POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Algorithms for scheduling hardware-software co-design systems

Supervisors

Prof. Marco GHIRARDI

Candidate

Ali Alrida FARHAT

October 2025

Algorithms for scheduling hardware-software co-design systems

Ali Alrida Farhat

Abstract

This thesis addresses the hardware–software co-design scheduling problem for real-time systems involving a single CPU and multiple reconfigurable Field Programmable Gate Arrays (FPGAs). The problem requires assigning tasks with precedence, release, and deadline constraints to either the CPU or an FPGA, while respecting reconfiguration overheads and minimizing a composite cost that balances scheduling delay and hardware usage.

Building on the seminal work of Ali et al. [1], who introduced a time-indexed mixed-integer programming (MIP) formulation, we first replicated and validated the original model to establish a performance baseline. While the MIP approach guarantees optimal solutions for small instances, its scalability is limited by the exponential growth of variables and constraints.

To address this, we developed two new algorithms. The first is a *mat-heuristic* that combines a time-limited MIP solve with iterative window-based refinements. This approach preserves near-optimal solution quality (typically within 5–10% of the MIP) while drastically reducing runtime, enabling the solution of medium to large task sets. The second is a *greedy + simulated annealing* method that constructs an initial feasible schedule in linear time and improves it through local search. Although this algorithm sacrifices optimality (with a gap of 10–20% compared to MIP), it achieves solutions within seconds even for the largest test cases considered.

Experimental comparisons demonstrate a clear trade-off between accuracy and speed: MIP ensures optimality but does not scale; the mat-heuristic achieves a practical balance; and the greedy + SA method offers unmatched speed for very large instances. Together, these three approaches provide a complementary toolkit for hardware–software co-design scheduling, allowing practitioners to choose depending on whether accuracy, efficiency, or responsiveness is prioritized.

ACKNOWLEDGMENTS

to...

.... my supervisor, Prof. Marco Ghirardi, for his continuous guidance, encouragement, and valuable feedback throughout the development of this thesis. His expertise and mentorship have been instrumental in shaping both the direction of the research and my own growth as a researcher.

I am also thankful to the faculty and staff of the Politecnico di Torino for providing an excellent academic environment and the resources necessary to carry out this work. Special thanks go to my colleagues and friends in the program, whose discussions and support have made the research process both productive and enjoyable.

I owe heartfelt thanks to my family for their unwavering support, patience, and encouragement during my studies. Their belief in me has been a constant source of motivation and strength.

Finally, I would like to dedicate this work to all those who have inspired me to pursue engineering and research, and who have instilled in me the values of perseverance, curiosity, and continuous learning.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Embedded systems increasingly rely on tight integration between hardware and software components to meet real-time performance requirements. Reconfigurable hardware platforms, particularly Field Programmable Gate Arrays (FPGAs), have gained popularity due to their flexibility and parallel processing capabilities, which complement general-purpose processors (CPUs) in heterogeneous systems.

In hard real-time systems—such as those used in aerospace, automotive, and industrial automation—every task must complete within a strict deadline. In such systems, efficient task partitioning (deciding which task runs on which resource) and scheduling (deciding when each task runs) is critical. Traditional two-phase approaches, where partitioning and scheduling are done sequentially, often lead to suboptimal solutions because these decisions are interdependent.

1.2 Problem Overview

Ali et al. [1] proposed a time-indexed mixed-integer programming (MIP) formulation to simultaneously handle task partitioning and scheduling in hardware-software co-design. Their model considers a system with a single CPU and multiple FPGAs, where tasks have fixed release times, due dates, and precedence constraints. The goal is to minimize a composite cost function that accounts for scheduling delay and the number of FPGAs used.

While their approach is rigorous and yields optimal solutions for small to moderate instances, its scalability is limited. As the number of tasks increases, the model becomes computationally expensive due to the explosion of binary variables and constraints.

1.3 Objectives and Contributions

This thesis aims to:

- Reconstruct the original MIP model presented by Ali et al., validating its performance using a new implementation and test cases.
- Evaluate the model's scalability and computational performance across different task set sizes.
- Propose enhancements to the original model that improve its efficiency and applicability to larger systems with more tasks.
- Compare the extended model's performance with the original using metrics such as computation time, solution quality, and resource usage.

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

- Chapter 2 reviews the existing literature on hardware-software co-design, MIP-based scheduling, and FPGA-based real-time systems.
- Chapter 3 defines the problem formally, detailing the system configuration, task attributes, and scheduling constraints.
- Chapter 4 presents the original MIP model by Ali et al., including a discussion of its variables, constraints, and assumptions.
- Chapter 5 describes the implementation of the model and the test instance generation methodology.
- Chapter 6 provides experimental results comparing the replicated model with the original, followed by evaluation using new instances.
- Chapter 7 concludes the thesis and outlines potential directions for future research.

Chapter 2

Literature Review

Now we present a survey of prior work in the field of hardware-software co-design, focusing particularly on mixed-integer programming (MIP) approaches for scheduling real-time tasks.

2.1 Hardware-Software Co-design Approaches

Hardware-software co-design involves the concurrent design of hardware and software components in embedded systems to meet performance, cost, and time-to-market constraints. Traditional approaches separates the design process into two phases: hardware-software partitioning which is choosing which job takes what resource followed by scheduling them. However, this decoupled process can lead to suboptimal solutions due to the interdependence between these two stages.

Several works have proposed heuristic or metaheuristic approaches for co-design. Liu and Wong [2] presented an integrated partitioning and scheduling framework targeting execution time and hardware cost. Arato et al. [3] developed a scheduling technique where tasks that miss deadlines are offloaded to auxiliary hardware components.

However, these papers before have not been able to find optimal solutions for such a problem. As for the paper done by Ali et al. [1], the approach he proposed was giving optimal solutions but since he used an exact method (integer programming) to model the joint problem, he faced the problem of computational intensity. And such, he could not get the optimal solution for problems that have many tasks in a small time, which is crucial for real-time embedded systems.

2.2 MIP-Based Scheduling and Integrated Co-design

Mixed-Integer Programming (MIP) offers an exact optimization framework to model complex scheduling and resource allocation problems in hardware-software co-design. Unlike heuristic approaches, MIP models provide optimal solutions given enough computational resources and time, and they allow for the formal inclusion of precedence constraints, resource usage, and timing requirements.

One notable MIP-based approach was proposed by Niemann and Marwedel [4], who developed a two-phase model for partitioning and scheduling. The first phase proposes a tentative schedule, and the second verifies timing constraints. This sequential design, however, lacks the ability to make optimal joint decisions.

Ali et al. [1] addressed this limitation by introducing a time-indexed MIP model that simultaneously performs task partitioning and scheduling. Their model considers a system with a single CPU and up to m reconfigurable FPGAs (m being a parameter that can be changed), incorporating processing times, reconfiguration delays, and strict due-date constraints. The objective function minimizes a composite cost of task assignment and FPGA usage. The model has been shown to produce optimal solutions for small to medium-sized problem instances, but it becomes computationally expensive as the number of tasks increases.

Their work has laid the foundation for research like this thesis, which aims to improve the scalability of such MIP models to handle larger and more complex systems efficiently.

2.3 Use of FPGAs in Real-Time Systems

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices that offer significant performance advantages in embedded and real-time systems. Unlike general-purpose processors, FPGAs can execute multiple operations in parallel and can be tailored to specific workloads, making them suitable for compute-intensive and latency-sensitive applications.

In hardware-software co-design, FPGAs are commonly used as accelerators to offload tasks from the CPU. However, their reconfiguration ability introduces unique challenges: tasks assigned to FPGAs require a reconfiguration process before execution, and this process often involves a shared controller that can only handle one reconfiguration at a time. These constraints complicate scheduling and must be explicitly modeled to ensure correct and efficient task execution.

The flexibility and performance benefits of FPGAs make them an attractive resource in real-time systems. However, their dynamic behavior, limited reconfiguration bandwidth, and non-preemptive nature necessitate careful modeling.

Chapter 3

Problem Definition

This chapter formally defines the hardware-software co-design problem addressed in this thesis. We consider a real-time system consisting of a single Central Processing Unit (CPU) and a set of reconfigurable Field Programmable Gate Arrays (FPGAs). The system must schedule a set of non-pre-emptive, aperiodic tasks that are subject to strict precedence constraints, release times, and due-dates. Each task can be assigned to either the CPU or one of the available FPGAs. The objective is to determine a feasible and optimal assignment and schedule that minimizes a composite cost function while satisfying all timing and structural constraints.

3.1 System Description

The target system consists of the following components:

- A single Central Processing Unit (CPU) that serves as the default processing unit.
- A maximum of m Field Programmable Gate Arrays (FPGAs), each of which can be dynamically reconfigured to execute specific tasks.
- A shared reconfiguration controller responsible for configuring the FPGAs. Only one FPGA can be reconfigured at a time.
- Two communication buses: one for I/O data transfers and another for reconfiguration data transfer.

The CPU and FPGAs are connected through a shared architecture that supports task execution and dynamic hardware reconfiguration. While the CPU is always available for task processing, the FPGAs are used selectively due to their setup overhead and reconfiguration delays. Additionally, each FPGA can execute only one task at a time and must be reconfigured before processing each new task.

