Universität des Saarlandes

Master Thesis

Optimal Offline Scheduling for Multi-Core Systems with Dynamically Partitioned Caches

---

Author:
Darshit Shah

Supervisor:
Prof. Dr. Jan Reineke

Reviewer:
Prof. Dr. Jörg Hoffmann

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Real-Time and Embedded Systems Lab

June 4, 2019

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Unterschrift/Signature:

_____

Datum/Date:

_____

UNIVERSITÄT DES SAARLANDES

Abstract

Real-Time and Embedded Systems Lab

Master of Science

by Darshit Shah

Cache partitioning is often used in multi-core systems to improve predictability. Prior work has mostly focused on static per-core cache partitioning. We conjecture that there is significant remaining potential if the cache is partitioned at the granularity of individual jobs. The problem of finding the optimal cache partition and core assignment for a set of jobs is NP-Hard. In this thesis, we discuss our ideas for a new algorithm—an extension of the A* algorithm—to find optimal schedules for non-recurrent sets of jobs. We demonstrate how the A* algorithm can be adapted to the problem of real-time scheduling and present some methods on how to better guide the algorithm towards the goal of a optimal schedule. Several state-space pruning approaches, which can help the overall performance of the search, are also discussed. Finally, we also demonstrate how these approaches can be modified to obtain feasible schedules with guarantees on their quality.

A part of the work in this thesis was also published at ECRTS-2018 [SR18], during the Work-In-Progress (WiP) session.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Jan Reineke for his supervision and guidance throughout this thesis. The support and excellent working environment provided by him were crucial to the work presented here.

I would be remiss to forget Sebastian Hahn for the countless hours of discussions around all topics under the sun and for proof reading this thesis.

Further thanks go to everyone in the Compiler Design Lab / Real-Time and Embedded Systems Group for the multitude of discussions related to this work and research in general.

I would also like to thank Prof. Dr. Jörg Hoffmann for agreeing to review this thesis.

Nithya Mogane for proof-reading this thesis and being my rubber duck[1] every time I needed to find a solution on my own.

Finally, I would like to thank my parents for their constant support and backing throughout my studies.

[1] `https://www.rubberduckdebugging.com`

Contents

List of Figures

Chapter 1

Introduction

Upon impact, one expects the airbag in their car to deploy at precisely the right time. Too early, and the airbag may prevent reaction from the driver, too late and it may result in the driver's death. Such systems which need to satisfy temporal constraints are known as real-time systems.

Real-time systems are divided into two categories, hard and soft real-time systems. Soft real-time systems are those that *should* meet most of their temporal constraints. Not meeting a few temporal constraints will degrade the quality of the system, but it will continue functioning without any major issues. Examples of soft real-time systems are audio/video players, weather recording stations. Hard real-time systems on the other hand *must* meet all of their temporal constraints or risk complete and catastrophic failure of the system. Examples of hard real-time systems are autonomous cars, all aviation and avionic systems, medical devices, etc. In many jurisdictions, manufacturers of such systems are required by law to prove the correct behaviour of their systems under critical conditions. In this thesis, we focus only on hard real-time systems.

Traditionally, such safety-critical systems have been deployed on single-core machines. However, in order to address size, weight, and power constraints (SWaP) there is a recent trend to transition from federated architectures, where each application is deployed on a private single-core processor, to integrated architectures, where multiple applications share a single multi-core platform. Today, multi-core chips are smaller, faster and cheaper than ever before. As a result, real-time systems are being increasingly deployed on such multi-core chips [Bak10].

In safety critical real-time systems this trend poses a major challenge for the verification process due to the possibility of interference on shared resources such as

buses [RMMA15], networks-on-chip [SS16], shared caches [WHKA13], and memory controllers [YMWP14]. Safely and tightly bounding this interference is extremely difficult as the shared resources and their hardware management mechanisms have usually not been designed with predictability in mind, and very often their internal workings are not publicly available either.

A rather general approach to address this challenge is to partition the shared resources for temporal and spatial isolation. By partitioning the resources, interference can be eliminated, making the use of multi-core platforms feasible. Such an approach is also advocated in the recent position paper on certification issues CAST-32A [Cer16], where it is referred to as *robust partitioning*.

In this thesis, we focus on the challenge of efficiently partitioning shared caches and scheduling in multi-core systems. The problem of finding a perfect schedule is NP-Hard. It has already been shown that this cannot be solved by an online scheduler in multi-core systems [DM89]. However, with the benefit of foresight, an offline scheduler may explore the entire state space to find the optimal schedule. Most prior work in this area takes the simple approach by partitioning caches statically among the cores. In this thesis, we first show that such partitioning leads to non-optimal use of the available hardware resources. Later, we present the idea of dynamically re-partitioning the cache at runtime in order to provide adequate cache capacity to individual jobs when needed.

Contributions We present a novel technique to adapt the A* path-finding algorithm to searching for the optimal schedule in multi-core hard real-time systems with dynamically partitioned caches. We show how A* can be used in this unique setting and present state-space pruning approaches to improve the performance of the algorithm. We also demonstrate two different heuristics for guiding the search towards the optimal schedule and evaluate their effectiveness.

Thesis Structure Chapter 2 introduces caches and the different ways in which prior work has accounted for caches during scheduling analysis. A brief introduction to the A* algorithm is presented in Section 2.3. Next, we formally describe the entire system model for this thesis in Chapter 3. We also formally state the problem and define the precise optimization criteria for schedules in Section 3.2. In Chapter 4 we describe our modifications to the A* algorithm and how it is adapted to the problem of finding the optimal schedule for multi-core systems. It also discusses our

implementation of the various parts of the modified A* algorithm in more detail. Chapter 5 introduces two different methods for early pruning of the state space. We then show a novel technique for finding a feasible schedule with bounded quality guarantees in Chapter 6. In Chapter 7 we present an evaluation of the performance of the techniques presented in this thesis and the various trade-offs that can be made in practice.

