}$ of the constructed flow $\tilde{f}$ might not match $-N_{ft}(\phi)$: there might exist a context-switch between jobs of tasks τ_k and τ_l in the sequence, such that τ_l requires a FT, but the corresponding edge cost for the context-switch is 0. Consider the example of Figure 4.1, where a flush is required before executing the first job of τ_2 once it preempts τ_3. In this case $noleak(\tau_3, \tau_2) = F$, so sending flow on the edge $\tau_3.PR \rightarrow \tau_2.ST$ has a cost of 0, but we still need to flush because $noleak(\tau_1, \tau_2) = T$ and τ_1 has been executed since the last FT. To solve the problem, we can intuitively obtain $\hat{f}$ from $\tilde{f}$ by removing the execution of τ_3 between τ_1 and τ_2, so that we send flow directly on the edge from $\tau_1.END$ to $\tau_2.ST$, which has a cost of -1.

More precisely, assume that a FT is required for a job of τ_l in ϕ, that the task of the job that causes the FT is τ_p (*i.e.,* the task of the latest job to execute in ϕ before the job of τ_l, such that $noleak(\tau_p, \tau_l) = T$) and that the two jobs are not executed one after the other in ϕ. Then to obtain $\hat{f}$ from $\tilde{f}$, for any such job τ_l we add one unit of flow to the edge from the corresponding vertex of τ_p (either $\tau_p.END$ or $\tau_p.PR$, based on ϕ) to the corresponding vertex of τ_l ($\tau_p.ST$ or $\tau_p.RE$) and we remove the flow that would circulate between vertices corresponding to jobs that are executed in ϕ between the jobs of τ_p and τ_l. The resulting cost $\hat{A}$ of $\hat{f}$ must be equal to $-N_{ft}(\phi)$, since by construction we send one unit of flow on an edge with $a = -1$ for each FT in ϕ. Example: in Figure 5.7, $\hat{f}(\tau_1.END \rightarrow \tau_3.RE) = 1, \hat{f}(\tau_3.PR \rightarrow \tau_2.ST) = 0, \hat{f}(\tau_1.END \rightarrow \tau_2.ST) = 2$ (rather than values of $3, 2, 0$ for $\tilde{f}$), since for each of the two jobs of τ_2 we need to send flow directly from $\tau_1.END$ to $\tau_2.ST$ rather than circulating flow from $\tau_1.END$ to $\tau_3.RE$ and from $\tau_3.PR$ to $\tau_2.ST$.

We finally show that $\hat{f}$ is still a valid flow. First note that removing one unit of flow circulating through a task τ_j cannot violate graph constraints for τ_j itself: reducing the amount of flow cannot violate a capacity constraint, and since we are removing both one unit of incoming flow from either $\tau_j.ST$ or $\tau_j.RE$, and one unit of outgoing flow from either $\tau_j.END$ or $\tau_j.PR$, the flow conservation at $\tau_j.B$ is still respected. It remains to show that we can add one unit of flow to the edge from the vertex of τ_p to the vertex of τ_l; this is not trivial since the corresponding edge might not exist in the graph. If the job of τ_p sends flow from $\tau_p.END$ and the job of τ_l receives flow on $\tau_l.ST$, then this is trivially true since there is an edge between the END and ST vertices

$noleak$		to				
		τ_1	τ_2	τ_3	τ_4	τ_5
from	τ_1	-	F	F	T	F
	τ_2	F	-	T	F	F
	τ_3	T	F	-	F	F
	τ_4	F	T	F	-	F
	τ_5	F	F	F	F	-

Table 5.2: Example non-tight task set: *noleak* relation. Tasks indexed in inverse priority order; τ_5: task under analysis. $I_j = 1$ for all tasks in hep_5. τ_3 is the only preemptive task.

of any two tasks. Therefore, consider the two following remaining cases:

Case 1: the job of τ_p sends flow from $\tau_p.PR$. Since the considered job is the last of τ_p to execute before the one of τ_l, it follows that τ_l is preempting τ_p in the schedule of ϕ. Hence, τ_l must be higher priority than τ_p, and furthermore the considered job of τ_l must be starting execution (rather than resuming from preemption), thus it must be receiving flow to $\tau_l.ST$. Therefore, we can add one unit of flow to the edge from $\tau_p.PR$ to $\tau_l.ST$, which exists in the graph.

Case 2: the job of τ_l receives flow on $\tau_l.RE$. This case is specular to the previous one, in the sense that τ_p must be preempting τ_l. Therefore, we can add one unit of flow to the edge from $\tau_p.END$ to $\tau_l.RE$, which exists in the graph. □

Graph Bound Tightness: Note that Theorem 1 only shows that the resulting bound is safe. As a matter of fact, there are task sets where the computed bound is not tight; an example is shown in Table 5.2. Solving the flow graph results in a value $N_{ft} = 5$, for the implied schedule shown in Figure 5.8. Note that this schedule satisfies the constraints of the graph, since each direct preemption and resumption (PR to ST and END to RE events) satisfies priority ordering. Nevertheless, this schedule is invalid, since lower-priority task τ_4 is indirectly preempting task τ_3. One possible valid worst-case schedule is shown in Figure 5.8, where the number of FTs is equal to 4; in fact, in this case it is easy to see that there is no valid schedule that results in $N_{ft} > 4$, since for each task $\tau_j \in hep_5$, there is a unique task τ_k such that $noleak(\tau_k, \tau_j) = T$.

Graph Bound Computational Complexity: Orlin's algorithm [35] for the min-cost flow problem has a complexity of $O(|E|^2)$, where $|E|$ is the number of edges in the graph. The number of edges based on Definition 4 is $O(N^2)$ in the number of tasks in Γ, since we need a fixed number of edges between any two tasks in hp_i. Hence, the overall complexity of deriving the graph bound is $O(N^4)$.

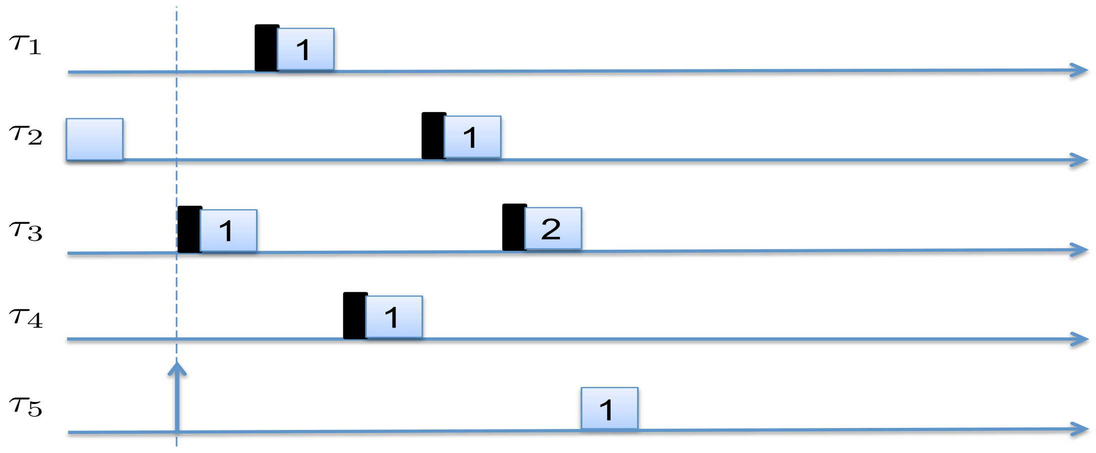


Figure 5.8: Example non-tight task set: invalid schedule implied by the FT Graph.

5.3 Optimal Preemptivity Assignment

Based on the schedulability analysis in Section 5, Algorithm 1 details how to assign preemptivity $preempt_i$ to every task $\tau_i \in \Gamma$. The algorithm is optimal, in the sense that if there exists any $preempt_i$ assignment that makes the task set schedulable according to our analysis, then Algorithm 1 will find one such assignment.

The algorithm iterates on all tasks starting from the highest priority task τ_1 to the lowest priority task τ_N. At each step, the algorithm tries to determine if the current task τ_i can be executed non-preemptively; it does so by checking that the blocking time that executing τ_i non-preemptively would cause on each higher priority task τ_j would not make τ_j unschedulable (Line 2), by checking that $\bar{c}_i - 1$ is not greater than the slack Δ_j of τ_j, computed assuming zero blocking time (Line 7, since we do not know if any lower priority task is non-preemptive this point). The main intuition is that if τ_i can be executed non-preemptively, then doing so is convenient; setting $preempt_i = F$ can potentially reduce the number of FT suffered by τ_i and lower priority task (Lemma 1), and furthermore reduce the number of jobs interfering with τ_i (Lemma 3). Based on the lemmas, Theorem 2 then states that Algorithm 1 is optimal.

Lemma 1. *Consider the bound on $N_{ft}(noleak_k, \{I_j | \tau_j \in hp_i\})$ computed by either the trivial, graph or exact algorithm, and let $\tau_j \in hep_i$ be a non-preemptive task. Changing τ_j to execute preemptively results in a bound on N_{ft} that is no less than the original one.*

Proof. In the case of the trivial bound, the proof follows immediately from Equation 8, since changing τ_j to execute preemptively cannot reduce the set $\Gamma^2_{i,k}$.

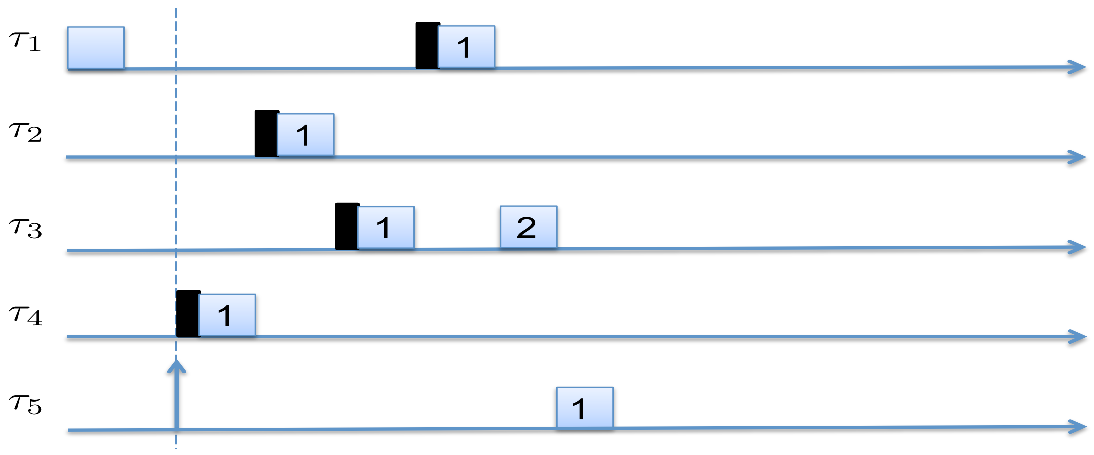


Figure 5.9: Example non-tight task set: valid worst-case schedule.

For the graph bound case, it suffices to note that if τ_j is executed preemptively, additional vertices and edges are added to the min-cost graph, but none are removed. Hence, any valid flow when τ_j is non-preemptive is also a valid flow when τ_j is preemptive, which implies that the resulting bound for N_{ft} cannot decrease.