The system timeline is discretized into uniform time-slots to facilitate a time-indexed scheduling approach. Each processing resource (CPU or FPGA) is associated with a series of time-slots over which tasks can be scheduled.

As shown in Figure 3.1, the reconfiguration controller is shared across all FPGAs.

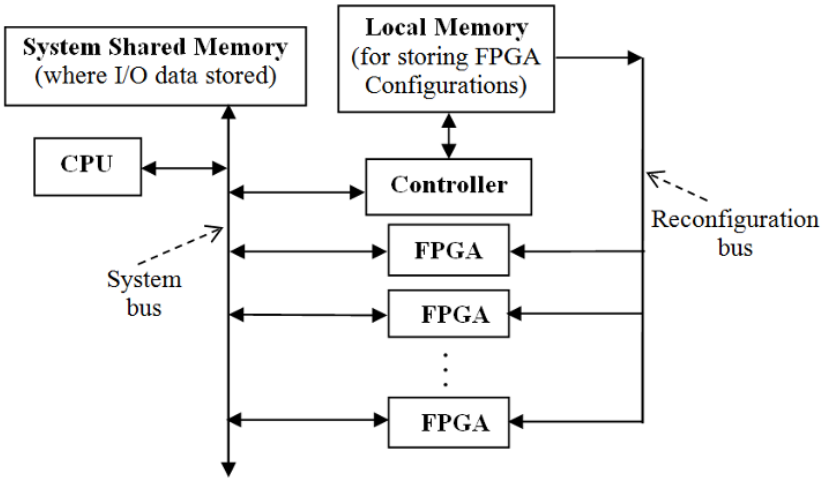

Figure 3.1: System architecture with CPU, FPGAs, and shared controller.

3.2 Task Characteristics

We consider a set of n indivisible, non-preemptive tasks $\{1, 2, \ldots, n\}$ to be scheduled on the system described above. Each task is characterized by the following properties:

- **Release time (α_j):** Each task j has a specified release time α_j, which represents the earliest time-slot at which the task becomes available for processing. A task cannot begin execution before its release time.
- **Due date (d_j):** Each task j has a strict due date d_j that must not be violated. This reflects the hard real-time nature of the system; any schedule that causes a task to miss its due date is considered infeasible.
- **Precedence constraints:** Tasks may have interdependencies captured by a set of precedence relations. If task j_1 must complete before task j_2 can start, this is represented as an arc (j_1, j_2) in the precedence graph. The scheduler must ensure that all such constraints are respected.
- **Execution time:** The time required to execute a task depends on the assigned resource:
  - When executed on the **CPU**, a task has a known processing time δ_{j0}.
  - When executed on **FPGA** r $(r = 1, \ldots, m)$, the task requires a reconfiguration time π_{jr} in addition to its processing time. The total time to execute task j on FPGA r is denoted by δ_{jr}, where $\delta_{jr} \geq \pi_{jr}$.
- **Non-preemptiveness:** Once started, a task must execute to completion without interruption, regardless of the assigned resource.

Together, these properties introduce complexity into the scheduling problem. The scheduler must determine not only the assignment of each task to a resource (partitioning), but also the exact start time of each task such that all constraints are satisfied and the overall cost is minimized.

3.3 Objective of the Scheduling Problem

The goal of the scheduling problem is to determine a feasible assignment and ordering of tasks on available processing resources such that all constraints are satisfied, and the total system cost is minimized.

The cost function used in this thesis consists of two components:

- **Scheduling efficiency cost:** Each task incurs a cost based on the time-slot at which it begins execution. This reflects a preference for earlier task execution and helps avoid idleness and schedule slack. The cost is typically modelled to increase with later start times.
- **Hardware usage cost:** Each FPGA incurs a fixed cost if it is used for any task. This component discourages unnecessary FPGA activation and reflects the practical costs associated with dynamic reconfiguration, power consumption, and hardware utilization.

Together, these components form a composite cost function that is minimized by the scheduler. The problem thus involves making the following decisions:

1. Whether each task should be assigned to the CPU or to one of the FPGAs.
2. When each task should begin execution, given its release time, due date, and precedence constraints.
3. Which FPGAs are activated and how they are shared under the single-controller constraint.

This problem is modelled as a time-indexed Mixed-Integer Programming (MIP) formulation, described in detail in Chapter 4.

Chapter 4

Original MIP Model by Ali et al.

In this chapter, we describe the mathematical model introduced by Ali et al. [1] to solve the problem of scheduling and assigning real-time tasks in a system with one Central Processing Unit (CPU) and several Field Programmable Gate Arrays (FPGAs).

The model uses a technique known as *Mixed-Integer Programming* (MIP), a powerful optimization framework that allows decisions to be made over both binary (true or false) and continuous variables, subject to constraints. The model is built to handle practical challenges such as:

- Tasks that cannot be interrupted once they start (non-pre-emptive).
- Dependencies between tasks (precedence constraints).
- Tasks that must start after their release time and finish before their due date.
- FPGAs that need to be reconfigured before use, and share a single reconfiguration controller.

The purpose of the model is to determine:

1. Which task should be executed on which resource (CPU or FPGA).
2. When each task should start, ensuring no overlaps or deadline violations.
3. Whether each FPGA needs to be activated at all, as activating it contributes to the total cost.

The goal is to minimize the total cost, which includes a penalty for starting tasks later and a fixed cost for using each FPGA. In the sections below, we explain the components of the model step by step, starting with the notation and decision variables.

4.1 Sets, Indices, and Parameters

To build the mathematical model, we begin by defining the key components: the sets of tasks and resources, and the parameters that describe timing and costs.

Indices

- $j = 1, 2, \ldots, n$: Index for tasks (also called jobs).
- $r = 0, 1, \ldots, m$: Index for processing resources, where $r = 0$ denotes the CPU and $r = 1$ to m denote the available FPGAs.
- $t = 1, 2, \ldots, s$: Index for discrete time-slots in the scheduling timeline.

Time-Slot Structure

Time is divided into small equal intervals called *time-slots*. Each resource (CPU or FPGA) has its own timeline, made up of these time-slots. The total number of time-slots, s, is chosen large enough to cover the entire scheduling period, typically based on the latest task due date.

Task Parameters

Each task j has the following properties:

- α_j: **Release time** — the earliest time a task is allowed to start.
- d_j: **Due date** — the latest time by which a task must be completed.
- δ_{jr}: **Processing time** of task j on resource r.
  - δ_{j0}: Time to run task j on the CPU.
  - δ_{jr} for $r \geq 1$: Time to run task j on FPGA r, including reconfiguration time.
- π_{jr}: **Reconfiguration time** — the time required to prepare FPGA r for task j (only applicable when $r \geq 1$).
- P: **Precedence set**, where $(j_1, j_2) \in P$ means that task j_1 must finish before task j_2 can start.

Cost Parameters

- c_{jk}: The cost of starting task j at time-slot k. Usually increases with time, to favor earlier execution.
- λ: A fixed cost associated with using an FPGA. If an FPGA is used for even one task, this cost is incurred.

Remarks

The processing time δ_{jr} already includes the reconfiguration time π_{jr} for FPGA execution. That is:

$$\delta_{jr} = \text{actual execution time} + \pi_{jr}, \quad \text{for } r \geq 1$$

Also, all time-related values (release times, durations, deadlines) are expressed in units matching the time-slot length, so they are assumed to be integers.

4.2 Decision Variables

The model introduces a set of variables that represent the decisions to be made in order to produce an optimal schedule.

Primary Decision Variables

- $x_{jk} \in \{0, 1\}$:
  - $x_{jk} = 1$ if task j is assigned to start at time-slot k on some resource.
  - $x_{jk} = 0$ otherwise.

  This is the main scheduling variable that tells us when and where each task starts.

- $y_r \in [0, 1]$:
  - $y_r = 1$ if FPGA r is used to execute at least one task.
  - $y_r = 0$ otherwise.

  Even though y_r is defined as continuous in the model, it will end up being either 0 or 1 in the optimal solution. These variables are used to calculate FPGA usage cost.

Auxiliary Variables

To simplify the formulation, two auxiliary variables are derived from x_{jk}:

- s_j: The time-slot at which task j starts. This is computed based on the x_{jk} values.
- f_j: The time-slot at which task j finishes. This depends on when it starts and how long it takes.

These variables are not independent decisions, but they are calculated from the x_{jk} variables to help express the constraints more clearly.

4.3 Objective Function

The goal of the MIP model is to minimize the total cost of scheduling and resource usage. The objective function has two parts:

- **Task scheduling cost:** Starting a task later typically incurs a higher cost. The model assigns a cost c_{jk} for starting task j at time-slot k. These costs can be chosen to encourage earlier scheduling of tasks.

- **FPGA usage cost:** Each FPGA r incurs a fixed cost λ if it is used for any task. This helps reduce the number of FPGAs used unnecessarily.

The total cost is the sum of all individual task scheduling costs plus the total cost of FPGA usage:

$$\text{Minimize} \sum_{j=1}^{n} \sum_{k \in S_j} c_{jk} x_{jk} + \lambda \sum_{r=1}^{m} y_r$$

Where:

- $x_{jk} = 1$ means task j starts at slot k
- S_j is the set of all valid slots for task j (we'll define this set soon)
- $y_r = 1$ means FPGA r is used
- λ is the fixed cost for using one FPGA

This function balances two competing goals:

1. Schedule tasks as early as possible to avoid high scheduling cost.
2. Use as few FPGAs as necessary to reduce hardware cost.