Chapter 2

Background and Related Work

2.1 Processor Caches

Over the last couple of decades, processor speeds have grown much faster than the speed to access main memory [LGBS05]. Modern systems thus use a hierarchy of small, but fast *caches* to store the most frequently used data. The principle of *temporal* and *spatial locality* tells us that memory regions that have been accessed will often be accessed again in the near future, and that adjacent locations also have a high probability of being accessed in the near future. Thus, storing the recently accessed memory locations along with its immediate neighbours in a fast cache, allows modern systems to amortise the long access times to main memory over many accesses.

Typically, a computer system has multiple layers of gradually larger caches between the processor core and the main memory. The L1 cache is usually core-private. That is, each core has a small, private, and extremely fast cache which stores data most likely to be used in the very-near future. The next levels of caches (L2, L3, ...) are progressively larger and slightly slower, and are usually shared among the different cores of a multi-core system.

When a core tries to access a memory location, it first tries to find it in the nearest cache. If it is found, this is known as a *cache-hit*. Else, it iteratively searches each of the other caches till it finds the data. In the worst case the memory location has not been cached; this is known as a *cache-miss* and forces an expensive access to the main memory to fetch the data. This data, and its adjoining memory locations are then cached for future reuse. If the cache is full, some older data must be *evicted* from it,

to make space for the new data. Caches use various *cache replacement policies* to decide which data gets selected for such eviction.

In the analysis of hard real-time systems, it is important to consider the role of the cache when evaluating the Worst-Case Execution Time (WCET) of tasks. Since caches significantly speed up the average memory access times, not considering them would lead to very pessimistic estimations of the WCET. Most WCET analysis tools try to predict whether a memory access will be a cache-hit or a cache-miss in an attempt to not be too pessimistic about the WCET estimations.

2.1.1 Cache Contention

In multi-core systems, accounting for the effects of shared caches adds a lot of unpredictability and pessimism to the analysis. Consider for example, a system with two cores executing different programs. It might be the case, that they keep trying to evict each others' data in the cache in order to make room for their own. This results in severe degradation of performance since both the cores need to keep accessing the main memory very frequently. This is known as cache contention.

Such severe contention on the shared cache is not only bad for performance but for analysis as well. Accounting for cache contention requires knowledge about all the other tasks that may be executing simultaneously on different cores. In order to deal with the unpredictability of shared caches in multi-core systems, prior work has generally picked one of the two following approaches:

1. Cache Partitioning: These approaches [MCHvE06, LPM09, BCSM08] partition the shared cache using either hardware or software techniques. In this way they achieve temporal isolation between different tasks, thereby easing timing verification.

2. Analysing Cache Contention: Other approaches [ACD06, DZ13, ZWN17, XAP17, NS14] compute bounds on the contention between different tasks on a shared cache and use these bounds within response-time analysis.

Precisely analysing cache contention is challenging, as very different cache states and cache behaviours may arise, depending on the precise relative timing of memory accesses coming from different cores. The problem is slightly easier if tasks are

scheduled non-preemptively. Still, even analyses assuming non-preemptive scheduling [XAP17] make use of coarse abstractions of the cache behaviour, and thus we expect them to be rather imprecise.

On the other hand, Nagar and Srikant [NS14] try to analyse a task to find the worst interference points for cache accesses. This however leads to highly pessimistic estimate of the WCET.

Cache partitioning is often the suggested way to improve the predictability of multi-core hard real-time systems. This is largely due to the fact that partitioning the cache helps prevent inter-task contention for the same location in the cache, thus simplifying the computation of the WCET of a task. Cache partitioning thus makes it possible to use existing analysis techniques for single-core systems to analyse tasks in a multi-core system. Even outside the realm of real-time systems, cache partitioning is used to improve the predictability and the overall throughput of systems [FAK+12, LCG+15, MCHvE06].

2.1.2 Cache Partitioning

Cache partitioning allows the system to provide spatial isolation across the different cores of a system. Instead of having one large cache that is shared across multiple cores, the system divides the cache into smaller parts and provides them to the various cores for exclusive, private access. As a result, a core may no longer evict data belonging to another core on a shared cache. Such isolation makes analysis of tasks significantly easier and helps reduce the estimated WCETs.

There are two cache partitioning methodologies: hardware- [Kir, LSK04, SKI08, SM08, RLT06, Lee16] and software- [KKR13, MKR10, LLD+, CM05, LHH97, GSYY09] based approaches.

In hardware partitioning, the processor provides the operating system with a set of instructions allowing it to define rules on how to partition the caches. It is then the duty of the processor itself to enforce such partitioning in a transparent manner. Intel has recently released COTS processors with built-in hardware support for cache partitioning [HVA+16, Int16].

On the other hand, software partitioning of the cache is done with the help of the compiler and the operating system. In this method, both the compiler and the operating system decide on a set of rules and cooperatively enforce them. There

exist a growing number of modules to add software-based cache partitioning support to the Linux Kernel [SKS16, PTH11]. Software partitioning is usually more flexible but incurs higher overhead than hardware partitioning.

When partitioning caches in real-time systems, we are aware of two lines of approach in prior work:

1. Per-core cache partitioning [BCSM08, LPM09]: Here, the shared cache is statically partitioned among the cores of a multi-core, and tasks are assigned to cores depending on their cache footprint.
2. Per-task cache partitioning [GSYY09]: Here, each task is allocated a certain share of the cache, and whenever a job of the task is scheduled, its share of the cache is made available to it privately.