Finally, in the case of the exact bound, notice that any task ordering that is valid for the non-preemptive case is also valid in the preemptive case: since we make no assumption on the exact arrival time of jobs, if a job of task τ_j with $j < l$ follows a job of τ_l rather then preempting it, we can simply assume that the job of τ_j arrives immediately after the one of τ_l finishes executing. Again, this implies that the preemptive case cannot result in a lower N_{ft} number, concluding the proof. $\square$

Lemma 2. *Consider the bound on $N_{rl}(\{a_{j,k}\}, \{I_j|\tau_j \in hp_i\})$ computed by Equation 9, and let $\tau_j \in hep_i$ be a non-preemptive task. Changing τ_j to execute preemptively results in a bound on N_{rl} that is no less than the original one.*

Proof. As in the proof of Lemma 1, it suffices to notice that changing τ_j to execute preemptively cannot reduce the set $\Gamma^2_{i,k}$. $\square$

Lemma 3. *The slack Δ_i computed according to Equation 7 for the case when τ_i is non-preemptive cannot be less than the slack when τ_i is preemptive.*

Proof. According to Lemmas 1 and 2, when τ_i is non-preemptive the number N_{ft} of FTs and the number N_{rl} of reloads suffered by the task are each less than or equal to the corresponding numbers for the preemptive case. Furthermore, note that the values I_j computed according to

Algorithm 1 Preemptivity Assignment

```
1:  for $i = 1 \dots N$ do
2:      if $\forall j = 1 \dots i-1 : \bar{c}_i - 1 \leq \Delta_j$ then
3:          $preempt_i \leftarrow F$
4:      else
5:          $preempt_i \leftarrow T$
6:      end if
7:      Compute $\Delta_i$ based on Equation 7 using $B_i = 0$
8:      if $\Delta_i < 0$ then
9:          Return FAIL
10:     end if
11: end for
12: Return SUCCESS
```

Equation 3 in the non-preemptive case are similarly less than or equal to the values computed according to Equation 4. Therefore, based on Equation 7 the slack with $preempt_i = F$ is greater than or equal to the slack with $preempt_i = T$. □

Theorem 2. *The preemptivity assignment of Algorithm 1 is optimal for schedulability analysis based on the slack time computation in Equation 7, where $N_{ft}(noleak, \{I_j | \tau_j \in hp_i\})$ is derived according to either the trivial, exact or graph bound.*

Proof. The proof proceeds by induction on the task index i in Algorithm 1. In particular, we prove the following property: if Algorithm 1 sets $preempt_i = T$, then there exists no preemptivity assignment for $\{\tau_1, \dots, \tau_{i-1}\}$ such that $preempt_i = F$ and tasks $\{\tau_1, \dots, \tau_{i-1}\}$ are schedulable. In other words, the algorithm always assigns a task τ_i to execute non-preemptivily if it is feasible to do so. Based on Lemma 3, this implies that the algorithm is optimal.

Base case: the property is trivial for $i = 1$, since τ_1 is always assigned to execute non-preemptively.

Inductive step: by contradiction, assume that the algorithm assigns $preempt_i = T$, but there exists a preemptivity assignment $\{\overline{preempt}_1, \dots, \overline{preempt}_{i-1}\}$ such that $preempt_i = F$ and $\{\tau_1, \dots, \tau_{i-1}\}$ are schedulable. Let $\{preempt_1, \dots, preempt_{i-1}\}$ be the preemptivity assignment that the algorithm picked at previous steps $1, \dots, i-1$. Since the algorithm assigns $preempt_i = T$, then tasks $\{\tau_1, \dots, \tau_{i-1}\}$ cannot be schedulable with assignment $\{preempt_1, \dots, preempt_{i-1}, preempt_i = F\}$: Line 2 must have evaluated to false, meaning that making τ_i non-preemptive would cause at least one task in $\{\tau_1, \dots, \tau_{i-1}\}$ to miss its deadline due to excessive blocking time. Therefore, it follows that the two assignments $\{\overline{preempt}_1, \ldots, \overline{preempt}_{i-1}\}$ and $\{preempt_1, \ldots, preempt_{i-1}\}$ must be different. We now have two cases:

Case 1: there exists at least one task $\tau_k \in hp_i$ such that $\overline{preempt}_k = F$ and $preempt_k = T$. This contradicts the inductive hypothesis: since τ_k is feasible with the assignment $\{\overline{preempt}_1, \ldots, \overline{preempt}_k = F\}$, at step k the algorithm must have assigned $preempt_k = F$.

Case 2: for all tasks $\tau_k \in hp_i$ such that $\overline{preempt}_k \neq preempt_k$, it holds $\overline{preempt}_k = T$ and $preempt_k = F$. This contradicts the assumption that $\{\tau_1, \ldots, \tau_{i-1}\}$ are schedulable under $\{\overline{preempt}_1, \ldots, \overline{preempt}_{i-1}, preempt_i = F\}$ but not under $\{preempt_1, \ldots, preempt_{i-1}, preempt_i = F\}$: based on Lemmas 1, 2, 3, the slack of a task cannot decrease when either the task itself or a higher priority task is executed non-preemptively.

Since neither case is possible, the inductive step follows.

□

Chapter 6

Partition Assignment

Following the resource partitioning mechanism introduced in Chapter 4, in this chapter we discuss how to assign partitions to tasks by introducing an optimization algorithm for the resource partitioning problem. Due to the complexity of the problem, we decided to pursue an heuristic algorithm based on a meta-optimization strategy, namely, Genetic Algorithms (GA). We first provide an overview of the heuristic framework. Then, we describe the genetic algorithm solution, and finally we provide algorithms of different operators and their relevant details.

6.1 Heuristic Scheme

In order to develop a heuristic for our resource partitioning problem, we need to explore the problem to find special properties which help us with a near-to-optimal solution. We tried to exploit critical resource requirements for each task, in addition to tasks interferences.

In order to develop our heuristic, two important properties are as follows: (1) for each task, we try to determine an assigned number of partitions which is a threshold, meaning that after the threshold, there is little improvement in task computation time. Thus, we used that specific critical partition number as an important property to develop our heuristic. (2) As discussed in the previous Chapters 4 and 5, there exists security requirements for each task with regards to another task in the system, introduced as $noleak$ relation; accordingly, we extracted interferences of each task with regards to their relevant values in $noleak$ relation; the way of calculating interferences will be described in Section 6.3.

We used GA to do optimization over the problem; the overall heuristic framework is illustrated in Figure 6.1.

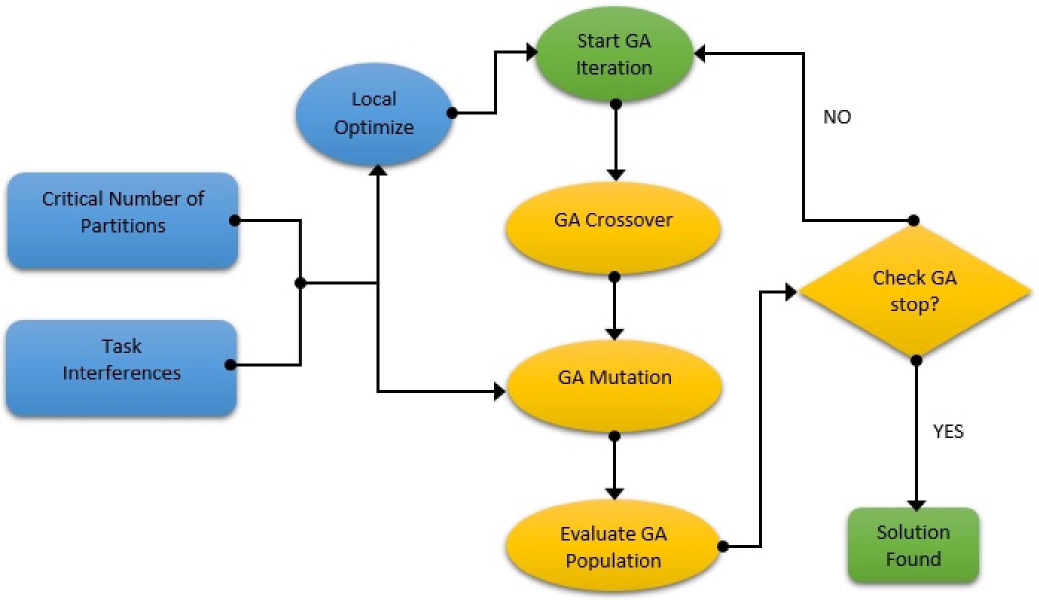


Figure 6.1: Overview of heuristic based GA algorithm.

In Figure 6.1, the task interference of each of the tasks and critical number of partitions will be defined in Section 6.3. Both of them are used to calculate the greedy local optimizer. In addition, those two properties are also used in the GA mutation operator, described in the Section 6.5. The local optimizer computes an initial approximate solution that is used to initialize the population of the genetic algorithm.

6.2 Genetic Algorithm

The partitioning GA is listed in Algorithm 2. According to the algorithm, input a_{NK} is an initial matrix $\{a_{i,k}\}$ of assigned partitions for each task. This value is generated by the local optimizer and used for computing initial population.

In the algorithm, $P(g)$ represents the current population, g is the generation number, $C_c(g)$ is the set of selected chromosomes created by the crossover operator, $C_m(g)$ is the set of chromosomes derived by the mutation operator, and *terminating condition* is the condition which stops the GA from iterating over the population. Our selected values for both of population and generations is equal to 5. In this work, the terminating condition is either the maximum number of generations, or is based on the improvement in the fitness value, which is the cumulative value of

Algorithm 2 Cache Partitioning GA

```
    Input:$\{a_{NK}\}$
    Output:$\{a_{NK}\}$
1: $g \leftarrow 0$;
2: initialize $P(g)$ using locally optimized $a_{NK}$;
3: while not terminating condition do
4:     create $C_c(g)$ from $P(g)$ by crossover;
5:     create $C_m(g)$ from $P(g)$ by mutation;
6:     $C(g) = C_c(g) \cup C_m(g)$;
7:     evaluate $C(g)$;
8:     $g \leftarrow g + 1$;
9: end while
```

all the response times, whether the tasks are schedulable or not; for unschedulable task sets, we might not have converged response times for some of the tasks, thus we considered them a very large value than any response time, which is 100000000 ms in our GA configuration. Here, no significant improvement means that the change in the fitness function for the best chromosome is less than or equal to a predefined threshold, that we configured to be equal to 1e-6. After terminating, the algorithm then outputs the partition assignment a_{NK} with the best fitness.

6.3 Greedy Local Optimizer

The greedy local optimizer, shown in Algorithm 3, computes an initial partition allocation by ordering tasks based on their caused interference and allocating to each task the set of partitions with the smallest number of already-assigned tasks. More in details, $ProtectionRank_i$ is calculated for each task τ_i as follows:

$$ProtectionRank_i = \sum_{j=1}^{N} noleak(\tau_i, \tau_j), \qquad (11)$$

i.e., it represents the interference caused by the task in terms of the number of other tasks for which a FT is required. As discussed in Section 6.1, $PartitionNumber_i$ represents the critical partition number for task τ_i; having more than the critical number of partitions does not significantly decrement the computation time of the task. More in details:

$$PartitionNumber_i = \{a | c_i(a) - c_i(a+1) < \epsilon\}, \qquad (12)$$

Algorithm 3 Greedy Local Optimizer

```
    Input:{$N$,$K$,noleak}
    Output:{$a_{NK}$}
 1: calculate $\forall i : ProtectionRank_i$;
 2: calculate $\forall i : PartitionNumber_i$;
 3: $\forall i, k : a_{NK}(i, k) = F$;
 4: sort tasks by non-increasing values of $ProtectionRank$;
 5: for $i = 1 \dots N$ do
 6:     $SortedRanks \leftarrow$ RankPartition($a_{NK}$,N,$K$);
 7:     for $k = 1 \dots PartitionNumber_i$ do
 8:         $a_{NK}(i, SortedRanks_k) \leftarrow T$;
 9:     end for
10: end for
```

where ϵ is a small value (configured to be equal to 1ms in our evaluation).