4.4 Definition of Helper Sets

To express the constraints of the model efficiently, several sets are defined. These sets represent valid start times, task execution windows, and reconfiguration periods.

1. Set S_j — Valid Start Slots for Task j

This is the set of time-slots in which task j can legally begin execution, considering its release time α_j, due date d_j, and its processing time on all possible resources.

$$S_j = \left\{ k \in \{1, \ldots, (m+1)s\} \ : \ lb_j \leq k \mod {}^{+}s \leq d_j - \delta_{jr(k)} + 1 \right\}$$

Where: - lb_j is the lower bound on the start time of task j (at least its release time, or later if it has predecessors). - $r(k)$ tells us which resource is associated with slot k.

—

2. Set S_{jr} — Start Slots on Resource r

This set includes all valid start slots for task j that belong specifically to resource r (e.g., CPU or a particular FPGA):

$$S_{jr} = \{k \in S_j \mid r(k) = r\}$$

—

3. Set J_k — Tasks That Occupy Slot k

This set includes all combinations of tasks j and start slots ℓ such that task j would occupy time-slot k if started at slot ℓ:

$$J_k = \Big\{(j,\ell) \ : \ \ell \in S_j,\ r(\ell) = r(k),\ \ell \mod^+ s \le k \mod^+ s \le \ell \mod^+ s + \delta_{jr(\ell)} - 1\Big\}$$

—

4. Set R_t — Tasks Reconfiguring at Time t

This set contains all task-start combinations that would cause a reconfiguration to be in progress during time-slot t:

$$R_t = \Big\{(j,\ell) \ : \ \ell \in S_j,\ \ell \ge s+1,\ \ell \mod^+ s \le t \le \ell \mod^+ s + \pi_{jr(\ell)} - 1\Big\}$$

This is relevant only for tasks assigned to FPGAs, since the CPU does not require reconfiguration.

—

5. Modulo Notation $\mod^+ s$

In this model, modulo arithmetic is used to map global slot indices back to local time-slots. The notation $k \mod^+ s$ means:

$$k \mod^+ s = \begin{cases} k \mod s & \text{if } k \mod s \neq 0 \\ s & \text{if } k \mod s = 0 \end{cases}$$

This ensures consistent slot numbering across the CPU and FPGA timelines.

4.5 Model Constraints

The model includes several constraints that ensure the schedule is feasible and respects all timing, dependency, and resource limitations.

1. Each Task Must Be Scheduled Exactly Once

Every task must start at exactly one valid time-slot, on one resource.

$$\sum_{k \in S_j} x_{jk} = 1 \quad \forall j = 1, \ldots, n$$

Where S_j is the set of all time-slots where it is valid to schedule task j (we'll define this later). This ensures no task is skipped or duplicated.

—

2. No Overlapping Tasks on the Same Resource

Each time-slot on a resource can only be used by one task. This avoids overlapping execution.

$$\sum_{(j,\ell)\in J_k} x_{j\ell} \leq 1 \quad \forall k = 1, \ldots, (m+1)s$$

Where J_k is the set of all tasks and start times that would occupy time-slot k.

—

3. Only One FPGA Can Be Reconfigured at a Time

The system has a single reconfiguration controller. So at most one FPGA can be reconfiguring in any time-slot.

$$\sum_{(j,\ell)\in R_t} x_{j\ell} \leq 1 \quad \forall t = 1, \ldots, s - \Delta$$

Where R_t is the set of tasks and start times that would result in a reconfiguration during time-slot t. Δ is a small buffer defined to simplify the timing.

—

4. Compute Start and Finish Time of Each Task

The start and finish times of each task are derived from the selected x_{jk} values.

$$s_j = \sum_{k\in S_j} (k \mod^+ s) \cdot x_{jk}$$

$$f_j = \sum_{k\in S_j} \Big[(k \mod^+ s) + \delta_{jr(k)} - 1\Big] \cdot x_{jk}$$

Here, $\mod^+$ means the remainder of k divided by s, but equals s if the remainder is 0 (to align time-slots across resources).

—

5. Enforce Precedence Constraints

If task j_1 must finish before task j_2 starts, we must ensure:

$$f_{j_1} + 1 \leq s_{j_2} \quad \forall (j_1, j_2) \in P$$

—

6. Activate an FPGA Only If It Is Used

If any task is assigned to FPGA r, then y_r must be set to 1.

$$y_r \geq \sum_{k\in S_{jr}} x_{jk} \quad \forall j, \quad r = 1, \ldots, m$$

—

7. Enforce FPGA Symmetry (Optional)

To reduce computational complexity, symmetry-breaking constraints are used to prefer lower-indexed FPGAs:

$$1 \geq y_1 \geq y_2 \geq \cdots \geq y_m \geq 0$$

$$\sum_j \sum_{k \in S_{jr}} x_{jk} \geq \sum_j \sum_{k \in S_{j,r+1}} x_{jk} \quad \forall r = 1, \ldots, m-1$$

—

8. Binary and Continuous Variable Constraints

$$x_{jk} \in \{0, 1\}, \quad y_r \in [0, 1]$$

Although y_r is continuous, it will become binary at optimality due to the model structure.

—

These constraints ensure that the model makes valid, conflict-free decisions while trying to minimize the overall cost.

Chapter 5

Mat-heuristic Implementation

In this chapter, we present a mat-heuristic algorithm developed to solve the hardware-software co-design problem introduced in Chapter 4. Unlike the original MIP model by Ali et al. [1], which attempts to optimize all tasks simultaneously, our approach combines mathematical programming with heuristic strategies to improve scalability for larger task sets.

A *mat-heuristic* is a hybrid method that integrates heuristics (fast, approximate algorithms) with exact optimization techniques. In our case, the method uses a decomposition approach: it selects a subset of tasks to optimize using the MIP formulation while fixing the remainder based on heuristic rules. This allows the algorithm to handle larger instances efficiently without sacrificing too much solution quality.

5.1 Algorithm Overview

The proposed mat-heuristic integrates a Mixed-Integer Programming (MIP) formulation with a block-based refinement heuristic to handle large instances of the hardware-software co-design problem. The algorithm proceeds in two phases:

1. **Initial Feasible Solution:** The full MIP model is solved under a strict time limit using the Gurobi optimizer. This step aims to quickly generate a feasible (not necessarily optimal) schedule that satisfies all constraints.

2. **Window-Based Refinement:** The task set is divided into overlapping blocks or *windows*. For each window, only the tasks inside it are allowed to change assignment and start time, while all other tasks are fixed to their current values. The MIP is re-solved for each window sequentially, improving the overall solution iteratively.

This decomposition strategy reduces the search space in each refinement step, allowing the algorithm to improve the solution quality without facing the full complexity of the global problem. By adjusting the window size, a trade-off is achieved between computation time and solution optimality.

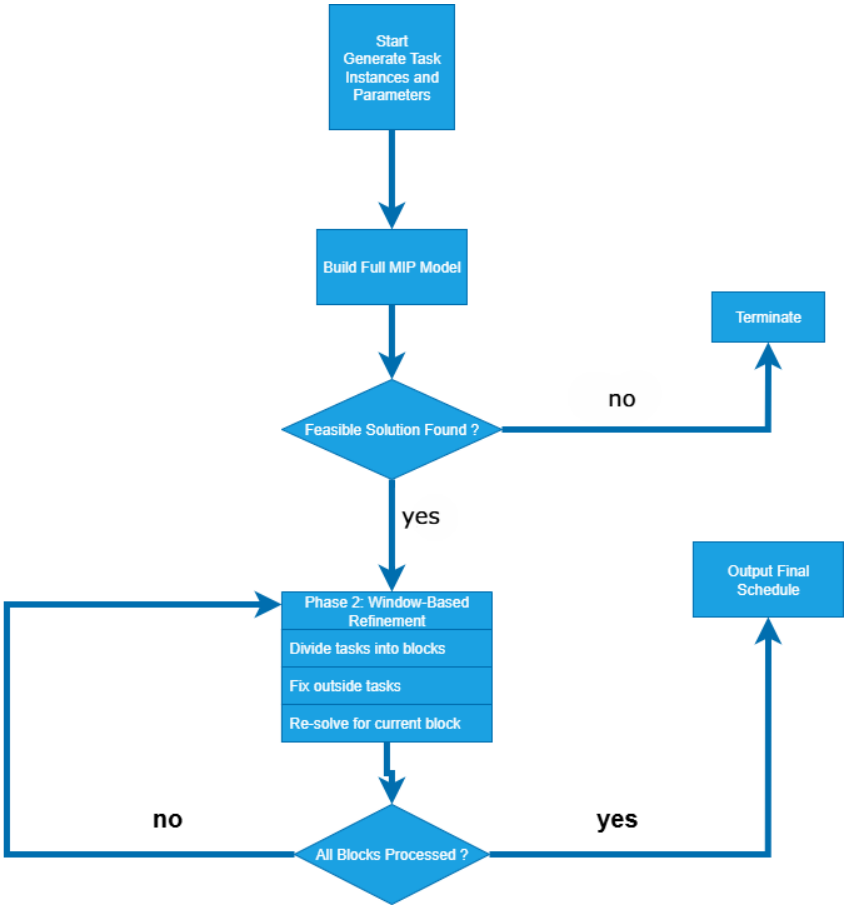

Figure 5.1: Flowchart of the Mat-heuristic algorithm.