Per-task cache partitioning is more flexible and may thus provide better performance.

The prior work by Guan et al. [GSYY09] considers the cache space allocated to each task to be an input to their non-preemptive scheduling algorithm. In other words, the amount of cache space allocated to each task is *not* optimised to globally improve schedulability.

Lokuciejewski et al. [LPM09] attempt to generate cache partitions using the WCET values of tasks with varying amount of cache available to them. However, they generate only static per-core cache partitions.

To our knowledge, none of the prior work has attempted to look at the schedulability of task sets using caches that can be dynamically repartitioned at runtime. Such a system can potentially improve the overall response time of a task set by co-scheduling tasks with complementary cache requirements on separate cores. Our aim is to fill this gap, by developing a novel method for computing offline schedules of sets of non-preemptible tasks using per-job cache partitioning.

2.2 Generalised Assignment Problem

The problem of assigning tasks to cores with a certain size of cache partitions is an instance of the *Generalised Assignment Problem* (GAP), which is a well-known NP-Hard problem. A naïve algorithm would need to explore the entire state space of all possible task assignments to find the optimal result. It is however possible to reduce the state space through some techniques.

Chou and Chung [CC94] proposed the idea for representing a set of tasks with precedence constraints as a DAG and to perform an exhaustive state-space search to find the optimal schedule. They also suggest using pruning techniques and domination relations in order to reduce the size of the state space. However, they assume each task to have a fixed execution time, preventing the approach from being used in more generalised scenarios.

Chand and Jiang [CJ94] also considered the idea of state-space searches for finding schedules of tasks on multiprocessor systems. However, they consider solving the NP-Complete problem from the optimal schedule to be too expensive and instead look for approximate approaches.

Kwok and Ahmed [KA05] propose the use of A* for navigating the DAG of a set of parallel tasks to find the optimal schedule. They also introduce the idea of *isomorphic schedules* as a state-space pruning approach. However, their approach is designed for a single parallel task with temporal and spatial precedence constraints, and does not generalise well to scheduling multiple independent tasks.

Moreover, none of the approaches described here account for the effect of caches on the execution time of the tasks. Neither are they designed for a hard real-time setting, where tasks must also finish before a prescribed deadline.

2.3 A* Algorithm

A* is a path-finding algorithm [HNR68, KK83, KP82] for finding the cheapest path through a graph. It is a best-first search algorithm that uses information about the graph to guide the search in the direction of the cheapest path. This is an improvement over Djikstra's Algorithm [Dij59] which is a uniform cost search algorithm.

Given a graph, A* finds a path from a given starting state to the goal state, having the smallest cost. Algorithm 1 shows the high level functioning of A*.

It evaluates at every iteration of the main loop which of the possible paths to follow next. The evaluation is performed by considering both, the cost of the path already covered (g) and the estimated (heuristic, h) cost of completing that path to the goal state. It steers the algorithm towards following the most promising path leading to the goal state. This is why, A* is also known as an informed search algorithm. A more detailed explanation of evaluation functions and their features is presented in Section 2.3.1

Most implementations of A* include a priority queue called `OPEN` which contains the set of states that have not yet been explored. The `OPEN` queue is sorted in increasing order of the heuristic cost of reaching the goal state. The `CLOSED` list is the set of states that have already been explored. Φ is the state in the graph from which we want to find the path to the goal state, γ. Let s be an arbitrary state in the graph. Then Algorithm 1 shows the functioning of the A* search.

Algorithm 1 A* Algorithm

```
1:  procedure ASTAR(Φ, γ, g, h)
2:      OPEN ← {Φ}
3:      CLOSED ← φ
4:      while |OPEN| > 0 do
5:          s ← ExtractMinimum(OPEN)
6:          if s = γ then
7:              return BacktrackPath(s, Φ)
8:          end if
9:          CLOSED ← CLOSED ∪ {s}
10:         for all v ∈ ChildState(s) do
11:             if v ∈ CLOSED then
12:                 continue
13:             end if
14:             f = g(s) + h(s, γ)
15:             Insert(OPEN, s, f)
16:         end for
17:     end while
18:     return φ                          ▷ No path found from Φ to γ
19: end procedure
```

2.3.1 Cost Evaluation Function

The cost evaluation function is used by the A* algorithm to guide the search towards the path with the least cost leading to the goal state. Let s be any arbitrary point on the graph that lies between a path from Φ to γ. Then, the cost evaluation function for state s is defined as:

$$f(s) = g(\Phi, s) + h(s, \gamma) \tag{2.1}$$

Where, $f(s)$ estimates the minimum cost of any path through state s for the given optimization objective. $g(\Phi, s)$ is the minimum cost of a path for the objective from

Φ to the state s. And finally, $h(s, \gamma)$, the heuristic cost function, estimates the minimum cost of a path from the state s to γ.

Since for any non-trivial graphs, the cost evaluation function must be computed for a large number of states, it is imperative that it is cheap to compute.

In most cases, the exact cost of the path from Φ to s is already known when evaluating the cost function. This is computed iteratively as each new state is explored and stored alongside as metadata. Hence, almost all the cost of evaluating the cost function comes from the evaluation of the heuristic cost function, h.

The job of the cost evaluation function is to guide the search towards the cheapest path to the goal state. Hence the closer it, and transitively the heuristic cost function, estimates the real cost of a path, the better the algorithm will perform. Given a perfect heuristic cost function, the cost evaluation function will lead A* to only explore states on the cheapest path, ignoring all the others.

Next, we discuss some of the features of the heuristic cost function and how they affect the performance of A*.