After tasks are sorted by decreasing interference values, partitions are sorted using the results of the $RankPartition$ procedure, described in Algorithm 4. This procedure calculates the partition ranks based on their eligibility to be assigned to a task. The eligibility is determined in terms of the number of tasks which are already assigned to a given partition. After calculations, partitions are sorted non-decreasingly. Therefore, partitions with the lowest numbers of already assigned tasks are chosen to be allocated to the requested tasks.

Algorithm 4 RankPartition

```
    Input:{$a_{NK}$,$N$,$K$}
    Output:{SortedRanks}
 1: tmpRank $\leftarrow$ 0;
 2: for $k = 1 \dots K$ do
 3:     $tmpRank_k \leftarrow \sum_{i=1 \dots N} a_{NK}(i, k)$
 4: end for
 5: $SortedRanks \leftarrow$ list of partition indexes sorted by non-decreasing values of $tmpRank$;
```

Based on our evaluation in Chapter 8, the local optimizer is effective in providing a better choice of partitions for each of the tasks, which significantly helps in decreasing the number of flushes because there would be lower interferences between the tasks which are supposed to occupy the same partition. As overviewed in Figure 6.1, the initial population calculated by the heuristic will be fed to GA. By feeding such initial population of local optimized values, the chance of finding the best solution can be increased. Alternatively, in Chapter 8 we also evaluated the performance of the local optimizer alone without running the further GA optimization step.

6.4 Crossover Operator

The basic idea of a crossover operator is to generate a child (*IndTaskPartition3*) from two parents (*IndTaskPartition1* and *IndTaskPartition2*), using some randomized combination.

Algorithm 5 Crossover Operator

```
    Input:{IndTaskPartition1,IndTaskPartition2}
    Output:{IndTaskPartition3}
 1: for $i = 1 \dots N$ do
 2:     $SelectionVector \leftarrow$ random binary string with $K$ length;
 3:     for $k = 1 \dots K$ do
 4:         if $SelectionVector_k = 1$ then
 5:             $IndTaskPartition3_{i,k} \leftarrow IndTaskPartition1_{i,k}$;
 6:         else
 7:             $IndTaskPartition3_{i,k} \leftarrow IndTaskPartition2_{i,k}$;
 8:         end if
 9:     end for
10: end for
```

According to Algorithm 5, we have a partition matrix of $N \cdot K$ size. Two for loops are used to generate a randomized bit string corresponding with each of the tasks and evaluate each task against each partition, in order to decide to copy a bit from either parent; if the corresponding bit of the string is 1, then the bit from the first parent will be chosen for the new child, otherwise the second parent's bit will be chosen. The $SelectionVector$ is a binary vector of generated independent pseudorandom values using a uniform distribution.

6.5 Mutation Operator

Finally, Algorithm 6 shows the employed mutation operation. The threshold numbers of required partitions for each task, namely $PartitionNumber$ array, are given as static pre-computed values as discussed in Section 6.1. The array is used in the mutation operator to determine the

Algorithm 6 Mutation Operator

```
    Input:{$a_{NK}$,PartitionNumber}
    Output:{$a_{NK}$}
1: for $i = 1 \ldots N$ do
2:     $P \leftarrow PartitionNumber_i/K$
3:     for $k = 1 \ldots K$ do
4:         with probability P, assign $T$ to $a_{NK}(i,k)$, otherwise $F$;
5:     end for
6: end for
```

probability of having the required number of partitions for each of the tasks. In fact, our used probability of allocating a partition to τ_i is equal to $PartitionNumber_i/K$. The reason that this method is exploited in mutation operator is related to the fact that GA is essentially based on randomization that on one hand will help us with finding good solution in the evolutionary way, but on the other hand might unbalance the number of assigned partitions to each of the tasks; hence, potentially increasing a task's execution time and decreasing the schedulability of the system. Also, if more than required number of partitions are assigned to a task, it might increase the overhead of resource flushing which might also cause a loss of schedulability. Therefore, this functionality of mutation operator can generally help with the drawback of the randomization.

Chapter 7

Implementation

In this chapter, we provide an overview of the platform used to implement the discussed techniques and constraints, explaining its main software and hardware components. In particular, we implemented and tested a prototype case study based on the application described in Section 3.1.

Our implementation focuses on the processor cache, which is the most easily exploitable "stateful" resource on the platform; we employ available hardware functionality to flush the entire cache content (both L1 and L2). Here, we are assuming that the covert channel can take advantage of cache line evictions, in the sense that the attacker can leak information from the victim by tracking the overhead timing that an eviction causes. The partitioning implementation is based on way-partitioning scheme for last-level cache (L2).

We selected for our implementation a Zedboard development board employing a Xilinx Zynq System-on-a-Chip (SoC). The Zynq SoC comprises a dual-core ARM Cortex-A9 processors, a PL310 cache memory controller (A9), and related peripherals. Since our research focuses on single-core systems, we used only one of the available A9 cores.

The default system configuration comprises a 512 KB, 8-way associative level 2 cache; this gives us a number of cache partitions $K = 8$, each with size 64 KB. For measurements, each core features a 32-bit clock cycle counter (called CCNT) that we used to accurately count cpu cycles in our experiments, and evaluate the performance in terms of timing.

In order to implement cache flushing and cache way partitioning in our system, we mostly used Xilinx software tools to have direct control over hardware components. Specifically, in order to do L2 cache flushing, we used functions which were implemented to invalidate and clean cache L2. L2 cache flushing′s underlying implementation is, in order: (1) disables Write-back and line fills, (2) finds the address with "XPS_L2CC" base address and "Cache Invalidate and Clean by Way" offset that is 0x07FC, (3) then writes 0x0000FFFF to that, (4) finally enables Write-back and line fills.

For cache way partitioning, there are available methods as: "lock by master", "lock by way" and "lock by line". We decided to use cache lock by way, since our platform is single-core and accessing to ways rather than lines could meet our needs to lock particular ways and have some partitioned cache resources. Unfortunately, we have not found an official Xilinx function to do cache way locking, so we wrote our code in this order: (1) all maskable interrupts and exceptions are disabled, (2) the single 32-bit lockdown register will be used to lock/unlock cache ways. The register's first eight least significant bits are represented for 8 available ways, and the rest of 24bits are reserved. By writing 1 to each bit, we lock the equivalent way. For example bit 0 is equivalent to way 0 and by setting that bit, we lock way 0. (3) After writing to the lockdown register, all the interrupts and exceptions will be enabled.

We measured a time of 340 μs for the whole cache flush. Based on such value, we estimated that it would take 340/8=42.5 μs for each cache way to be flushed, that is our definition of c_{ft} for the partitioning mechanism. The value was measured by first filling the cache with dirty (modified) cache lines; this forces the flush procedure to write back the entire cache content, which represents the worst-case timing. Hence, we assumed a similar timing for a single cache way reloading overhead, that is we set $c_{rt} = c_{ft}$.

Although, ARM Cortex-a9 provides an interesting feature called Trustzone which tries to guarantee secure access to L2 cache lines and avoiding non-secure cache evictions, it cannot protect cache memory from being a victim of complicated timing attacks such as covert channel. In fact, in terms of timing attack possibility, there is no significant difference for a system with or without Trustzone feature [3].

7.1 Hardware Components

To allow indoor experimentation, the UAV part of the platform comprises a combination of an actual aerial vehicle and a Hardware-In-the-Loop (HIL) simulator [4]. The vehicle has 3 degrees of freedom, with its actual position being fixed. The HIL component uses the vehicle′s dynamics to simulate changes in position and returns a GPS-like positional signal to the ECU. The ECU platform is based on a Xilinx FPGA using an ARM Cortex A9 processor core running at 667Mhz.

The HIL, as the tested hexacopter, was running on a Beagleboard development board. As part of our implementation, we changed the hardware for ECU to run on a Zedboard development board. In addition, ECU is running with FreeRTOS kernel rather than RTL kernel. More details on the kernel is discussed in Section 7.3.

7.2 Software Components

As discussed in Chapter 4, to demonstrate the research, an example avionics case study has been used. Figure 7.1 shows the main components of the case study [4].

The demonstrator has been implemented upon the control software, called Autopilot, of Hexacopter case study research project, developed in Real-Time Systems Lab (RESL) at the University of Waterloo [6] for emSYSCAN participants. The simulated plant, or HIL, in this case is considered as a black box and fully responsible for sending valid sensor data with response to the valid received actuation data, then system state will be calculated by the Autopilot application.

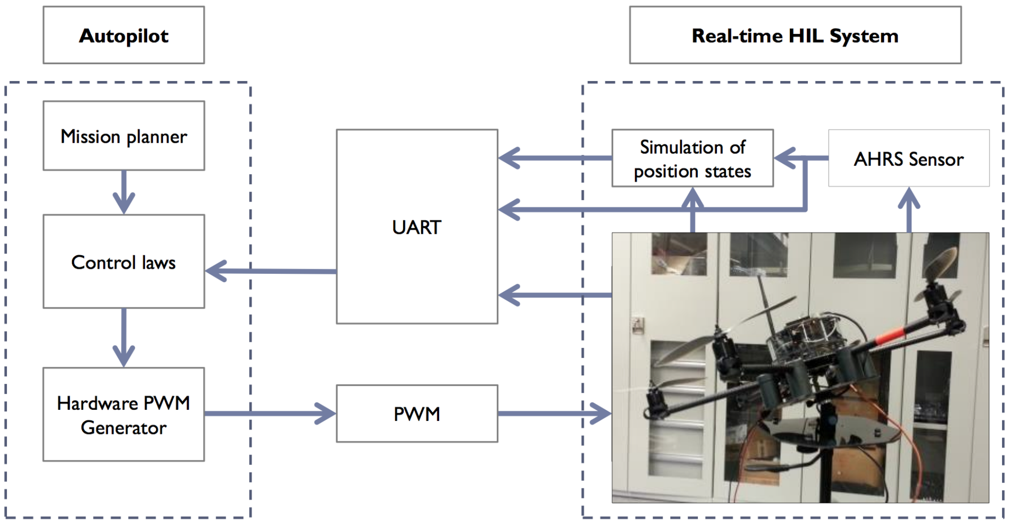


Figure 7.1: Hexacopter Components

The vehicle was tested as QNX and Real-Time Linux (RTL) implementations. Basically, the case study consists of two parts: Autopilot and Real-time HIL system, each part running on separate Beagleboards. Each BeagleBoard has OMAP3530 System-on-a-Chip (SoC) and designed with open source software development in mind. The QNX based system has been validated outdoors and the RT-Linux based system with a QNX hardware-in-the-loop (HIL) module has been in use as part of the online version of the platform. Our extended case study has focused on the RT-Linux based Autopilot. Autopilot and HIL communicate through Universal Asynchronous Receiver/Transmitter (UART) protocol. The hexacopter has been changed and also extended to meet our research requirements, which is in line with the goal of the hexacopter project, that is to enable researchers with a practical and tested platform to experiment and demonstrate their models [4]. Our changes include hardware boards, operating system and adding more real-time tasks up to Autopilot.

Table 7.1 summarizes the key timing parameters of the implemented tasks, as illustrated in Figure 3.1. We report the cumulative worst-case execution time for all control tasks because they run at the same frequency. Note that communication in the image and integrator subsystem can use non-blocking buffers to avoid long blocking times.