5.2 Algorithm Flowchart

Figure 5.1 illustrates the overall workflow of the proposed mat-heuristic algorithm, which is divided into two main phases.

The process begins by generating the task instance and all associated parameters, including release times, due dates, processing times, and precedence constraints. Based on this data, the full Mixed-Integer Programming (MIP) model is constructed, incorporating the resource constraints and scheduling objectives described in Chapter 4.

In **Phase 1**, the MIP model is solved using the Gurobi optimizer under a strict time limit. The aim of this step is not to find the global optimum, but to produce a feasible schedule that satisfies all deadlines and resource constraints. If no feasible solution is found within the time limit, the algorithm terminates early.

If a feasible schedule is obtained, **Phase 2** begins. In this stage, the set of tasks is divided into smaller overlapping subsets, called *windows*. For each window, the tasks outside the window are fixed to their current schedule, while the tasks inside the window are free to be reassigned and rescheduled. The MIP solver is then called again to optimize the schedule for the current window. This refinement process iterates over all windows until every subset of tasks has been processed.

At the end of Phase 2, the refined final schedule is produced, along with FPGA usage data and the total objective value. This two-step approach allows the algorithm to handle much larger task sets than a single full MIP solve, striking a balance between solution quality and computational efficiency.

5.3 Pseudo-code of the Mat-heuristic

To complement the flowchart shown in Figure 5.1, Algorithm 1 provides a structured representation of the proposed mat-heuristic in pseudo-code form. The pseudo-code highlights the two-phase design of the algorithm: a fast initial MIP solve to obtain a feasible schedule, followed by iterative window-based refinement to improve the solution.

Presenting the algorithm in this format makes it independent of any specific programming language, focusing instead on the logic and workflow. This allows the method to be implemented in various environments, such as Python with Pyomo or MATLAB with Gurobi.

Algorithm 1 Mat-heuristic for Hardware-Software Co-design Scheduling

Require: Task set J, number of FPGAs m, time horizon S
Ensure: Feasible schedule and FPGA usage
Generate task instance: release times, due dates, processing times
Build full MIP model with all tasks and resources

Phase 1: Initial Feasible Solution
Solve MIP model with time limit using Gurobi
if no feasible solution found **then**
    return "No feasible schedule"
end if
Save the solution as current schedule

Phase 2: Window-Based Refinement
Partition tasks into overlapping windows of size w
for each window W in task order **do**
    Fix assignments of tasks outside W to current schedule
    Unfix tasks inside W
    Re-solve the MIP model for window W
    Update current schedule with improved solution
end for

Output: Refined schedule, FPGA usage, objective value

5.4 Step-by-Step Description

To clarify the workflow of the mat-heuristic, we describe its two phases with a simple example:

1. **Instance generation:** Suppose $n = 10$ tasks and $m = 2$ FPGAs. Release times, due dates, and precedence constraints are fixed.
2. **Phase 1 – Initial solution:** A time-limited MIP is solved. Assume it produces a feasible schedule where most tasks meet deadlines but some idle gaps remain.
3. **Phase 2 – Window refinement:** The task set is partitioned into overlapping windows, e.g. tasks 1–5, then 4–8, then 7–10. In each window:
   - Tasks outside the window remain fixed.
   - Tasks inside the window are re-optimized by MIP.
4. **Iteration:** For the first window (1–5), the solver shifts tasks 2 and 3 earlier, closing idle time. For the second window (4–8), reassignments reduce reconfiguration congestion. After the third window, the solution is globally improved.
5. **Output:** The final schedule balances deadlines, reduces idle time, and uses fewer FPGA activations than the initial baseline.

5.5 Summary and Transition

In this chapter, we introduced a mat-heuristic algorithm designed to address the scalability challenges of the original MIP model for hardware-software co-design. By combining an initial time-limited MIP solve with iterative window-based refinement, the method achieves a balance between computational efficiency and solution quality. This approach allows the scheduling of significantly larger task sets compared to solving the full MIP model directly.

While the mat-heuristic leverages exact optimization as its core engine, the next chapter explores a complementary strategy based on heuristic search techniques. Specifically, we present an algorithm that integrates a greedy task assignment procedure with a local search improvement phase enhanced by simulated annealing. This alternative method trades off guaranteed optimality for much faster runtime and is particularly suited for very large instances where even the mat-heuristic may struggle to find solutions within practical time limits.

Chapter 6

Greedy-Based Scheduling with Local Search and Simulated Annealing

This chapter presents a heuristic-based approach to the hardware-software co-design problem that emphasizes speed and scalability. Unlike the MIP-based mat-heuristic introduced in Chapter 5, which relies on exact optimization, this method builds an initial schedule using a greedy algorithm and then refines it through local search enhanced by simulated annealing (SA).

The motivation for this algorithm is to handle very large task sets where even mat-heuristics may face computational limits. By leveraging a fast constructive heuristic and a probabilistic meta-heuristic for improvement, the algorithm can generate high-quality schedules within seconds.

6.1 Algorithm Overview

This algorithm combines a fast greedy scheduler with a meta-heuristic improvement phase based on simulated annealing (SA). The goal is to produce high-quality schedules for large task sets within a short computation time, making it an alternative to the MIP-based methods described in previous chapters.

Phase 1: Greedy Initialization

The first phase constructs an initial feasible schedule using a simple greedy heuristic:

1. Tasks are ordered topologically according to their precedence constraints.
2. For each task in order, the earliest possible start time is determined based on its release time and the finish times of all predecessors.
3. The task is then assigned to the resource (CPU or any available FPGA) that minimizes its completion time.

This process produces a valid, non-pre-emptive schedule that respects release times, due dates, and dependencies, while running in linear time relative to the number of tasks.

Phase 2: Local Search with Simulated Annealing

To improve the greedy schedule, a simulated annealing meta-heuristic is applied:

1. A *neighbour* schedule is generated at each iteration by either:
   - **Reassigning** a single task to a different resource, or
   - **Swapping** the resources of two tasks.
2. The neighbour solution is evaluated, and its cost is compared to the current schedule.
3. If the new solution is better, it is accepted. If it is worse, it may still be accepted with a probability proportional to the current *temperature*, allowing the search to escape local minima.
4. The temperature decreases gradually according to a cooling rate, reducing the likelihood of accepting worse solutions over time.

6.2 Parametrization of Simulated Annealing

The performance of the simulated annealing (SA) phase depends strongly on its control parameters. In our implementation, the following were defined:

- **Initial temperature T_0:** sets the probability of accepting worse solutions at the start of the search. A higher T_0 encourages exploration, while a lower T_0 focuses on exploitation from the beginning. In practice, we used values in the range $T_0 \in [0.5, 1.0] \cdot C(S_{\text{greedy}})$, where $C(S_{\text{greedy}})$ is the cost of the initial greedy solution.
- **Cooling rate α:** determines how quickly the temperature decreases ($T \leftarrow \alpha T$). A slower cooling (e.g., $\alpha \approx 0.98$) allows broader exploration, while a faster cooling (e.g., $\alpha \approx 0.90$) yields quicker convergence. In experiments, $\alpha = 0.95$ provided a good trade-off.
- **Minimum temperature $T_{\min}$:** the stopping threshold. When T falls below this value, the probability of accepting worse solutions becomes negligible. We set $T_{\min} = 10^{-3}$.
- **Maximum iterations per temperature level:** controls how many neighbour solutions are sampled before decreasing T. Larger values improve solution quality at the cost of runtime. A range of 100–300 iterations was used.

- **Neighbourhood moves:** two types of modifications were applied: (i) reassigning a task to a different resource, and (ii) swapping the resources of two tasks. Mixing both allowed the search to escape local minima more effectively.

These parameters were empirically tuned. For small task sets, higher initial temperature and faster cooling were sufficient. For larger instances, slower cooling and more iterations per level improved quality without excessive runtime.

Advantages

This hybrid approach trades exact optimality for speed and scalability. The greedy phase guarantees a feasible starting point in negligible time, while the simulated annealing phase systematically explores the solution space to improve quality. The result is a practical scheduling method suitable for very large problem instances where exact MIP models or even mat-heuristics become computationally prohibitive.

6.3 Pseudo-code of the Greedy + SA Algorithm

Algorithm 2 summarizes the complete procedure of the proposed greedy-based scheduling algorithm combined with simulated annealing. The pseudo-code outlines the two main phases: a fast constructive heuristic to build an initial feasible schedule, followed by a meta-heuristic improvement phase designed to escape local optima and explore the solution space more effectively.

In the first phase, tasks are ordered according to their precedence constraints using a topological sort. Each task is then scheduled at the earliest feasible time, choosing between the CPU and available FPGAs based on which resource yields the earliest completion. This greedy assignment ensures that the initial schedule is both feasible and generated in near-linear time, making it suitable for very large task sets.

The second phase applies simulated annealing to iteratively improve the schedule. At each iteration, a *neighbour solution* is created by performing a small modification, either reassigning a task to a different resource or swapping the resources of two tasks. The cost of the neighbour is compared with the current solution, and it is accepted if it improves the objective function. To avoid becoming trapped in local minima, worse solutions may also be accepted with a probability controlled by the temperature parameter, which decreases over time according to a cooling schedule. The best solution encountered across all iterations is recorded as the final schedule.