Admissibility

A cost function is said to be *admissible* if it never overestimates the true cost of reaching the goal state. Let $f^*(s)$ be the real cost of the optimization objective on the cheapest path from Φ to γ. Then an admissible cost function is one which always meets the following criteria:

$$f(s) \leq f^*(s) \tag{2.2}$$

With an admissible cost function, Pearl et al [KP82] show that A* is guaranteed to terminate with the cheapest path if one exists. A simple way to reason about this is as follows:
When A* terminates with a path, it has found a path with an actual cost that is lower than the estimated cost of all the other paths it is yet to explore. However, since each of these is an underestimation of the real cost, they can never lead to a path with a lower actual cost.

It would also be possible to run A* with a non-admissible heuristic. In such a case, it is possible that A* misses the optimal path due to an overestimation of the cost.

There exist several ϵ-admissible algorithms as well, which provide guarantees that the returned path has a cost no worse than $(1 + \epsilon)$ times the optimal path. Some such strategies include, Static Weighting [Poh70, Pea84], Dynamic Weighting [Poh73] and A^*_ϵ [GA83].

Monotonicity

A heuristic cost function is said to be monotone or consistent, if its cost estimate to a goal state is never greater than the estimated cost from a successor state, plus the cost to reach said state. Formally, it can be stated as follows:

$$\forall P \in successor(N),\ h(N, \gamma) \leq g(N, P) + h(P, \gamma) \tag{2.3}$$

This ensures that the value of the heuristic is always monotonic, and that it is never more optimistic than a previous guess. It is trivial to see that a consistent heuristic function is always admissible. However, the opposite is not true. An admissible heuristic is not always consistent.

A consistent heuristic makes the A* search more efficient. Without the guarantee of consistency, it is feasible that the same node in the graph is reached from different paths, each time with a different cost. In such a case, every time a state is reached again with a better cost than the previous times, it must be expanded again. This can severely degrade the performance of the search in the worst case where the state is reached multiple times, each time with an iteratively better cost. The algorithm presented in Algorithm 1 also assumes the use of a consistent heuristic.

Chapter 3

Real-Time System Model and Problem Statement

In this chapter, we introduce the system model that will be used throughout this thesis. We also describe our problem statement in terms of the system model and the criteria for optimality.

3.1 System Model

3.1.1 Machine Model

We consider a machine with K identical cores and a shared global cache of size M. The shared cache can be partitioned into S equal slices, such that $S \geq K$. At runtime, the set of cache slices may be (re-)partitioned arbitrarily among the cores and thus, among the tasks running on those cores.

Given a set-associative cache and way-based partitioning, the cache slices would correspond to the ways of the cache. Given a set-associative cache and set-based partitioning, which can be implemented in software, the cache slices would correspond to subsets of the cache's sets. The methods proposed here work independently of the way the cache is partitioned.

3.1.2 Task Model

We consider a system which must schedule a set of n real-time jobs, $J = \{j_1, j_2, \ldots, j_n\}$ on a machine as described in Section 3.1.1. Each job j_i is characterised by a 3-tuple (C_i, a_i, D_i), where C_i is the job's worst-case execution time, a_i is the arrival time of

job j_i, and D_i is the (absolute) deadline before which the job must be completed. As the execution time depends on the amount of allocated cache space, C_i is a function, $C_i = \{0, \ldots, S-1\} \rightarrow \mathbb{N}$, capturing the dependence of the job's execution time on the number of cache slices available to it. Evidently, the allocation of cache slices is done only once at the time of dispatch and cannot be changed later.

We also implicitly assume that C_i is a monotonically decreasing function, i.e. the WCET of a job only decreases with increasing cache slices. This is not necessarily true, since timing anomalies [RWT+06, SHK14, RS09] can cause the WCET to increase with larger cache sizes. However, such cases are easily dealt with by dropping them from C_i.

The model also makes an assumption that a job requires at least one slice in the shared cache in order to execute on the core.

Since our approach allows each job of a periodic task to be executed with different cache slices, we only deal with jobs in this thesis. Periodic tasksets are handled by unrolling them for the duration of their hyper-period in order to generate a jobset for scheduling.

Given a particular schedule, the lateness of a job j_i is defined as $L_i = f_i - D_i$, where f_i denotes the finishing time of job j_i under the given schedule.

3.1.3 Scheduling Model

We consider global non-preemptive scheduling, where each job is assigned a fixed number of cache slices throughout its execution. Global scheduling implies that jobs are free to be scheduled on any core without any restrictions. Under non-preemptive scheduling, once a job is scheduled on a core it runs to completion without any interruptions.

A job is said to *arrive* when it is available to the scheduler for being scheduled. This is the earliest time at which the job can begin execution. The scheduler may instead place the job in the *ready queue* where it waits for its turn to be scheduled. Jobs in the ready queue are called *ready jobs* and the scheduler is free to schedule them at any point. A job is *released* when the scheduler removes it from the ready queue and assigns it to a core for execution.

3.2 Problem Statement

Our goal is to find an optimal offline schedule that considers the available number of cores and cache slices. For an optimal schedule, we would like to minimise the maximum latenesses across all jobs.

For the purpose of finding such a schedule, we propose a new algorithm in Chapter 4 which takes into account the effect of the available cache on the execution time of a job and partitions the shared cache on a per-job basis.