Task Name	Worst Case Timing(ms)	Period(ms)
Software Control Tasks	2	20
Mission Planner	0.002	100
Encryption	3	42
Image Encoding	18	42
Image I/O	1.46	42
Network Manager	0.03	10

Table 7.1: case study timing parameters

7.3 Kernel Support

We implemented the described NLF mechanism by modifying the FreeRTOS real-time kernel. FreeRTOS natively supports preemptive fixed-priority scheduling. We extended the task descriptor to include preemptability information, and modified the scheduling function to avoid rescheduling while a non-preemptive task is running. We then modified the context-switch function to determine whether a FT is required before executing a task. Our implementation is able to perform the check in constant time as long as the number of tasks in the system is less than the bit-width of the machine (32 bits for our hardware platform) by using bit-arrays encoded as integers. In details, we maintain a binary array $mustflush[i]$ which determines whether a FT is required before executing task τ_i. The array is reset to 0 whenever a FT is executed. Whenever a task τ_j is executed, we then update the array such that $mustflush[i] \leftarrow mustflush[i]|noleak(\tau_j, \tau_i)$, i.e., $mustflush[i]$ is set to 1 if executing τ_j requires a FT for τ_i. Since the array is implemented as an integer, we can perform the update in constant time using bit-wise logical operations.

To provide more details, in order to make task scheduler enable to trace security requirements, we extended TCB data structure to have a specific security requirement. This property is initialized when a task is created. This security property, similar to other TCB properties, can be globally accessed by the scheduler that indicates if the task is protected or not. In FreeRTOS kernel, we also have the capability of sending parameters while creating the task, so we can send the id of a task as a parameter at the task creation time. The parameters of each task will be stored on top the stack of that specific task. This parameter will be used to check against the given “noleak” relation in each context switch, in order to decide whether a cache flush is required or not.

Cache Level 2 Flushing is fulfilled using Xilinx library function, called Xil_L2CacheFlush function. If the byte in cacheline has cached by the Data cache, then the cache line will be flagged as modified (dirty), then the entire contents of the cacheline are written back to memory before the line is invalidated [8], so it is guaranteed that cache data will not be lost.

Since L2 cache flush is implemented in the Xilinx official software tools, it technically guarantees the correctness of its functionality. In addition, because the L2 flushing is performed by hardware output operation, called Xil_Out32(u32 OutAddress, u32 Value) function, for a 32-bit memory location by writing the specified Value to the the specified address [8], we can appreciably say that the cache flushing performs as fast as possible, that is very important for our system that cache flushing is the only significant security overhead for threatening schedulability.

Chapter 8

Evaluation

To evaluate the effectiveness of the proposed mechanisms and schedulability analysis, we performed a large number of simulations based on both our demonstrator example and sets of synthetic tasks. This evaluation is divided to two parts: (1) Section 8.1 evaluates the flushing mechanism alone, without resource partitioning. In this case, each task uses the full resource (cache). (2) Section 8.2 evaluates the combined partitioning and flushing mechanism (partitioned-aware NLF mechanism).

8.1 Cache Flushing Approach

We evaluate the effects of the flushing approach alone. First, we discuss schedulability for our described avionics case study. Then, we show simulation results based on synthetic tasks. Simulations are written in Java, using the IBM academic initiative version of CPLEX optimization studio [23] to derive the graph bound.

8.1.1 Avionics Case Study

Based on the implementation discussed in Chapter 7, we used a c_{ft} value of $340\mu s$ for our avionics case study. We then used our derived preemptability assignment and schedulability analysis to assign $preempt_i$ values and determine feasibility; note we assigned task priorities based on Rate Monotonic (RM) ordering. We tested using the trivial bound, graph bound and exact bound, as well as normal RM scheduling with no flushes. Table 8.1 shows results in terms of preemptability assignment, which is the same for all algorithms, while Table 8.2 shows response time results in terms of the maximum response time / period ratio for any task.

Sens.	Laws	Act.	MP	Net.	AES	JPEG	I/O
F	F	F	F	F	F	T	F

Table 8.1: Demonstrator: Preemptivity Assignment

Algorithm	Max Response Time Ratio	Min Period(ms)
RM (no flush)	64%	27
Exact (Z3)	75%	32
Graph	75%	32
Trivial	83%	36

Table 8.2: Demonstrator: Schedulability Results

All mechanisms result in a schedulable system, which corresponds to our observation running the demonstrator. Response time ratio for the trivial bound is appreciatively worse than for the graph bound; RM has the best schedulability but does not provide any security guarantees. We also used the analysis to determine the minimum period at which the image subsystem could be run while still remaining schedulable. Once again, results for the graph bound are better than for the trivial bound. Finally, results for the exact bound matched the graph bound, although the analysis took 7 minutes, whereas computing the graph bound took less than a second.

8.1.2 Synthetic Results

Table 8.3 summarizes the parameters used for the generation of the synthetic task sets used in our evaluation. We generated 3000 task sets that fall into each utilization group, $[0.02+0.1\cdot i, 0.08+0.1\cdot i]$ for $i = 0, \ldots, 9$, *i.e.,* 300 sets per group. The base utilization of a task set is defined as the total sum of the task utilizations. Each group is generated with the following three different settings; the first 100 sets with the probability of $noleak(\tau_i, \tau_j)$ for any pair of tasks being 10%, the next 100 with the probability of 20%, and the last 100 sets with a probability of 50%.

Each input instance consists of $[5, 20]$ tasks, each τ_i of which has a period $p_i \in [5ms, 100ms]$ and an execution time $c_i \in [0.3ms, 3ms]$; note that since we do not employ partitioning, there is no need to specify the execution time as a function of number of partitions. The deadline of each task is equal to its period, *i.e.,* $d_i = p_i$. Task priorities are assigned according to the Rate Monotonic (RM) algorithm [29]. Except for the last set of experiments, the preemptiveness of

Parameter	Value
Number of tasks, N	$[5, 20]$
Task period, p_i	$[5ms, 100ms]$
Task execution time, c_i	$[0.3ms, 3ms]$
FT overhead, c_{ft}	$\{0.1ms, 0.5ms\}$

Table 8.3: Experimental Parameters.

each task is assigned in a random manner. These task parameters are in line with the values measured on the demonstrator platform where a typical FT execution time was $0.34ms$ (Section 8.1.1).

Evaluation of FT bound

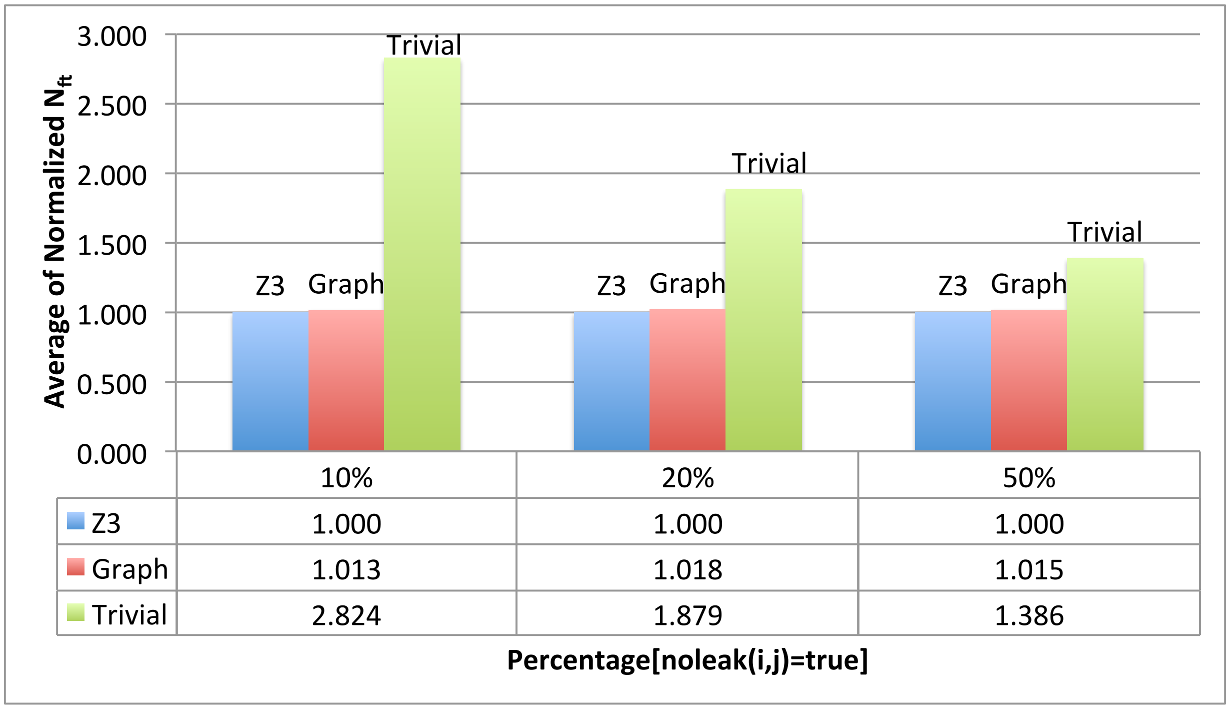
We first evaluate our (approximate) FT bound by comparing it with the exact bound found by the SMT solver as well as the trivial bound. For each task set, we first calculate the worst-case response time of the lowest-priority task using the graph-based analysis from Section 5. Then, we feed the information on the number of higher priority jobs and the no-leak matrix to an SMT solver so that it can calculate the exact bound on the number of FT invocations. The results are based on the calculation of the number of flushes that occur during the worst-case busy period of the lowest-priority task.

Figure 8.1 shows the geometric mean of the number of flushes (that occur during the lowest-priority task's busy period) found by the graph-based approximation (*Graph*) and by the trivial bound (*Trivial*) normalized to the exact bound, *Z3*. As we can see from the results, our graph-based approximation method computes a *tight bound on the number of flushes irrespective of the percentage of no-leak relation*. On average, this number is 1% or 2% more compared to the exact bound. On the other hand, the trivial bound is often three times the exact bound. Note that the trivial bound tends to be more pessimistic when the probability of $noleak(\tau_i, \tau_j) = T$ is low, since it is only based on the number of context-switches and ignores the no-leak relation itself.

Evaluation of Schedulability

We also evaluated the effects of the graph-based and trivial bounds on the schedulability of task sets by varying the flush times, c_{ft}, as well as the no-leak percentage. The results of these experiments are shown in Figures 8.2, 8.3 and 8.4.

The X-axis plots the utilization bins for the experiments while the Y-axis represents the total percentage of schedulable instances for task sets for each bin (100 tasks per bin). The graphs



	10%	20%	50%
🟦 Z3	1.000	1.000	1.000
🟥 Graph	1.013	1.018	1.015
🟩 Trivial	2.824	1.879	1.386

Figure 8.1: Flush Task Bounds as calculated by the SMT solver, the Graph-based Analysis and the Trivial Analysis

represent the schedulability of *(a)* no flushes (RM), *(b)* the number of flushes computed by the graph-based approximation (Graph) and *(c)* the trivial bound for the number of flushes (Trivial). The graphs also show the effects of varying the FT execution times (the different values of C_{ft}) and the percentage of $noleak(\tau_i, \tau_j)$ for each experiment.

Figures 8.2 and 8.3 show the two extreme cases *i.e.,* when the difference in the schedulability between 'Graph' and 'Trivial' is the largest and the smallest. The graph-based method outperforms the trivial bounds when the no-leak percentage is low since the latter is highly pessimistic (Figure 8.1). The schedulability of 'Trivial' further decreases as the overhead for flushing, c_{ft}, increases. On the other hand, the difference becomes smaller as the no-leak percentage becomes larger (and thus the pessimism of 'Trivial' becomes smaller) and the flushing time becomes shorter. Figure 8.3 shows the case when the performance of the graph-based method is similar to that of 'Trivial'. These figures also show that *(i)* schedulability drops as c_{ft} increases because of the extended blocking times due to longer flushes and, *(ii)* schedulability drops as the no-leak percentage increases because of the greater chance of a flush occurring during a context switch.