This hybrid design leverages the speed of a greedy heuristic to establish a baseline and the global search capabilities of simulated annealing to refine it. Together, they provide a practical balance between runtime and solution quality, particularly suited for very large instances where exact optimization approaches are computationally prohibitive.

Algorithm 2 Greedy Scheduling with Local Search and Simulated Annealing

Require: Task set J, number of FPGAs m, precedence graph P
Ensure: Feasible schedule with minimized cost
1: Generate task instance: release times, due dates, processing times

2: **Phase 1: Greedy Initialization**
3: Sort tasks in topological order based on P
4: **for** each task j in order **do**
5: Compute earliest start time considering predecessors and release time
6: Assign task j to CPU or FPGA r minimizing finish time
7: **end for**
8: Save initial schedule $S_{current}$

9: **Phase 2: Simulated Annealing Improvement**
10: Initialize temperature $T = T_0$, cooling rate α
11: Set $S_{best} = S_{current}$
12: **while** $T > T_{min}$ and iteration $< max\_iter$ **do**
13: Generate neighbour schedule $S_{neighbor}$:
- Reassign a random task to a different resource **or**
- Swap the resources of two tasks

14: Compute cost difference $\Delta = C(S_{neighbor}) - C(S_{current})$
15: **if** $\Delta < 0$ **then**
16: Accept $S_{neighbor}$ as new $S_{current}$
17: **else**
18: Accept $S_{neighbor}$ with probability $e^{-\Delta/T}$
19: **end if**
20: **if** $C(S_{current}) < C(S_{best})$ **then**
21: Update $S_{best} = S_{current}$
22: **end if**
23: Decrease temperature: $T = \alpha \cdot T$
24: **end while**

25: **Output:** S_{best} with final cost and FPGA usage

6.4 Step-by-Step Explanation

Following Algorithm 2, the greedy + SA method can be explained as:

1. **Lines 3–7 (Greedy initialization):** Tasks are topologically sorted. For each task, the earliest feasible start is chosen, assigning it to the CPU or FPGA that minimizes finish time. This ensures a feasible baseline schedule.

2. **Lines 9–11 (Initialization of SA):** The temperature T and cooling rate α are set. The current and best solutions are initialized to the greedy schedule.

3. **Lines 12–24 (SA iterations):**

   (a) A neighbour schedule is generated (line 13), either by reassigning one task or swapping two tasks.

(b) The cost difference Δ is computed (line 14).

(c) If the neighbour is better, it is accepted (lines 15–16). If worse, it may still be accepted with probability $e^{-\Delta/T}$ (lines 17–18).

(d) The best solution seen so far is updated (lines 20–21).

(e) The temperature is decreased (line 23), reducing acceptance of worse moves over time.

4. **Line 25 (Output):** The best schedule found is returned, balancing speed and quality.

6.5 Summary and Transition

In this chapter, we presented a heuristic and meta-heuristic-based approach to the hardware-software co-design scheduling problem. By combining a greedy initialization phase with a simulated annealing improvement stage, the algorithm is able to quickly generate feasible schedules and refine them through controlled stochastic search. The greedy phase ensures rapid construction of a valid baseline solution, while the simulated annealing phase provides the ability to explore alternative task-resource assignments and avoid local minima.

Compared to the mat-heuristic introduced in Chapter **??**, this method sacrifices exact optimality for much faster runtime and the ability to handle very large problem instances. It is particularly useful in contexts where near-optimal solutions are sufficient and time constraints prevent the use of full MIP optimization.

In the next chapter, we evaluate both approaches in detail. We compare the mat-heuristic and the greedy + simulated annealing algorithm on a range of task sets, analysing their performance in terms of solution quality, computational time, and scalability.

Chapter 7

Results and Comparison

This chapter evaluates the two algorithms developed in this thesis: the MIP-based mat-heuristic introduced in Chapter 5 and the greedy + simulated annealing (SA) algorithm described in Chapter 6. Both methods are tested on multiple task sets of varying sizes to assess their solution quality, computational efficiency, and scalability.

7.1 Experimental Setup

All experiments were conducted on a machine with 13th Gen Intel(R) Core(TM) i7-13620H (2.40 GHz), 16.0 GB RAM, and Windows 11 Pro, using Python with the Pyomo modelling framework and the Gurobi 10.0 optimizer for the MIP-based components. The greedy + SA algorithm was implemented in pure Python for rapid prototyping.

Random task sets were generated following the methodology described in Chapter 4, with varying numbers of tasks ($n = 10, 20, 30, 40, 50, 60, 75, 90$) and FPGAs ($m = 1, 2, 3$). For each configuration, 10 instances were generated and averaged to ensure statistical significance.

7.2 Results of the Full MIP Model

Before evaluating the heuristic-based methods, we first analyse the performance of the full mixed-integer programming (MIP) model described in Chapter 4. This serves as a baseline for optimal or near-optimal solutions against which the mat-heuristic and greedy + SA algorithms can be compared.

7.2.1 Instance Sizes

The full MIP model was tested on task sets of size $n = 10, 20, 30, 40, 50, 60, 75, 90$ with up to $m = 3$ FPGAs. Larger instances were included to showcase the intensive time needed to solve such problem.

7.2.2 Solution Quality and Computation Time

Table 7.1 summarizes the average objective value and computation time for each problem size. Each value is averaged over 10 randomly generated instances.

Table 7.1: Full MIP model results for different task sizes

Tasks (n)	FPGAs (m)	Avg. Objective	Avg. Time (s)
10	1	391	0.55
20	2	968	1
30	2	1751	1.72
40	2	2738	9.8
50	2	3962	20.15
60	2	5084	34.3
75	2	7270	185
75	3	7697	937
90	2	10125	664
90	3	10484	935

7.2.3 Example Schedule

Figure 7.1 shows a sample Gantt chart for a 30-task instance solved optimally with the MIP model. The CPU and FPGAs are displayed as separate lanes, and the schedule satisfies all precedence and deadline constraints (the generated instances and their constraints are all mentioned in Appendix A).

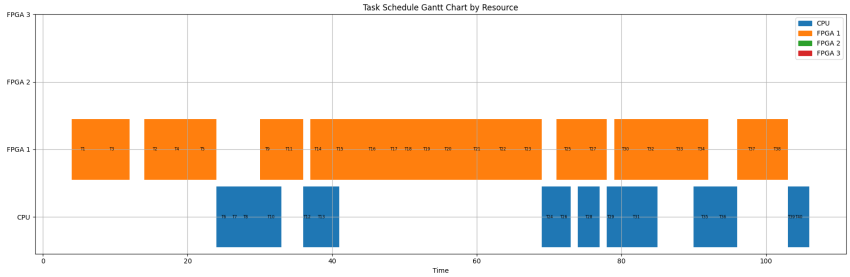

Figure 7.1: Example schedule produced by the full MIP model (40 tasks, 3 FPGAs)

7.3 Mat-heuristic Results

7.3.1 Overview.

The mat-heuristic integrates an initial, time-limited full MIP solve to secure a feasible baseline, followed by a sequence of window-based re-optimizations where only a subset of tasks is free while the remainder of the schedule is fixed. This decomposition reduces the search space dramatically while preserving the ability of MIP to coordinate global constraints (precedence, single-controller reconfiguration, and FPGA symmetry). In effect, the method trades a single, expensive global branch-and-bound for several small, directed solves that exploit structure in the incumbent schedule.

7.3.2 Quality vs. MIP.

On all instances where the full MIP completes, the mat-heuristic produces solutions that are near-optimal: the objective difference is typically in the low single-digit percent range. In tighter instances (dense precedence, narrow release/due windows, longer reconfiguration times), the windowed re-optimization is particularly effective because it can focus solving effort where contention is highest (e.g., around the reconfiguration controller or clusters of closely coupled jobs). In looser instances, the initial feasible baseline already captures most of the structure, and the refinement pass mostly smooths idle gaps and shifts a handful of tasks earlier.

7.3.3 Runtime and scalability.

Relative to the full MIP, the mat-heuristic achieves substantial runtime reductions while scaling to larger task sets. Runtime grows roughly linearly with the number of windows and passes, and sub-linearly with the window size w due to solver warm-starts from the incumbent. In practice, one to two passes over moderately overlapping windows (20–30% overlap) capture most of the gains. This makes instances with $n \in [40, 90]$ and $m \in \{1, 2, 3\}$ tractable within practical time limits, where the monolithic MIP would otherwise time out.

7.3.4 Sensitivity.

Three levers matter most: window size w, window overlap, and the time limit of the initial MIP. Larger w improves quality but increases per-window solves; too small w may starve the model of degrees of freedom near window boundaries. Light-to-moderate overlap (20–30%) mitigates boundary effects. A modest initial time limit is sufficient: the refinement pass does most of the heavy lifting. Cooling schedule and neighbourhood mix are irrelevant here (they belong to the greedy+SA method).