3.2.1 Optimization Criteria

Optimality of a schedule can be defined in many different ways. The A* algorithm and its modifications presented in this work can be easily adapted for each of these definitions. For the purpose of this thesis, an optimal schedule is one that:

1. Is *feasible*, i.e. $\forall j_i \in J, f_i \leq D_i$, and
2. Minimises the maximum lateness across all jobs

Let Ψ be the set of all possible schedules for a given system model. Each of the schedules is represented by a sorted vector of the latenesses of each of the jobs in the jobset J. Then, the optimal schedule is one which has the smallest number as the last element of such a vector. Ties are broken by considering the next highest lateness. If all the latenesses are the same, the two schedules are considered equivalent and ties are broken arbitrarily.

3.3 Static Partitioning Not Optimal

Here, we show that static partitioning of the cache is not optimal in terms of feasibility of the given jobsets. We show this by presenting a counter-example where a jobset is infeasible under static per-core partitioning but would be feasible if the caches were dynamically repartitioned at runtime.

3.3.1 Example System

First we describe an instance of the system model for demonstrating that static per-core partitioning is not optimal. This system will also be used for demonstrating the algorithms and their impact throughout this document.

Let there be a machine with 2 identical cores and a shared global cache that is split into 4 equal slices. Table 3.1 describes a set of 3 jobs along with their execution times corresponding to the number of cache slices available during execution. This set of jobs must be scheduled on the machine described above such that every job meets its deadline.

TABLE 3.1: Jobset showing non-optimality of static cache partitioning

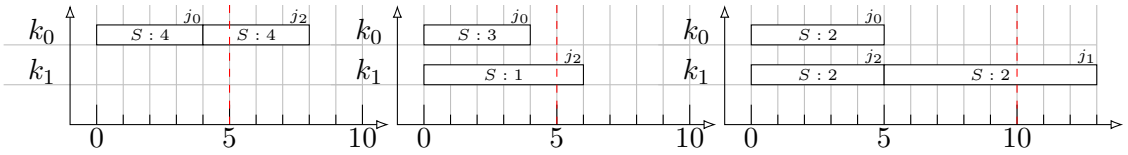
	D_i	C_i			
		s: 1	2	3	4
j_0	5	7	5	4	4
j_1	10	8	8	5	5
j_2	5	7	5	4	4

There are 3 different ways in which 4 slices of cache can be statically partitioned across 2 cores: (4,0), (3,1), (2,2). The schedules with each of these potential partitions are shown in Figure 3.1. In Figure 3.1a, core 0 has all 4 slices of cache, while core 1 has none. Thereby, effectively rendering it useless. This allows j_0 to finish in 4 units of time, ahead of its deadline. However, j_2 then misses its deadline at $t = 5$. If instead, a single slice of cache is provided to core 1, then j_0 still finishes at the same time, but j_2 can now be run in parallel on core 1. Nevertheless, the schedule is still infeasible as with just one slice of cache j_2 finishes at $t = 6$, as shown in Figure 3.1b. An equal partitioning of the cache will allow both j_0 and j_2 to meet their respective deadlines, as shown in Figure 3.1c. j_1 will still miss its deadline since it needs 8 units of time to execute with 2 slices of the cache, whereas its deadline is 5 units later.

As can be seen above, with static partitioning of the cache, the given jobset cannot be scheduled on this system. However, it is trivial to see that if we could provide j_1 additional cache in Figure 3.1c, the deadline would be met. This is shown in Figure 3.2, where the cache is repartitioned at $t = 5$. This allows j_1 to use the remaining unused cache and makes the jobset feasible.

Another way to see the example is through the vector of latenesses for each of the possible schedules:

$$
\begin{aligned}
3.1a: &< -1, \ 3, \ 3 > \\
3.1b: &< -1, \ -1, \ 1 > \\
3.1a: &< 3, \ 0, \ 0 > \\
3.2: &< 0, \ 0, \ 0 >
\end{aligned}
$$



(A) 4-0 Split (B) 3-1 Split (C) 2-2 Split

FIGURE 3.1: Scheduling Graphs with Static Partitioning

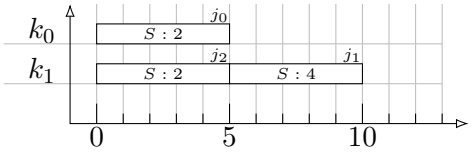


FIGURE 3.2: Dynamic Repartitioning of Cache

The vectors of latenesses show that the schedule shown in Figure 3.2 has the smallest maximum lateness. Being that none of the latenesses are higher than zero, this schedule is feasible; making it an optimal schedule for the given instance of a system. In this simple case, it is trivial to manually verify that no other schedule has a smaller maximal lateness.

Chapter 4

Adapting A* for Real-Time (AART)

In this chapter, we describe how the A* graph search algorithm can be adapted for finding the optimal schedule of a set of real-time jobs.

4.1 Scheduling Trees

A *scheduling decision* (Figure 4.1a) defines *when* a job will execute, *where* it will execute and with *what* resources it will execute. That is, a scheduling decision D is characterised by the 4-tuple, $D = (j_i, k, s, t)$. Where, j_i is the job being scheduled, $k \in [0, K)$ is the core on which the job will execute, $s \in [0, S)$ is the number of cache slices with which the job executes and t is the absolute time at which the job begins executing.

A *schedule* (Figure 4.1b) then can be seen as a sequence of scheduling decisions that account for each of the jobs in the jobset. For a given jobset there may exist many different schedules each corresponding to a different permutation of job ordering, core assignment and resource allocation. A large subset of the set of all possible schedules will share a common prefix and hence can be merged to form a prefix-tree or *trie* [FBN60, De 59]. We call such trees, *scheduling trees*.

The set of all possible schedules of a jobset possibly contains many schedules which do not meet the *feasibility* criteria laid out in Section 3.1.3. Hence, it can be seen that traversing any path that leads to an infeasible schedule is not useful when searching for the optimal schedule. The task of our algorithm then is to quickly traverse the

scheduling tree and find the optimal schedule while trying to avoid exploring as many infeasible schedules as possible.