Figure 8.4 shows behavior that lies between the two extreme cases presented above. It might even be a 'typical' case where 20% of the tasks have the noleak relationship between them to

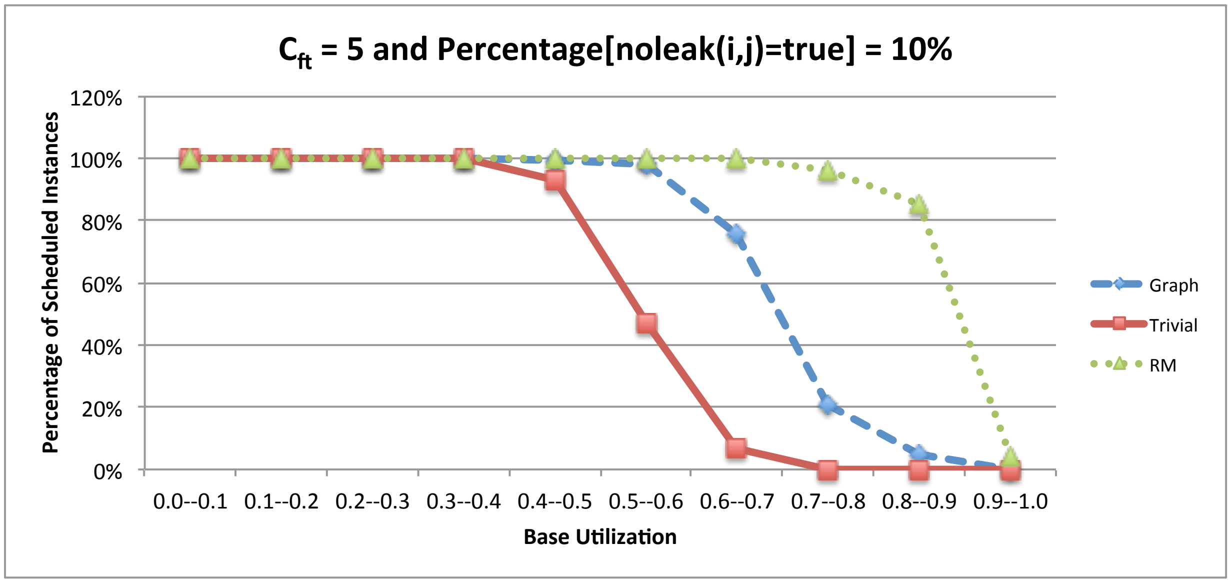


Figure 8.2: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 5, noleak = 10%]

be true and the flush time overheads are around $0.5ms$. As expected, our graph-analysis based bounds outperform the trivial bounds by being able to schedule more instances of task sets.

Preemptability Assignment

Finally, we performed synthetic experiments to evaluate the effect of task preemptability. The results are shown in Figure 8.5, where we plot the total percentage of schedulable task sets as a function of the utilization bin, similar to the previous experiment. The three graphs represent the optimal assignment using Algorithm 1 (Optimal), the case where all tasks are either preemptive (P) or non-preemptive (NP), and using random assignment (Random). Note that while NP performs better than P on average for this specific parameter assignment, in general the two approaches are incomparable, i.e., there are task sets which are schedulable by P but not NP and vice-versa. Being optimal, the assignment generated by Algorithm 1 performs better than either.

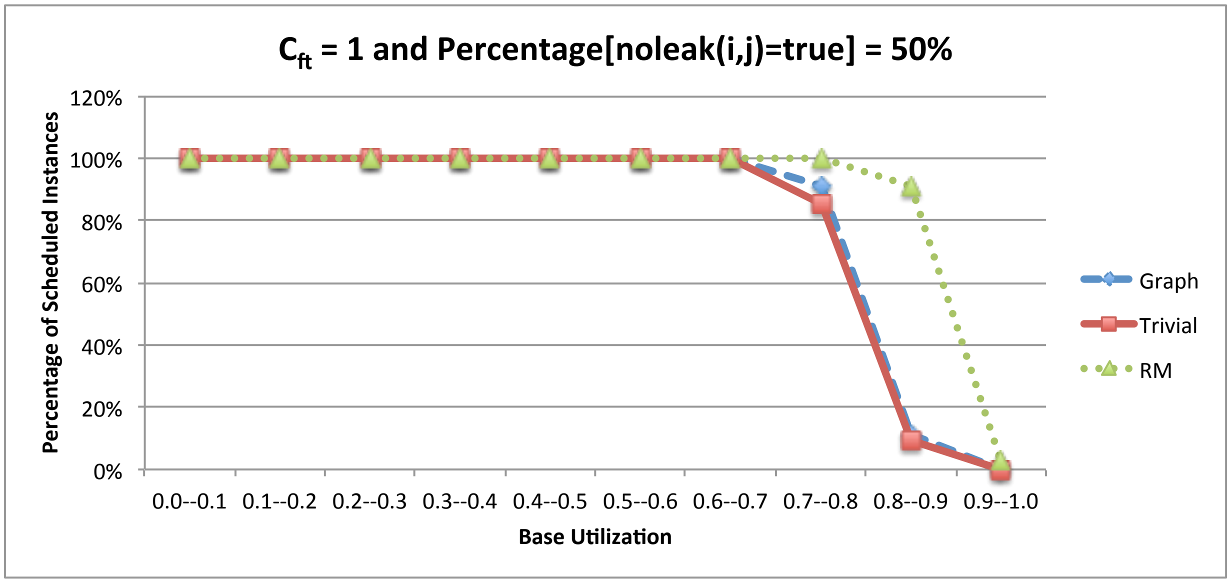


Figure 8.3: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 1, noleak = 50%]

8.2 Cache Partitioning Approach

We next evaluate the combined partitioning and flushing approach discussed in Section 4.3. The partitioning optimization framework is run in Matlab [31], whereas the simulations are integrated with Java codes and CPLEX as discussed in the previous section.

8.2.1 Cache-Aware Worst-Case Execution Time

To properly evaluate our partitioning mechanism, it is essential to construct a reasonable model for the worst-case execution time $c_i(k_i)$ of each task as a function of the number of assigned partitions k_i. In the case of cache resource, reducing the number of assigned partitions, and hence the size of usable cache, can increase the number of capacity misses suffered by the task. In turn, this leads to an increase in its worst-case execution time. While the effect can be directly observed by running benchmarks on our implemented platform, to conduct simulations on synthetic tasks we need an analytical model to randomly generate realistic $c_i(k_i)$ curves for a variety of different tasks.

We found the analytical model described in [46] to be close to our needs; the model has been previously employed in related work on cache-aware real-time scheduling [13]. More in details

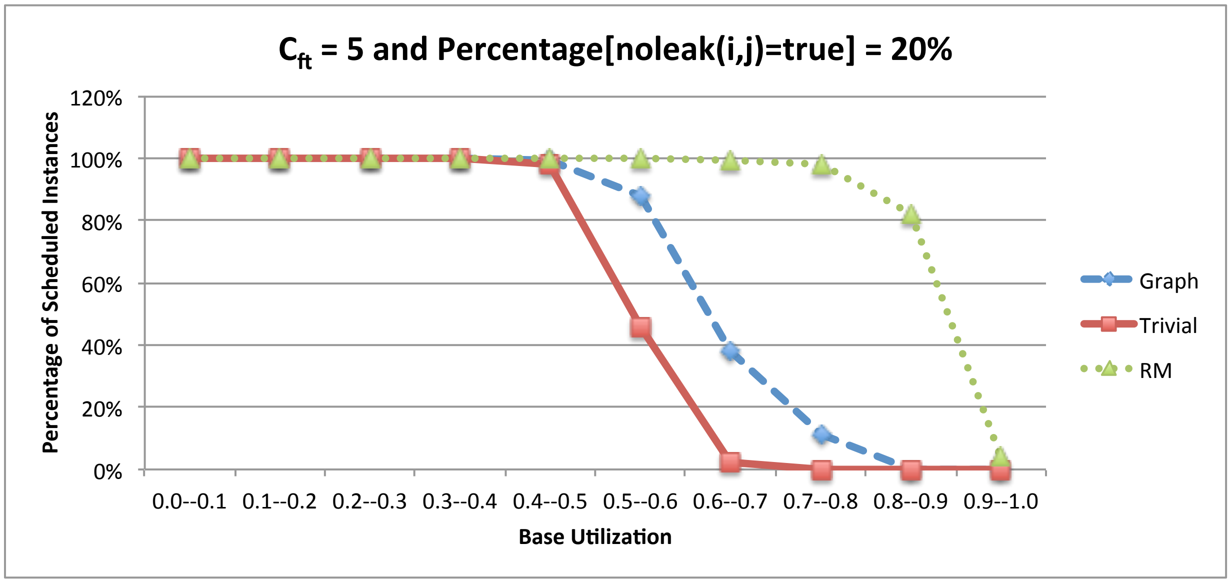


Figure 8.4: Schedulability of Task Sets: Graph-based Bounds vs. Trivial Bounds [FT Overheads = 5, noleak = 20%]

and similar to [13], we use $MissRate(k)$ function that is the L2 *cache miss rate* which is in the range of $[0, 1]$ for a task with k allocated cache ways:

$$MissRate = \begin{cases} \frac{1-\frac{k.(1-\frac{1}{\theta})}{A_1}-A_2}{1-A_2} & if k \leq A_1 \\ \frac{\frac{A^{\theta}k^{(1-\theta)}}{\theta}-A_2}{1-A_2} & if A_1 < k \leq k^0 \\ 0 & if k^0 < k \end{cases}$$

$$A_1 = A^{\theta/(\theta-1)}$$
$$A_2 = \frac{A^{\theta}}{\theta}(k^0)^{(1-\theta)} \quad (13)$$

In the above functions, θ is the locality parameter and is inversely proportional to the probability of making large jumps, the probability that a memory access visits a new cache line diminishes as the locality increases. According to [46], based on the studied traces, the range of $[1.5, 3]$ can be reasonable. (k^0) is the size of last level of cache memory, A_1 is critical cache sizes for, i.e.,

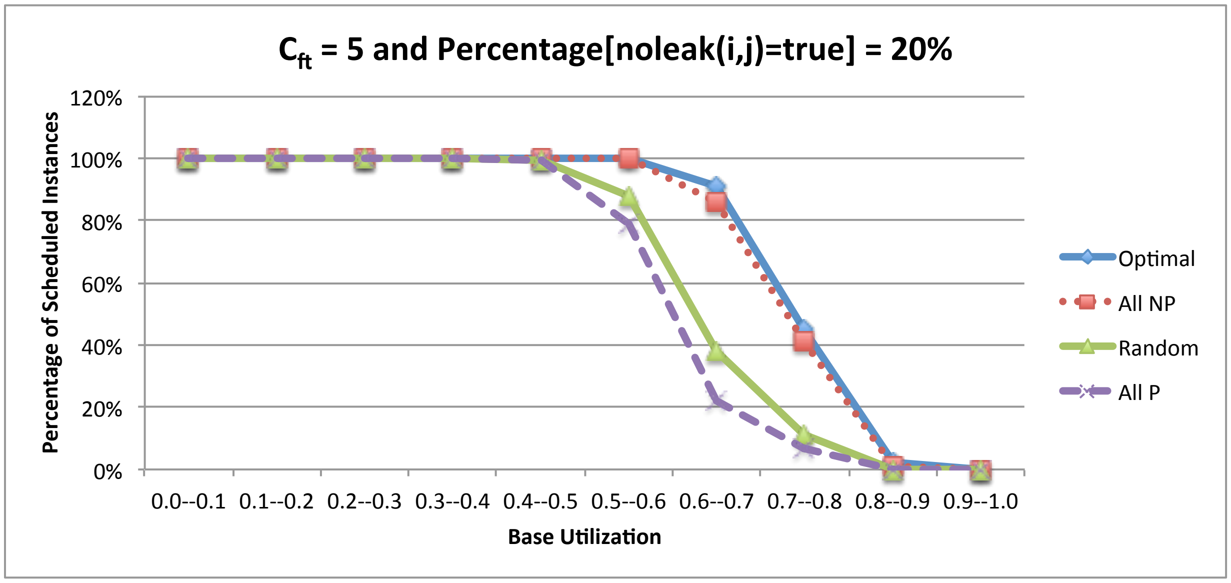


Figure 8.5: Preemptivity Assignments [Optimal vs. All Non-Preemptive vs. All Preemptive vs. Random]

it is working set size. Based on [46], if the cache size is larger than the working-set size, θ can describe the miss rate.