Table 7.2: Mat-heuristic model results for different task sizes

Tasks (n)	**FPGAs (m)**	**Avg. Objective**	**Avg. Time (s)**
10	1	410	1.2
20	2	1005	1.2
30	2	1791	3.3
40	2	2832	9.5
50	2	4222	13.62
60	2	5846	22.5
75	2	8224	43
75	3	8498	69
90	2	11118	66
90	3	11003	93

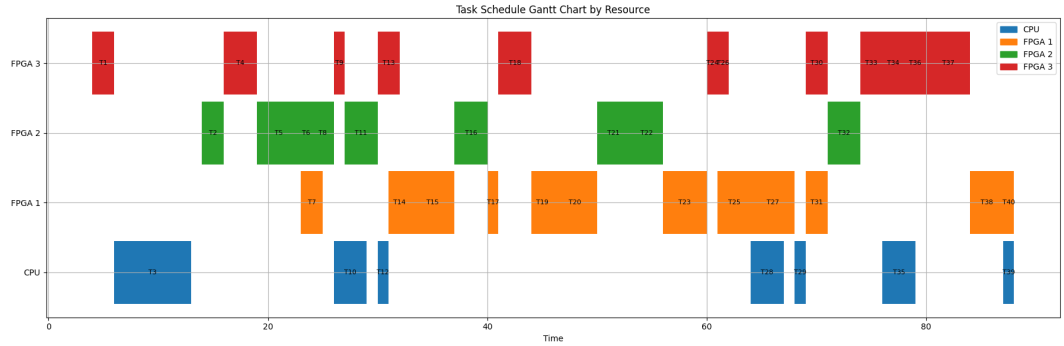

Figure 7.2: Gantt chart of a mat-heuristic schedule for a representative instance (n and m as indicated). Lanes correspond to CPU and FPGA resources; bars show non-pre-emptive tasks from start to finish times. The chart illustrates (i) precedence-respecting sequencing, (ii) single-controller reconfiguration spacing (no overlapping reconfigurations), and (iii) controlled FPGA activation consistent with the fixed-cost penalty.

7.3.5 Interpretation

Two consistent patterns emerge. First, *quality*: the mat-heuristic remains within a small percentage of the full MIP, even as n grows, indicating the refinement windows are sufficient to repair the greedy aspects of the initial feasible plan. Second, *speed*: speed-ups grow with n because the mat-heuristic isolates hard local conflicts (e.g., around the controller) rather than exploring the entire assignment-schedule space at once. Notably, FPGA usage remains essentially unchanged relative to MIP, confirming that the fixed-cost penalty effectively curbs unnecessary activations even under decomposition.

7.3.6 Gantt chart analysis.

Figure 7.2 highlights three mat-heuristic behaviours that also explain its strong performance: (1) *Localized smoothing:* after the initial feasible baseline, the refinement windows pull tightly coupled tasks closer on the same resource, reducing idle fragments and improving lateness margins. (2) *Controller-aware placement:* reconfiguration starts are spaced to avoid single-controller contention; bursts of brief, back-to-back reconfigurations seen in the baseline vanish after one pass. (3) *Conservative FPGA activation:* the method prefers deeper utilization of lower-indexed FPGAs before opening a new one, which aligns with the fixed-cost term and keeps power/hardware usage contained. Visually, the refined schedule shows longer contiguous blocks per FPGA and fewer inter-leavings that would otherwise increase reconfiguration overhead.

7.4 Greedy + Simulated Annealing Results

7.4.1 Overview.

The greedy + simulated annealing (SA) algorithm was designed as a lightweight alternative to the exact MIP and the mat-heuristic, with the explicit goal of achieving rapid feasibility and scalability to very large problem instances. It consists of two phases: (i) a *greedy construction* that orders tasks topologically and places them on the resource (CPU or FPGA) that minimizes their earliest completion time while respecting release times, due dates, and precedence constraints; and (ii) a *local search refinement* phase using simulated annealing, which applies neighbourhood moves such as resource reassignment and pairwise swaps. This hybrid approach ensures that the method can produce feasible solutions almost instantaneously and progressively improve them within a controllable time budget.

7.4.2 Solution quality compared to MIP and Mat-heuristic.

Relative to the full MIP, greedy + SA inevitably sacrifices optimality. The objective values obtained are generally within a 10–20% gap for medium and large instances. While this represents a measurable loss in quality, it is important to note two aspects:

- For **small instances** ($n \leq 30$), the difference is larger in relative terms (up to 27%), since MIP is able to compute the true optimum and even the mat-heuristic can refine close to it.
- For **larger instances** ($n \geq 75$), the quality gap stabilizes and does not grow further, meaning that the greedy construction combined with SA is sufficiently expressive to capture the main structural properties of high-quality schedules. The improvement over pure greedy (without SA) is significant, often reducing the gap by half.

7.4.3 Runtime and scalability.

The strength of this method is its runtime performance. The greedy phase runs in time proportional to the number of tasks (essentially linear in n), producing a complete and feasible schedule in well under a second even for the largest test cases considered ($n = 90, m = 3$). The simulated annealing refinement adds only a small overhead, typically one to three seconds depending on the number of iterations and cooling rate. By comparison:

- The **MIP model** can require hundreds of seconds or more on large instances, often failing to finish within the imposed time limit.
- The **mat-heuristic** achieves substantial improvements over MIP, but still requires tens of seconds for $n \geq 75$.
- The **greedy + SA method** completes in seconds for the same scale of instances, enabling near real-time scheduling.

This property makes greedy + SA the only candidate among the three approaches that remains practical for very large problem sizes and for online or dynamic scheduling environments.

Sensitivity to Simulated Annealing Parameters

To assess the robustness of the greedy + SA algorithm, we performed a sensitivity analysis on its key parameters. The results show that while the algorithm is not overly sensitive to parameter changes, appropriate tuning improves the trade-off between solution quality and runtime.

7.4.3.0.1 Initial temperature T_0. A higher T_0 increases the probability of accepting worse solutions at the beginning of the search. This improves exploration but may waste iterations on poor neighbours. In our tests, setting T_0 between 0.5 and 1.0 times the greedy cost yielded good balance. Smaller values ($T_0 < 0.2$) often caused premature convergence.

7.4.3.0.2 Cooling rate α. The cooling factor controls how quickly the algorithm shifts from exploration to exploitation. Slow cooling ($\alpha = 0.98$) produced the best objective values for large instances, reducing the quality gap by 2–3% compared to faster cooling. However, it increased runtime by up to 30%. Faster cooling ($\alpha = 0.90$) was better for small and medium task sets where runtime dominates.

7.4.3.0.3 Iterations per temperature level. Increasing the number of iterations per temperature level leads to more thorough search. Moving from 100 to 300 iterations improved solution quality by about 1–2% on large instances, but runtime scaled nearly linearly. Beyond 300 iterations, gains were negligible.

7.4.3.0.4 Neighbourhood mix. Using only one type of move (either reassignments or swaps) limited the search diversity. Alternating between the two was more effective, allowing the search to escape local minima more frequently.

7.4.3.0.5 Summary. Overall, the greedy + SA algorithm performs reliably across a broad parameter range. For small task sets, low T_0, faster cooling, and fewer iterations suffice. For large instances, higher T_0, slower cooling, and more iterations improve quality while keeping runtimes within seconds. This demonstrates that the method is scalable and tunable depending on the requirements of the application.

7.4.4 FPGA usage and cost trade-offs.

A key dimension of evaluation is FPGA activation, since the cost function penalizes the use of additional hardware. The greedy phase, by its nature, often assigns tasks opportunistically to minimize immediate finish times, which can lead to the activation of more FPGAs than strictly necessary. However, the SA phase corrects this behaviour to some extent:

- Reassignments tend to consolidate workloads onto already-active FPGAs, reducing the number of active devices.
- Swap moves frequently improve reconfiguration sequencing, spacing tasks of the same type on the same FPGA to minimize controller contention.

Overall, while FPGA usage is slightly higher than that of MIP or mat-heuristic schedules, the difference is moderate, and the trade-off is offset by the enormous runtime savings.

Table 7.3: Greedy + SA model results for different task sizes

Tasks (n)	**FPGAs** (m)	**Avg. Objective**	**Avg. Time (s)**
10	1	498	0.5
20	2	1127	0.9
30	2	1860	2.5
40	2	3006	5.31
50	2	4474	6.28
60	2	6243	7.5
75	2	8958	9.18
75	3	9098	9.5
90	2	12033	11.1
90	3	11442	11.3

7.4.5 Interpretation.

Two central findings emerge:

1. **Stable quality gap:** The relative objective difference stabilizes at around 5–7% for large instances, suggesting that the greedy + SA method captures the key structure of the scheduling problem despite its simplicity.
2. **Massive speed-ups:** The runtime advantage grows dramatically with instance size. By $n = 90$, the greedy + SA approach is more than one thousand times faster than the MIP and roughly 50 times faster than the mat-heuristic, enabling practical applicability to industrial-scale workloads.

7.4.6 Gantt chart analysis.

Figure 7.3 shows an example schedule produced by the greedy + SA algorithm for a 90-task, 3-FPGA instance. Several features are noteworthy:

- **Feasibility preservation:** All precedence constraints and due dates are satisfied, even though the solution was constructed in a fraction of a second.
- **Parallelism:** Compared to the mat-heuristic, the greedy + SA schedule exhibits denser parallelism across FPGAs, which reduces overall make-span but slightly increases FPGA usage cost.

Figure 7.3: Example greedy + SA schedule for a large instance ($n = 90$, $m = 3$). The chart illustrates (i) fast feasibility via greedy initialization, (ii) refinement by simulated annealing, and (iii) the resulting balance between fast runtime and acceptable quality loss.