4.2 State Space Exploration

We model the problem of finding the optimal schedule as one of searching for the shortest path within a scheduling tree. In such a tree, each node represents a scheduling decision and each edge represents the execution of the scheduling decision it points to. A set of edges and their corresponding nodes (beginning from the root node) represent a *partial schedule*. When a partial schedule contains nodes representing each job in the jobset, it is known as a *complete schedule*. A full scheduling tree will contain a large number of nodes and possible schedules, making an explicit exploration of the entire tree infeasible for all but the smallest jobsets.

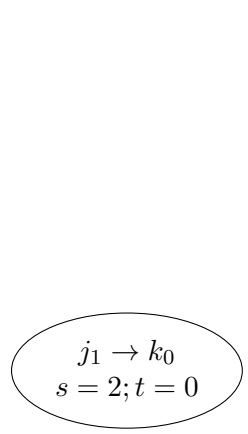
We propose to use the A* algorithm, introduced in Section 2.3, to efficiently explore this tree. As the term *state* is commonly used in the A* literature to refer to a node in the tree, we will also refer to nodes as states in the remainder of this document.

Let σ represent a partial schedule of the given jobset. Then, the A* algorithm as adapted for real-time scheduling is shown in Algorithm 2.

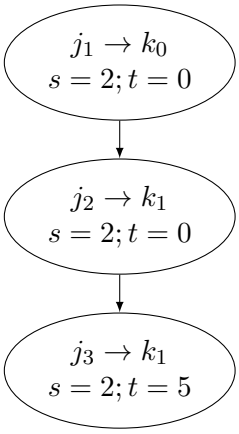
There are a few changes in this algorithm when compared to the standard A* algorithm presented in Algorithm 1:

1. `ExpandState` on Line 10 replaces `ChildState`. Unlike the standard A* algorithm, we do not feed the entire scheduling tree to `AStarRealTime`. Instead, `ExpandState` expands the given scheduling state by exhaustively matching all ready jobs to all available cores for each possible partition size from the available cache space. This is called *Dynamic State Expansion* and is described in more detail in Section 4.6

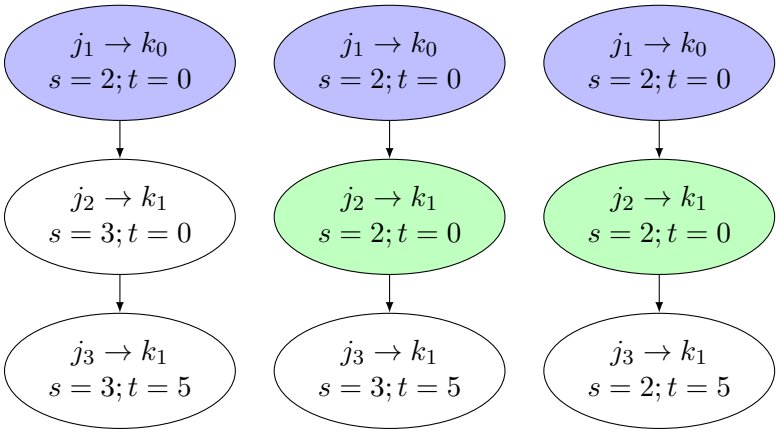
2. In hard real-time systems, we must also ensure that each job finishes before its deadline. There may exist partial schedules with a lower cost for the given optimization objective (lateness), but which cause one or more jobs to miss their deadline. Such partial schedules can never lead to a feasible schedule, which is one of our criteria for optimality. Thus, such partial schedules need not be explored once identified and can be safely pruned.



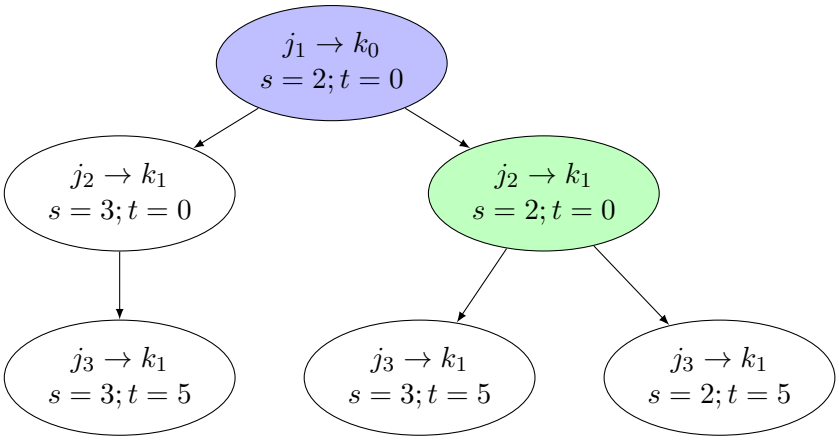
(A) Scheduling Decision



(B) Schedule



(C) Multiple Schedules



(D) Scheduling Tree

FIGURE 4.1: Scheduling Trees

Algorithm 2 Adapting A* to Real-Time Scheduling

```
1: procedure AStarRealTime(JobSet, h)
2:     OPEN ← {Φ}
3:     CLOSED ← φ
4:     while |OPEN| > 0 do
5:         σ ← ExtractMinimum(OPEN)
6:         if isCompleteSchedule(σ) then
7:             return BacktrackPath(σ, Φ)
8:         end if
9:         CLOSED ← CLOSED ∪ {σ}
10:        for all σ' ∈ ExpandState(σ) do
11:            f = Heuristic(σ')
12:            if isInfeasible(σ') then          ▷ If any job misses a deadline
13:                continue
14:            end if
15:            PruneExistingSchedules(OPEN, σ')
16:            if PruneNewSchedule(σ', {OPEN ∪ CLOSED}) then
17:                continue
18:            end if
19:            Insert(OPEN, σ', f)
20:        end for
21:    end while
22:    return φ                                  ▷ No feasible schedule possible
23: end procedure
```

3. In order to prevent a state-space explosion, we can also eagerly eliminate parts of the tree that are redundant or can never lead to the optimal schedule. We call this, *state space pruning* and it is discussed in more detail in Chapter 5.