To set the remaining parameters to reasonable values, we conducted experiments using the PIN tool [30] on memory intensive benchmarks of some commonly used suits, such as: StreamIt, MediaBench, MiBench and SPEC2006. Based on these experiments, Table 8.4 summarizes the parameters used for our simulations. The next paragraphs provide more details on the employed methodology.

A(KB)	θ	$k^0(KB)$	Cache Memory References
[1,5]	[1.5,3]	[A1,512]	[2e+4,5e+4]

Table 8.4: Parameters of simulated tasks

Using PIN tool, we draw the figures of experimentation results of cache miss ratios for different cache sizes, examples can be found in Figures 8.6 and 8.7. In order to instrument through PIN toolset, we configured cache related configuration codes to meet our the characteristics of our hardware platform described in Chapter 7; therefore, cache line size is defined to be 32 B, cache sizes for both L1 Data and Instruction are set for 32 KB, cache way associativity for L1 and L2 are set as 4 and 8, cache L2 is set to be unified, and the number of sets for both of L1 and

L2 is computed as $CacheSize/(LineSize * Associativity)$. To instrument for different number of partitions and find the total *cache miss ratio* of read+write request from/to L2 cache, we only needed to change the total cache size and cache associativity to run the tools for a benchmark to capture the relevant traces.

After experimenting on some benchmarks running with different size of unlocked cache ways, we found that there are some typical behaviours that can be seen in all the experiments. We found that, the cache aware execution time is composed of two sections, as can be observed in Figures 8.6 and 8.7. They have significant drop points for their miss ratio, after which we have a flat line (less than 3-4% changes), meaning that there will not be a significant change in their miss ratio reductions (or execution time) even by providing more cache size to the task. We found those points as ways to find critical cache sizes (number of partitons) for each of the tasks, as discussed in Chapter 6. The corresponding trace data for some of the benchmarks can be found in Table 8.5.

Benchmark	Cache Refs	Critical Cache Size (KB)	MinMissRatio%
Anagram	49668	192	25.94
Epic	32669	192	36.44
h264decode-block	28461	256	36.44
Sha	35795	192	33.44
gsm-encode	29508	192	36.96

Table 8.5: Benchmarks Experimental Data

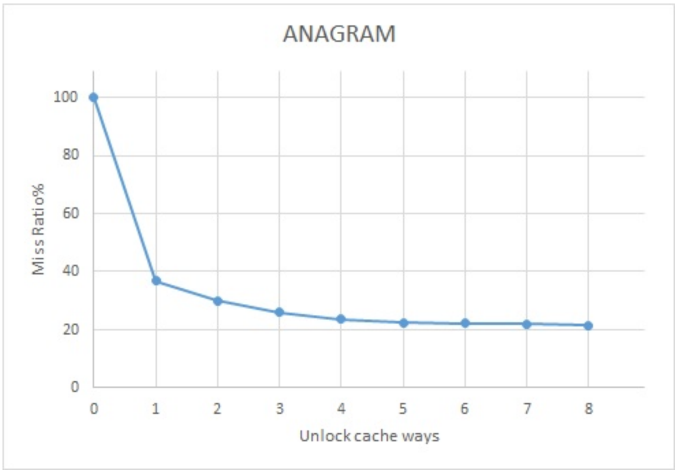


Figure 8.6: Anagram Benchmark task partitioning.

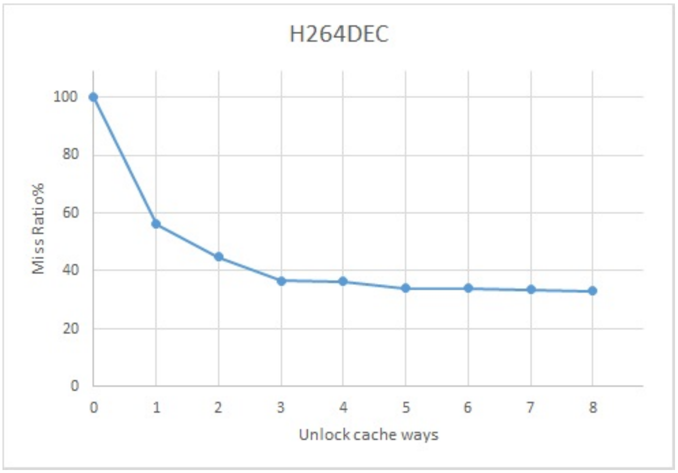


Figure 8.7: H264DEC Benchmark task partitioning.

The memory traces of some of the benchmarks are also listed in Table 8.6 which shows the total and unique number of address for read and write memory requests.

Benchmark	Total	Unique
Anagram	6067821	23771
Epic	17214190	23598
h264decode-block	1526038	23905
Sha	65534	1735

Table 8.6: Benchmarks Experimentations for Memory Traces

After obtaining data based on Table 8.5, we can also calculate execution times using the same equation as in [13]:

$$\begin{aligned} c_i(k) = (1 - MissRate(k)) * CacheRef * HitDelay + \\ MissRate(k) * CacheRef * MissDelay \end{aligned} \tag{14}$$

In order to generate $c_i(k)$ for each of the synthetic tasks, *CacheRef* is randomly generate from the range of *Cache Memory References* which is indicated in 8.4, and $MissRate(k)$ is computed by Equation 13. For calculating the values of *HitDelay* and *MissDelay*, since they are platform dependent, we used the datasheets of ARM A9 and PL310 cache controller to find the approximate cpu cycles of access and miss latencies for L2 cache and DDR3. The derived numbers for *HitDelay* and *MissDelay* are, in order, 19 ns and 73 ns.

8.2.2 Synthetic Results

Table 8.7 summarizes the parameters used for the synthetic task set generations. We evaluate a system with $K = 8$ partitions, similar to our implemented platform. Some of the parameters, such as: number of tasks and task period are the same as in Table 8.3. In addition, c_{ft} and c_{rt} are similarly the same, and the only difference is that, since here we wanted to consider a single partition overhead, we divided the relevant c_{ft} of Table 8.3 by the number of partitions. The cache-aware worst-case execution time $c_i(k)$ is derived as discussed in the previous section; the table reports the range of values observed after running 1 million traces of task generations.

Parameter	Value
Number of tasks, N	$[5, 20]$
Task period, p_i	$[5\,ms, 100\,ms]$
Task execution time, $c_i(k)$	$[0.24\,ms, 2.8\,ms]$
FT overhead, c_{ft}	$\{0.025\,ms, 0.062\,ms\}$
Reload overhead, c_{rt}	$\{0.025\,ms, 0.062\,ms\}$

Table 8.7: Experimental Parameters.

Regarding experiments, 1000 task sets have been generated distributed to different utilization intervals as $[i + 0.1, (i + 0.1) + 0.1)$ for $i = 0, ..., 9$, meaning that there are 100 incorporated synthetic task sets for each of the points in the following experimental figures. The deadline of each task is still equal to its period as before, i.e., $d_i = p_i$. The probability of noleak values to be true is varied between 10% and 80%. Results are illustrated in Figures 8.8, 8.10, 8.11, 8.12, 8.13, and 8.14 based on the percentage of schedulable task sets.

In order to better evaluate our GA optimization solution, we compare against three other approaches, namely: Greedy, Full Shared Partitioning, Randomized. The Greedy approach consists of running the local optimized described in Section 6.3 without the further GA optimization step. In the Full Shared Partitioning approach, each task uses all partitions available in the system, i.e., this algorithm is conceptually the same as the previous approach in Section 8.1. It is expected to show the worst performance, since we do not take advantage of resource partitioning to reduce the flushing overhead. Finally, in the Randomized case, we simply evaluate a fixed number of randomized partitioning assignments. This algorithm can be a good evaluation if our GA results are better solution than a trivial randomization since GA also works within the underlying principle of randomization. The number of randomized values is considered proportional to the maximum number of loops which are required to find a good solution in GA algorithm, that is equal to $(sizeofpopulation) * (numberofgenerations + 1)$; since we have $population = 5$ and $generations = 5$, the maximum of random generations for Randomized case will be equal to 30.

The randomized bit strings are pseudorandom, independent values with uniform distribution. In fact, the number of iterations for GA algorithm is variable, since a good solution is found as the stop criteria has been met; in the Randomized case, the algorithm might also be stopped earlier (without trying all possible number of randomized a_{NK}) as it found a scheduled case.

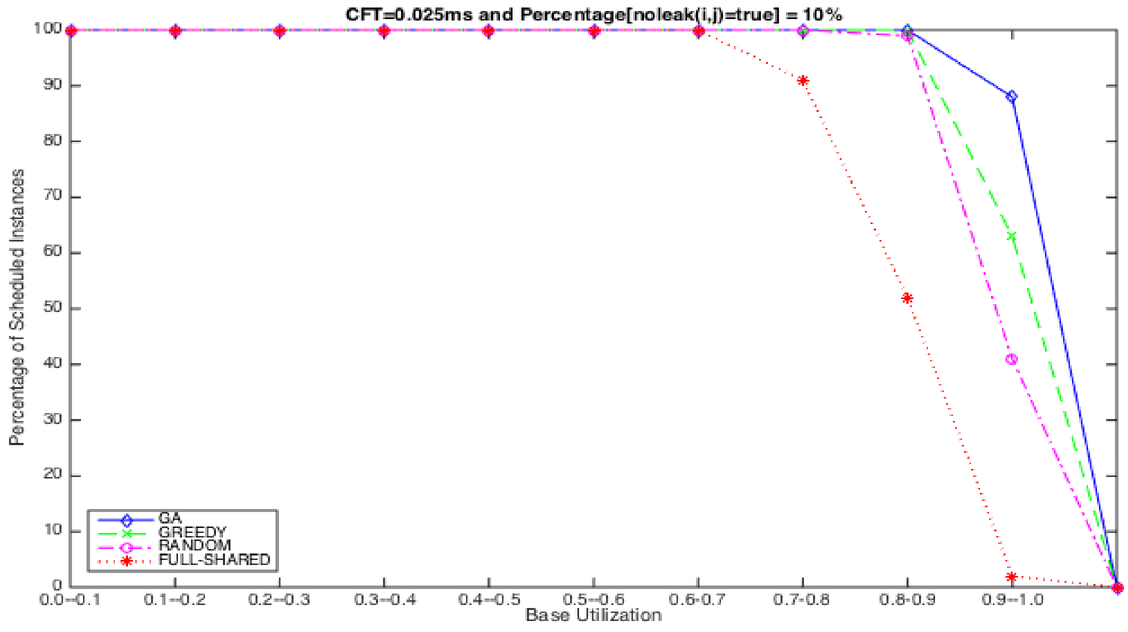


Figure 8.8: Synthetic Cache Partitioning Results. CFT=0.025 ms , NoLeak=10%

After observing the results of GA against Randomized, we have found that as the probability of $noleak$ true values increases, the utilization of $Randomized$ algorithm will get worse than GA. In fact, for the higher number of $noleak$ true values, more cache flushes are required and since tasks are randomly assigned to partitions, more partitions might happen to flush if they are already allocated to the tasks which need to be protected. Figures 8.12 and 8.14 can clearly show those distinctions.