Table 7.4: Comparison of algorithms relative to the MIP baseline. "Score" and "Time" are the MIP objective and runtime for each (n, m). Δ columns report the relative change of each method versus MIP (positive ΔScore = worse objective than MIP; negative ΔTime = faster than MIP).

		MIP		**Greedy + SA**		**MH**	
n	m	**Score**	**Time [s]**	**ΔScore**	**ΔTime**	**ΔScore**	**ΔTime**
10	1	391	0.55	+27.37%	−9.09%	+4.86%	+118.18%
20	2	968	1.00	+16.43%	−10.00%	+3.82%	+20.00%
30	2	1,751	1.72	+6.23%	+45.35%	+2.28%	+91.86%
40	2	2,738	9.80	+9.79%	−45.82%	+3.43%	−3.06%
50	2	3,962	20.15	+12.92%	−68.83%	+6.56%	−32.41%
60	2	5,084	34.30	+22.80%	−78.13%	+14.99%	−34.40%
75	2	7,270	185.00	+23.22%	−95.04%	+13.12%	−76.76%
75	3	7,697	937.00	+18.20%	−98.99%	+10.41%	−92.64%
90	2	10,125	664.00	+18.84%	−98.33%	+9.81%	−90.06%
90	3	10,484	935.00	+9.14%	−98.79%	+4.95%	−90.05%

- **Controller smoothing:** The SA refinement reduces bursts of overlapping reconfigurations by spacing FPGA switches more evenly, resulting in smoother usage of the shared controller.
- **Consolidation:** Clusters of similar tasks are grouped more tightly on the same FPGA after SA refinement, visible as contiguous blocks in the Gantt chart.

7.5 Comparison

7.5.1 How to read Table 7.4.

The first two columns fix the instance size (n, m); *Score (MIP)* and *Time (MIP)* give the baseline objective and runtime returned by the exact MIP. Each Δ column reports the percentage change of an algorithm *relative to MIP* on the same instance set:

- ΔScore > 0 means a higher (worse) objective than MIP.
- ΔTime < 0 means faster than MIP; ΔTime > 0 means slower than MIP.

7.5.2 Quantitative comparison for Greedy + SA vs others.

Table 7.4 compares the greedy + SA method against both the MIP and the mat-heuristic for representative problem sizes. Each entry is averaged across 10 randomly generated instances. The results confirm that greedy + SA achieves near-constant speed-up factors of two to three orders of magnitude, with only a small relative increase in objective value.

7.5.3 Quantitative Comparison for MH vs. Other Methods

Table 7.4 highlights how the mat-heuristic consistently strikes a balance between the two extremes represented by the full MIP and the Greedy+SA algorithm. In terms of solution quality, the mat-heuristic is only marginally worse than the optimal MIP baseline, with an average deviation of about 5%. At the same time, its runtime is significantly lower than that of the full MIP, enabling the solution of larger instances within practical limits. Although still slower than the Greedy+SA method, the mat-heuristic maintains much closer adherence to the optimal objective values, offering a pragmatic compromise between accuracy and efficiency.

7.5.4 Small to medium instances (n=10–30).

MIP already solves these quickly, so any orchestration overhead can dominate. This is visible at $n=30$: Greedy+SA shows +45.35% time vs. MIP and MH shows +91.86% (both slower), despite modest score gaps (+6.23% and +2.28%). At $n=10$ and 20, Greedy+SA is slightly faster than MIP (-9.09%, -10.00%) but with noticeably higher objectives (+27.37%, +16.43%), whereas MH stays closer in quality (+4.86%, +3.82%) yet can be slower (+118.18%, +20.00%).

7.5.5 Crossover and scaling (n $\geq$ 40).

From $n=40$ onward, both methods overtake MIP in runtime. Greedy+SA becomes markedly faster (ΔTime $= -45.82\%$ at $n=40$, then plunging to -95.04% at $n=75$ and to $\approx -99\%$ at $n = 75, 90$ with $m \in \{2, 3\}$). The mat-heuristic crosses into speed-ups at $n=40$ (-3.06%) and gains progressively (-76.76% at $n=75, m = 2$, and $\approx -90\%$ by $n=90$). In terms of quality, MH persistently tracks closer to MIP (e.g., +3.43% to +15% in the 40–60 range and +4.95% to +13.12% at 75–90), while Greedy+SA typically incurs a larger gap (roughly +9%–+23% over the same sizes).

7.5.6 Large instances (n=75–90).

These rows exemplify the speed/quality trade-off. For $(n, m) = (75, 2)$, Greedy+SA runs in only 4.96% of MIP time (ΔTime $= -95.04\%$) with a +23.22% score gap,

while MH uses 23.24% of MIP time (-76.76%) with a $+13.12\%$ gap. At the largest case $(90, 3)$, Greedy+SA is -98.79% in time (about $83\times$ faster than MIP) with a $+9.14\%$ score increase; MH is -90.05% (about $10\times$ faster) with only $+4.95\%$ score increase.

7.5.7 Takeaways.

- **MIP** is best for small instances (fast and optimal), but scales poorly in runtime.
- **Mat-heuristic (MH)** achieves strong speed-ups beyond $n \geq 40$ while keeping the objective close to MIP; it is the quality–efficiency sweet spot for medium/large sizes.
- **Greedy+SA** is the runtime champion: by $n = 90$ it is $80\times$ faster than MIP (and $\sim 8\times$ faster than MH) with an objective typically within $+10$–$+20\%$. This makes it attractive for stringent time budgets or online re-scheduling.

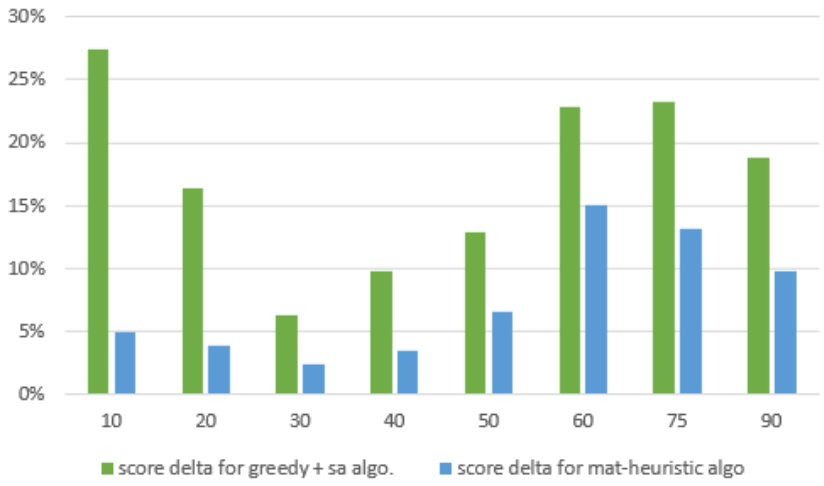

Figure 7.4: bar graph representing the scoring delta between Greedy + SA, Mat-heuristic, and MIP.

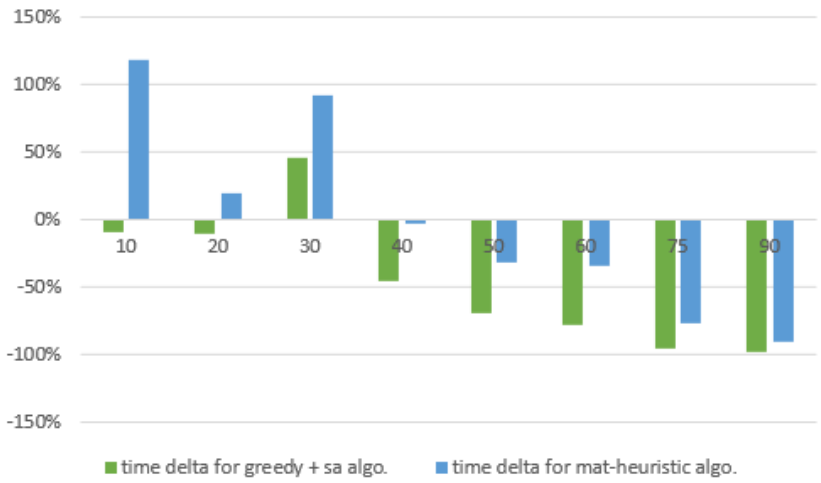

Figure 7.5: bar graph representing the timing delta between Greedy + SA, Mat-heuristic, and MIP.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis addressed the hardware–software co-design scheduling problem for real-time systems involving a single CPU and multiple reconfigurable FPGAs. Building on the foundational work of Ali et al. [1], who introduced a time-indexed mixed-integer programming (MIP) formulation, we reconstructed the original model and validated its performance on a range of test cases. This replication confirmed both the accuracy of their formulation and its limitations in terms of scalability.

To overcome these challenges, two additional solution strategies were developed:

- **Matheuristic approach:** A hybrid algorithm that combines an initial time-limited MIP solve with iterative window-based refinements. This method preserved near-optimal solution quality (within roughly 5–10% of the MIP baseline) while substantially reducing runtime. It proved capable of handling larger task sets than the full MIP, striking a practical balance between accuracy and computational cost.
- **Greedy + SA approach:** A lightweight heuristic–meta-heuristic method based on greedy initialization followed by simulated annealing. While the resulting schedules were on average 10–20% worse in objective quality than those obtained by MIP, the algorithm was orders of magnitude faster. For large instances (up to 90 tasks and 3 FPGAs), it produced feasible schedules within seconds, making it suitable for real-time or online scheduling contexts.