4.3 Cost Evaluation Function

Let $\Psi(\sigma)$ be the set of all schedules that are completions of the partial schedule represented by σ. Then, the cost evaluation function is defined as:

$$f(\sigma) = g(\sigma) \bigoplus h(\sigma) \tag{4.1}$$

Where, $f(\sigma)$ estimates the minimum value of the optimization objective across all schedules in $\Psi(\sigma)$. Unlike the cost evaluation function in equation 2.1, the function

here is *not* a sum of two entities. Rather, under our definition it is any meaningful combination of the exact cost and heuristic costs for the given optimization objective.

4.3.1 Admissibility

Let, $f^*(\sigma)$ be the real minimum of the optimization objective for all schedules in $\Psi(\sigma)$:

$$f^*(\sigma) = \min\{g(\sigma') \mid \sigma' \in \Psi(\sigma)\} \tag{4.2}$$

Then, an *admissible* heuristic is one that never over-estimates the value of $f^*(\sigma)$:

$$f(\sigma) \leq f^*(\sigma) \tag{4.3}$$

A trivial heuristic function which meets the admissibility criteria is one that assumes that all the pending jobs complete at time $t = 0$. In this case, A* would degenerate into a breadth-first search, looking for the optimal scheduling under the given objective.

4.4 Running Example

Figure 4.2 shows the scheduling tree that is generated when the modified A* algorithm using the trivial heuristic function described at the end of Section 4.3.1 is used to schedule the set of jobs shown in Table 3.1 on a system as described in Section 3.3.1. A total of 58 different scheduling decisions were evaluated by the algorithm before identifying the set that leads to the optimal schedule.

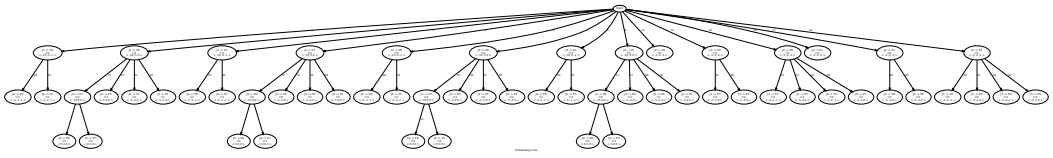


FIGURE 4.2: Scheduling Tree with Naïve Heuristic

As is evident from Figure 4.2, even a simple problem can potentially generate a large number of states. Hence, it is important that the algorithm uses efficient and effective heuristic functions (described in Section 4.5) and other approaches to smartly prune non-optimal paths from the search space (described in Chapter 5).

4.5 Heuristic Functions

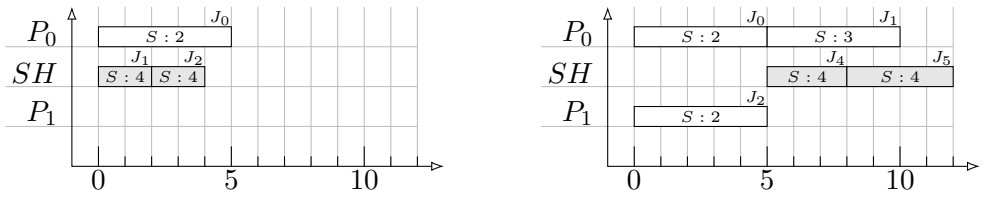
As mentioned in Section 4.3, the heuristic function is an important concept for the efficiency of this approach. In this section, we discuss different heuristic cost function implementations and how they affect the overall performance of the search.

4.5.1 Simulation-EDF Heuristic

The Simulation-EDF Heuristic works on the basis of the intuition that Earliest-Deadline-First (EDF) is an optimal scheduling strategy on uniprocessor systems [Hor74, SG92]. That is, it is guaranteed to minimise the vector of latenesses of all the jobs in the system [SG92].

Next, assume that there exists a machine with a single processor that has a cache of size M and is as fast as all the K cores combined, i.e. it executes jobs $K\times$ faster. Under EDF scheduling, such a machine is guaranteed to result in a vector of latenesses that is not greater than the optimal scheduling case on a K-core multi-core machine. This aligns perfectly with the admissibility criteria laid out in Section 4.3.1, that the heuristic function always underestimates the optimization goal.

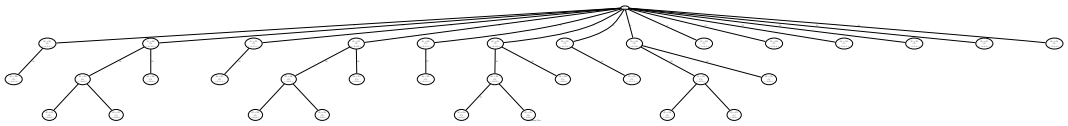
In order to compute the Simulation-EDF Heuristic, we simulate EDF based scheduling on a faster processor for all pending jobs. In order to ensure the admissibility of the computed heuristic it is mandatory that the simulation start from time $t = \min_K(t_K)$. Where t_K is the earliest time that any of the K cores is available for scheduling a job, starting from the instant of searching. Figure 4.3a shows an example of the computation of such a heuristic for the example problem in Section 3.3.1 after job j_1 has already been scheduled. A more general case with more jobs already scheduled is shown in Figure 4.3b. It shows that the computation of the Simulation-EDF Heuristic for j_4 begins at $t = t_K = 5$.