For the larger values of FT and Reload overheads, we can observe different performance for GA and solely Greedy results. It explains that for those overheads we need to have more iterations to find better partitions for each of the tasks, and being decided once (greedily) on the task partition allocation based on the number of required partitions with regards to tasks′ mutual security protections, is not enough to have a good solution. Figures 8.11 and 8.14 illustrate this behavior.

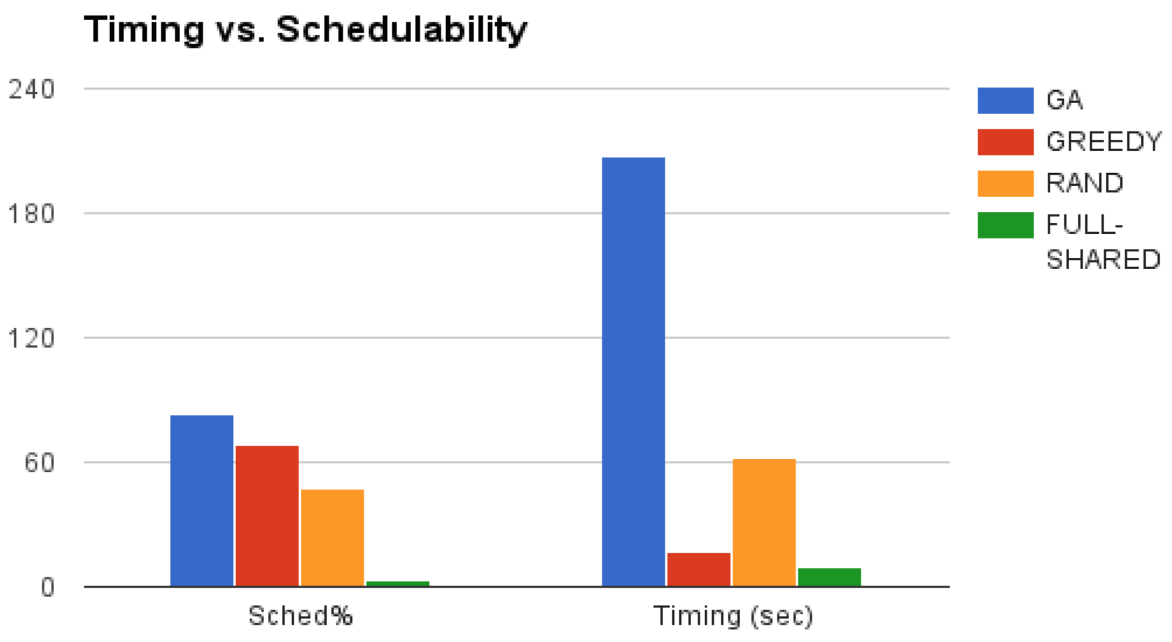


Figure 8.9: Timing vs. Schedulability. CFT=0.025 ms , NoLeak=50%

Concerning the run-time of the algorithms, there is a significant difference in two extremes, one is $FullSharedPartitioning$ algorithm which finds an answer in less than 10 seconds and another one is GA algorithm which might take 210 seconds to find a good answer. The comparative Figure 8.9 illustrates the timing to achieve one solution against the schedulability ratio for utilizations between 0.8-1.0 with $overhead = 0.025\,ms$ and $noleak = 50\%$, running on a Intel processor with 7 cores and 3.4 GHz frequency.