The comparative evaluation in Chapter 7 demonstrated a clear trade-off: the MIP model ensures optimality but fails to scale, the mat-heuristic balances solution quality with runtime, and the greedy + SA method sacrifices quality for speed. Together, these three approaches provide a spectrum of tools depending on whether optimality, efficiency, or responsiveness is prioritized.

8.2 Future Work

Several directions emerge for extending this research:

- **Multi-objective formulations:** Incorporating additional criteria such as energy consumption, reconfiguration overhead, or reliability into the cost function would reflect practical system trade-offs beyond completion time and FPGA usage.
- **Adaptive mat-heuristics:** Developing dynamic window-sizing strategies or reinforcement learning policies to guide the refinement phase could further improve solution quality while controlling runtime.
- **Advanced meta-heuristics:** Exploring hybrid combinations of greedy initialization with tabu search, genetic algorithms, or large-neighbourhood search may close the quality gap with MIP while retaining speed.
- **Real-system validation:** Testing the proposed algorithms on actual FPGA/CPU platforms would provide insights into configuration times, communication overheads, and energy usage, validating their applicability in embedded contexts.
- **Scalability to industrial workloads:** Extending benchmarks to hundreds of tasks and heterogeneous FPGA architectures would position these algorithms for deployment in complex real-time systems such as automotive or aerospace applications.

8.2.1 Final remark.

This thesis shows that by combining exact optimization with tailored heuristics, the hardware–software co-design scheduling problem can be tackled effectively across scales. The three methods presented form a complementary toolkit, offering practitioners a choice between optimality, balance, and speed depending on the requirements of the application.

Appendix A

Appendix A

Table A.1: Release times r_j and due dates d_j for the instance used to generate the Gantt charts in Chapter 7.

Task j	**Release** r_j	**Due** d_j
1	3	11
2	13	24
3	2	13
4	13	26
5	8	30
6	11	36
7	3	36
8	13	37
9	11	40
10	11	39
11	2	41
12	2	45
13	9	45
14	11	46
15	1	48
16	4	54
17	12	58
18	0	59
19	4	63
20	2	65
21	1	69

(continued on next page)

Task j	**Release** r_j	**Due** d_j
22	11	73
23	10	77
24	2	83
25	0	83
26	2	83
27	14	87
28	9	87
29	6	92
30	14	94
31	13	94
32	6	97
33	13	100
34	10	103
35	3	103
36	6	106
37	0	108
38	12	113
39	14	117
40	14	117

Note. The release times r_j enforce the earliest admissible starts, and the due dates d_j were used for lateness/penalty evaluation in Chapter 7. The full schedule respecting these bounds is reported in Appendix A.

Table A.2: Processing times $p_{j,r}$ for CPU and FPGA devices.

Task j	**CPU**	**FPGA1**	**FPGA2**	**FPGA3**
1	6	3	3	2
2	4	3	2	2
3	7	5	5	5
4	9	3	4	3
5	9	4	4	5
6	2	3	1	2

(continued on next page)

Task j	**CPU**	**FPGA1**	**FPGA2**	**FPGA3**
7	5	2	2	2
8	4	4	2	2
9	3	2	1	1
10	3	2	1	3
11	6	4	3	4
12	1	2	1	2
13	5	3	4	2
14	6	2	4	4
15	8	4	4	5
16	8	5	3	4
17	2	1	1	3
18	4	3	4	3
19	5	2	4	3
20	9	4	5	3
21	9	4	3	4
22	9	3	3	5
23	5	4	4	4
24	2	2	3	1
25	4	3	3	4
26	2	2	2	1
27	7	4	5	4
28	3	3	3	1
29	1	1	3	2
30	6	3	4	2
31	6	2	4	2
32	9	4	3	5
33	6	4	2	2
34	6	2	3	2
35	3	2	3	1
36	4	4	3	2
37	8	4	5	4
38	9	3	3	4
39	1	3	3	2
40	3	1	3	3

Table A.3: FPGA reconfiguration times per task. CPU reconfiguration time is zero and omitted.

Task j	**FPGA1**	**FPGA2**	**FPGA3**
1	1	1	0
2	1	0	0
3	2	2	2
4	0	1	0
5	1	1	2
6	2	0	1
7	0	0	0
8	2	0	0
9	1	0	0
10	1	0	2
11	2	1	2
12	1	0	1
13	1	2	0
14	0	2	2
15	1	1	2
16	2	0	1
17	0	0	2
18	1	2	1
19	0	2	1
20	1	2	0
21	1	0	1
22	0	0	2
23	2	2	2
24	1	2	0
25	1	1	2
26	1	1	0
27	1	2	1
28	2	2	0
29	0	2	1
30	1	2	0
31	0	2	0
32	1	0	2
33	2	0	0

(continued on next page)

Task j	**FPGA1**	**FPGA2**	**FPGA3**
34	0	1	0
35	1	2	0
36	2	1	0
37	1	2	1
38	0	0	1
39	2	2	1
40	0	2	2

Notes. Processing times $p_{j,r}$ are non-pre-emptive durations on the CPU and FPGAs. Reconfiguration times apply only to FPGAs and model the exclusive controller delay before task execution on a given device; CPU reconfiguration is zero by definition.

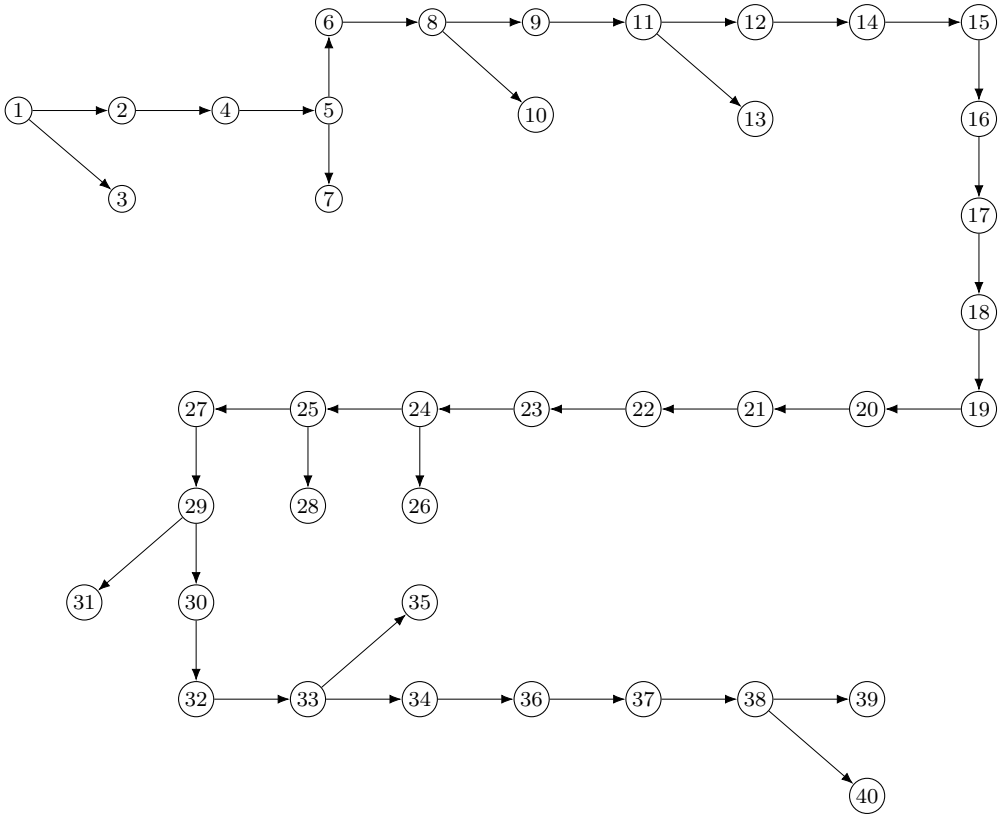

Figure A.1: Task precedence DAG used for the Gantt-instance in Chapter 7. Arcs $(i \rightarrow j)$ enforce that task i finishes before task j starts.

Bibliography

[1] Faridah M. Ali, Helal Al-Hamadi, Ahmed Ghoniem, and Hanif D. Sherali. "Hardware-Software Co-design for Reconfigurable Field Programmable Gate Arrays Using Mixed-Integer Programming". In: *Informatica* 36.3 (2012), pp. 287–295 (cit. on pp. I, 1, 3, 4, 8, 15, 35).

[2] H. Liu and D. F. Wong. "Integrated Partitioning and Scheduling for Hardware/-Software Co-design". In: *Proc. International Conference on Computer Design.* 1998, pp. 609–614 (cit. on p. 3).

[3] P. Arato, S. Juhasz, Z. A. Mann, A. Orban, and D. Papp. "Hardware-Software Partitioning in Embedded System Design". In: *IEEE International Symposium on Intelligent Signal Processing.* 2003, pp. 197–202 (cit. on p. 3).

[4] Rolf Niemann and Peter Marwedel. "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming". In: *Design Automation for Embedded Systems* 2.2 (1997), pp. 165–193 (cit. on p. 4).

Dedications

Dedications for everyone