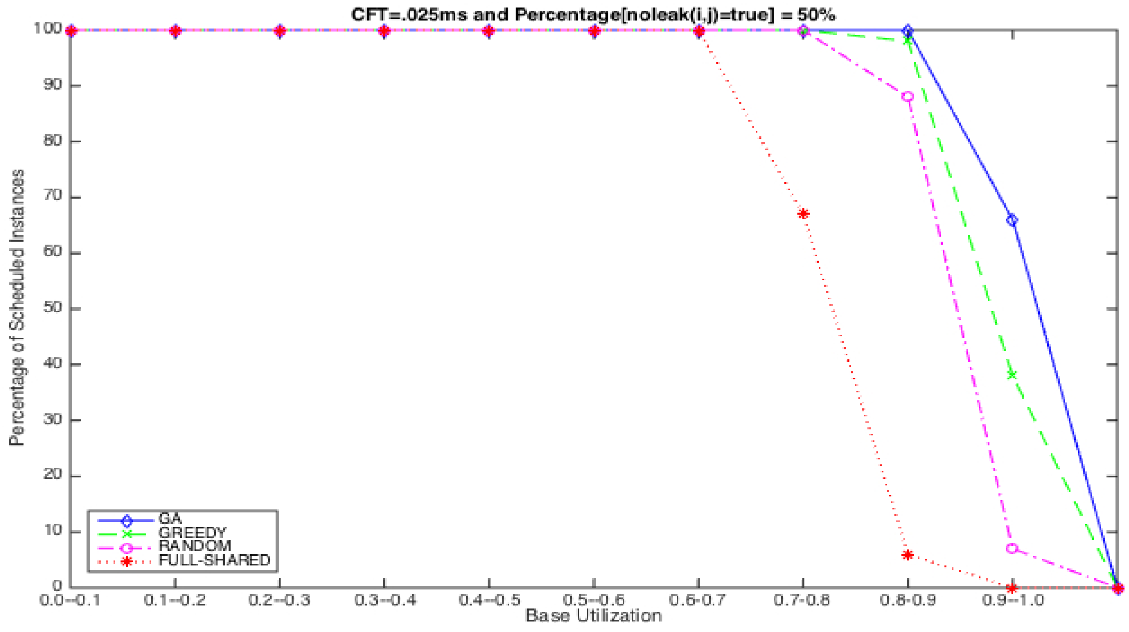


Figure 8.10: Synthetic Cache Partitioning Results. CFT=0.025 ms , NoLeak=50%

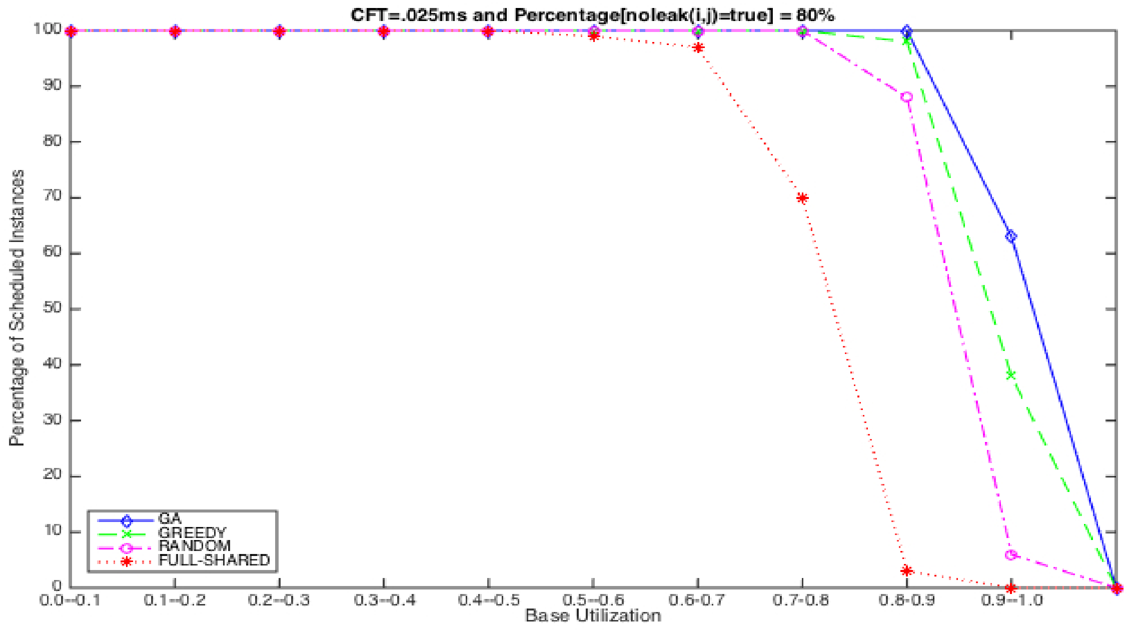


Figure 8.11: Synthetic Cache Partitioning Results. CFT=0.025 ms , NoLeak=80%

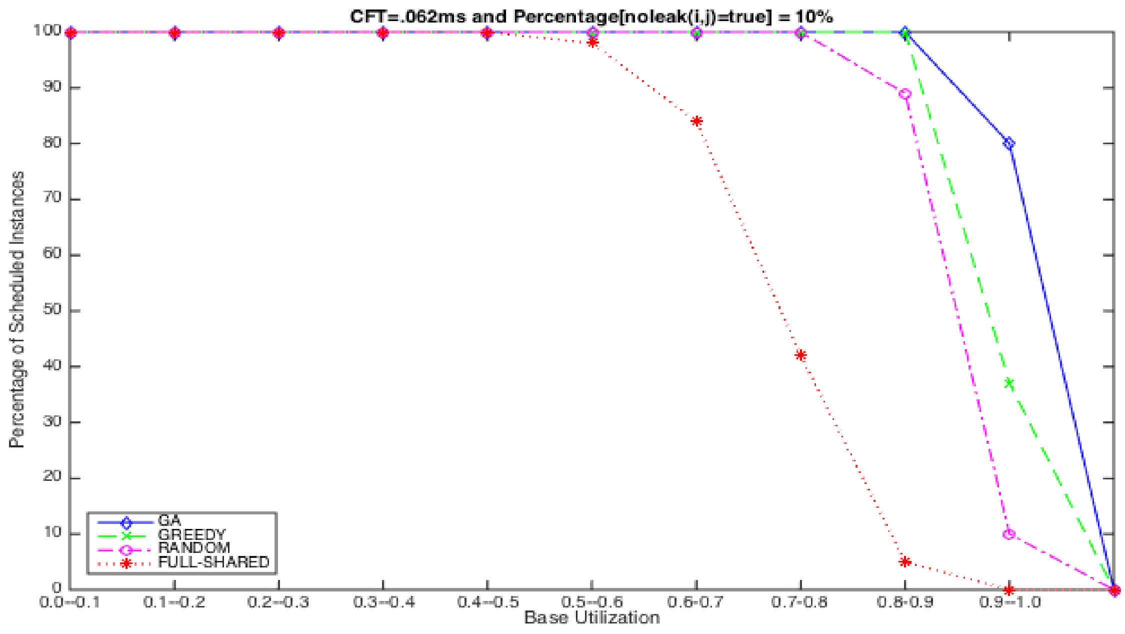


Figure 8.12: Synthetic Cache Partitioning Results. CFT=0.062 ms , NoLeak=10%

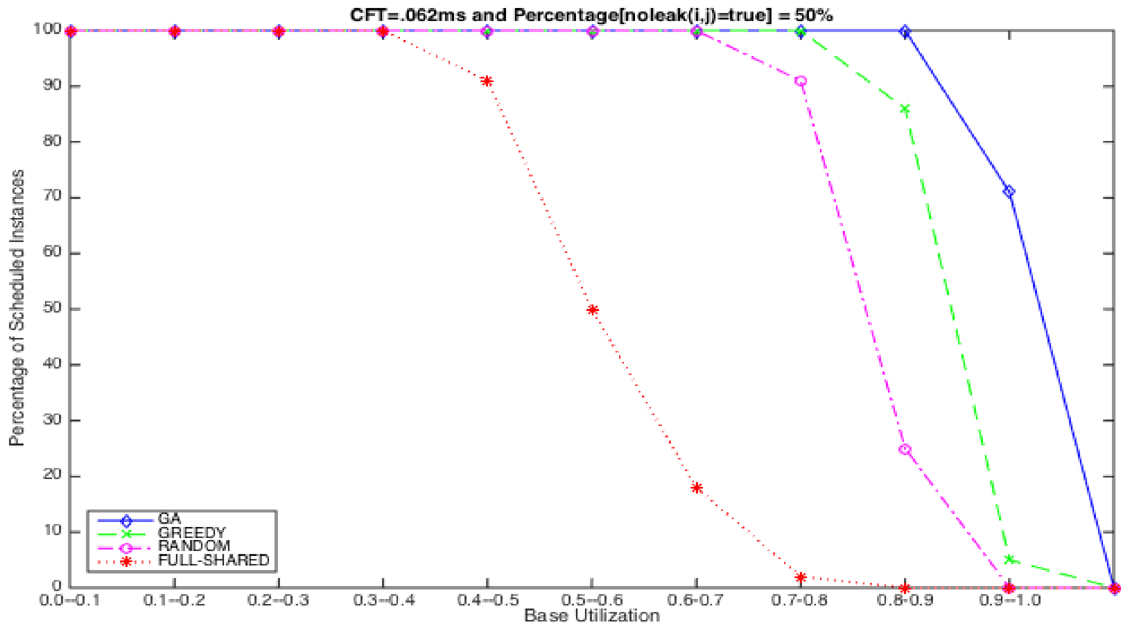


Figure 8.13: Synthetic Cache Partitioning Results. CFT=0.062 ms , NoLeak=50%

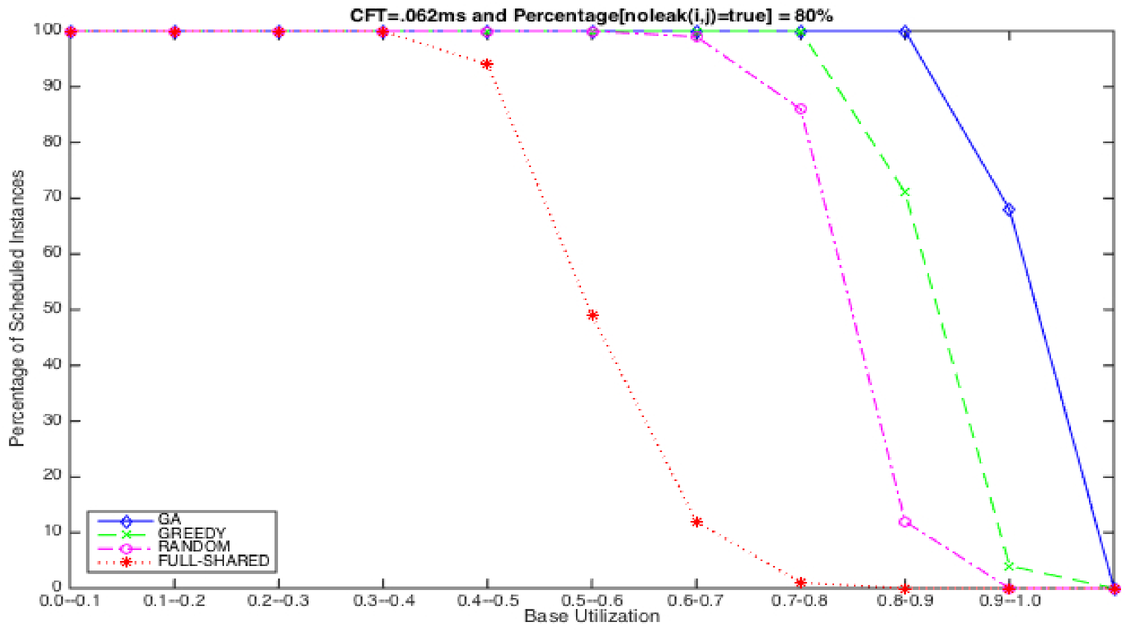


Figure 8.14: Synthetic Cache Partitioning Results. CFT=0.062 ms , NoLeak=80%

Chapter 9

Conclusions

This research explored some generalized solutions on the idea of integrating security-based constraints with hard RTSs using real-time task scheduling. The solutions have been demonstrated in theory, using extensive simulations, and also with implementation and running realistic tasks from different sub-systems. We believe that the whole work, including the previous works [33] and [36], is a first step towards helping researchers of real-time systems with evaluating their hard real-time systems against the cost of having security constraints.

As observed in Chapter 8, we evaluated two approaches of resource flushing and resource partitioning. The results show that resource partitioning can lead to lower overhead, hence improving performance in terms of task schdulability for hard RTSs. In addition, the devised optimization solution for the resource partitioning approach shows relevant improvements in comparison with trivial heuristics and the non-partitioned case. All simulation results are based on using the last level of cache memory as the partitioned resource in a single-core processor system, and use parameters derived from our implementation and application case study.

We believe that the solutions discussed in this work are not exclusively for cache, since information leakage possibility also exists through other shared physical resources, such as DRAMs or I/O bus. For example, I/O buses could be vulnerable to information leakage, in such a way that a previously run task might have affect the timing that a new task required to access the bus [34]. Also, since DRAM controllers handle an open row mechanism which resembles cache functionalities [49], our mechanism might also be applied to information leakage through DRAM controllers.

As mentioned, all the demonstrations of the security techniques have been applied to a single-core processor, since we wanted to concentrate on the correctness and effectiveness of the solutions on the simplest available platform. However, modern systems, even in the real-time domain, are progressively moving towards the adoption of multi-core processors or multi-processor architectures; thus, extending this work to such architectures would be a very relevant contribution.

A second worthwhile extension would be to explicitly demonstrate a security attacks, in the sense of information leakage. While cache-based attacks have been shown in the literature [26, 37, 53], including the ability to recover cryptographic keys [53], it would be relevant to quantify the amount and type of information that can be extracted by an attacker on our implemented platform with and without the proposed security mechanisms. Finally, there are some relevant open theoretical questions that have not been answered in this research, such as computational complexity bounds on the optimal solutions for the scheduling and partitioning problems.

References

[1] http://www.aosabook.org/en/freertos.html. Accessed: 2015-03-30.

[2] ARM Cortex A9 official technical reference. Accessed: 2015-03-30.

[3] ARM Security Technology: Building a Secure System using TrustZone Technology. Accessed: 2015-04-30.

[4] Experimenting For Everyone With a Hexacopter: Getting Practical Data for your Research. Accessed: 2015-03-30.

[5] FreeRTOS Official. http://freertos.org. Accessed: 2015-03-30.

[6] Real-time Systems Lab (RESL), university of Waterloo . Accessed: 2015-03-30.

[7] W32.Stuxnet Dossier. Accessed: 2015-05-10.

[8] Xilinx OS and Libraries Document Collection (UG643). Accessed: 2015-03-30.

[9] Zynq-7000 AP SoC Technical Reference Manual, UG585 (v1.8.1) September 19, 2014. Accessed: 2015-03-30.

[10] Quazi N Ahmed and Susan V Vrbsky. Maintaining security in firm real-time database systems. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 83–90.

[11] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[12] Enrico Bini and Giorgio C Buttazzo. Schedulability analysis of periodic fixed priority systems. *Computers, IEEE Transactions on*, 53(11):1462–1473, 2004.

[13] Bach Duy Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 101–110.

[14] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[15] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec 1992.

[18] Nicolas Falliere, Liam Murchu, and Eric Chien (Symantec). W32.stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2011.

[19] Rich Goyette. An analysis and description of the inner workings of the freertos kernel. *Carleton University*, 5, 2007.

[20] Northrup Grumman. RePLACE. http://www.northropgrumman.com/Capabilities/RePLACE/Pages/default.aspx.

[21] W-M Hu. Lattice scheduling and covert channels. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, pages 52–61. IEEE, 1992.

[22] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3):233–254, 1992.

[23] IBM ILOG. Cplex optimization studio. *URL: http://www-01. ibm. com/software/commerce/optimization/cplex-optimizer*, 2014.

[24] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204, 2012.

[25] David Kleidermacher and Mike Kleidermacher. *Embedded systems security: practical methods for safe and secure software and systems development*. Elsevier, 2012.

[26] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology–CRYPTO'96*, pages 104–113. Springer, 1996.

[27] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462.

[28] Man Lin, Li Xu, Laurence Tianruo Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. *Industrial Informatics, IEEE Transactions on*, 5(1):22–37, 2009.

[29] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[30] 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 *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[31] Mathworks Matlab. Simulink. *MathWorks Inc., Version*, 7(0), 2008.

[32] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3a: secure system simplex architecture for enhanced security of cyber-physical systems. *arXiv preprint arXiv:1202.5722*, 2012.

[33] Sibin Mohan, Man Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 129–140.

[34] Min-Young Nam, Rodolfo Pellizzoni, Lui Sha, and Richard M Bradford. Asiist: Application specific i/o integration support tool for real-time bus architecture designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 11–22.

[35] James B Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78:109–129, 1997.

[36] Rodolfo Pellizzoni, Neda Paryab, Man Ki Yoon, Sibin Mohan, Stanley Bak, and Rakesh Bobba. A generalized model for preventing information leakage in hard real-time systesm. In *Real-Time and Embedded Technology and Applications (RTAS), 2015 21st*. IEEE, 2015.

[37] Colin Percival. Cache missing for fun and profit, 2005.

[38] Derek Reinhardt. Certification criteria for emulation technology in the australian defence force military avionics context. In *Eleventh Australian Workshop on Safety Critical Systems and Software - Volume 69*, SCS '06, pages 79–92, Darlinghurst, Australia, 2006.

[39] Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Drone hack. *GPS World*, 23(8):30–33, 2012.

[40] Weidong Shi, Hsien-Hsin S Lee, Laura Falk, and Mrinmoy Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 102–113. IEEE Computer Society, 2006.

[41] Joon Son and J Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Information Assurance Workshop, 2006 IEEE*, pages 361–368.

[42] Sang H Son. Supporting timeliness and security in real-time database systems. In *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, pages 266–273. IEEE.

[43] Sang Hyuk Son, Craig Chaney, and Norris P Thomlinson. Partial security policies to support timeliness in secure real-time databases. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 136–147.

[44] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, volume 39, pages 85–96. ACM, 2004.

[45] Hugo Teso. Aircraft hacking. In *HITB Security Conference, Amsterdam, The Netherlands*, 2013.

[46] Dominique Thiebaut, Joel L. Wolf, and Harold S Stone. Synthetic traces for trace-driven simulation of cache memories. *Computers, IEEE Transactions on*, 41(4):388–410, 1992.

[47] Marcus Volp, Benjamin Engel, C Hamann, and Hermann Hartig. On confidentiality-preserving real-time locking protocols. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 153–162.

[48] Marcus Völp, Claude-Joachim Hamann, and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 44–55.

[49] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 372–383.

[50] Tao Xie and Xiao Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):20, 2007.

[51] Patrick Meumeu Yomsi and Yves Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 280–290. IEEE, 2007.

[52] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 21–32.

[53] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316.

[54] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 109–118. ACM, 2010.