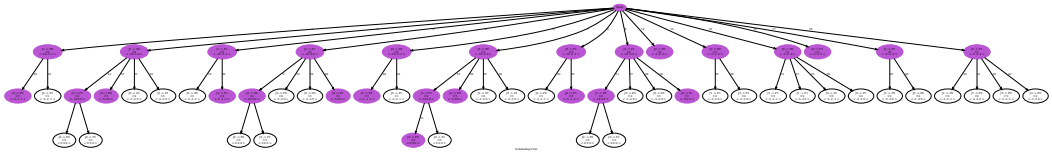
(A) Running Example (B) General Case

FIGURE 4.3: Simulation-EDF Heuristic

Figure 4.4a shows the scheduling tree that is generated during the process of finding the optimal schedule. A total of 34 different scheduling decisions are generated out of which the algorithm needed to consider and expand 28 decisions. Figure 4.4b shows the full scheduling tree generated by the trivial heuristic with the set of 28 decisions used by the Simulation-EDF Heuristic highlighted in purple.



(A) Simulation-EDF Heuristic



(B) Overlay on Full Scheduling Tree

FIGURE 4.4: Scheduling Tree with Simulation-EDF Heuristic

Proof of Admissibility

Let there be a machine with 2 identical cores, a and b, each running at a speed l. We define the schedule for each core as a mapping from a discrete point in time to the job it is executing at that point. Then, the schedule s_k for a core k is a function, $\mathbb{N} \rightarrow j_i \cup \perp$, where a mapping to $\perp$ implies that the core is idle.

Now, let us assume there is another machine with a single core, c, that runs at a speed of $2l$. Since it runs at double the speed, in each instant of time, it executes twice as many instructions compared to a and b. We can create a schedule for this core such that it leads to no worse lateness for each job compared to an arbitrary schedule available for the multi-core machine.

$s_c(2i) = s_a(i)$
$s_c(2i+1) = s_b(i)$

In such a schedule, every even instant of time executes the job from core a and every odd instant of time executes the job from core b. Hence, at every point during the execution, the schedule on core c is never executing things later than it would be executed on the two cores, a and b.

Here, we see that it is easily possible to create a single core schedule that is no worse than the schedules for both the slower cores. This method can be applied iteratively to an arbitrary number of cores, allowing us to make the following generalised statement:

Given a set of k cores and a corresponding schedule for each of them, it is possible to generate a schedule for a single core which runs at k times the speed, resulting in latenesses that are no worse than in each of the separate schedules.

Now, since we know that Preemptible-EDF (P-EDF) is an optimal scheduling strategy for single core machines, we can state that the latenesses of each of the jobs as scheduled by P-EDF will be no worse than the latenesses of our generated schedule. Combining the two, we can see that P-EDF will never result in a worse lateness than the optimal schedule on a multi-core system.

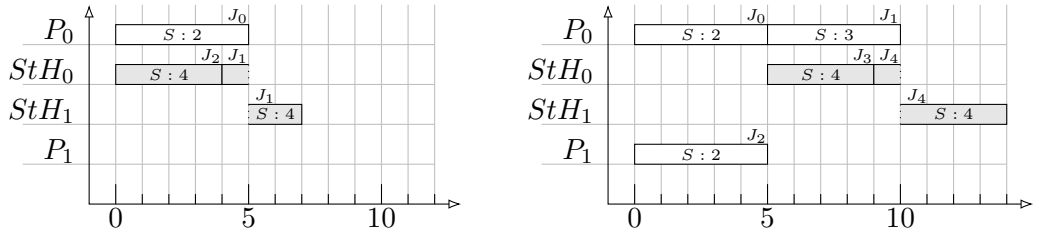
4.5.2 Stepped Heuristic

The Simulation-EDF heuristic is an improvement over the trivial heuristic of considering only the jobs that have already been scheduled. However, the Simulation-EDF heuristic is too optimistic about possible response times of each of the jobs. As mentioned in Section 4.5.1, the simulation of EDF begins at the earliest time that any core is scheduled to be free. This means, the heuristic does not account for the fact that there may exist cores which are scheduled to execute other jobs during this period and hence will be unavailable. This can be seen in action in Figure 4.3b, where between time $t = 5$ and $t = 10$ only the core P_1 was free and yet the computation assumes $2\times$ the core speed. Such optimism prevents the algorithm from pruning parts of the graph which will later lead to infeasible schedules.

The Stepped Heuristic is an improvement on the Simulation-EDF Heuristic. Instead of always assuming that we have available a uniprocessor machine running K times faster, we assume that the hypothetical machine has tunable speeds. When only one core is free, the hypothetical core also runs at a normal $1\times$ speed. When k different cores are free, the hypothetical core is capable of instantaneously ramping up its speed such that it executes jobs $k\times$ faster.

This method reduces the excess optimism that exists in the Simulation-EDF Heuristic and provides for a more accurate guess of the latenesses of pending jobs. Figure 4.5a shows the computation of this heuristic for the running example introduced in Section 3.3.1 for the case when only job J_0 has been scheduled. According to EDF,

the job J_2 is scheduled first and it executes for 4 time units on the hypothetical processor StH_0 which runs at the same speed as core P_0. Next, the job J_1 is scheduled which requires 5 time units. It executes for the first unit on the hypothetical core, at which time, both cores P_0 and P_1 would become free. Hence, the core doubles it clock speed instantaneously. This faster core is shown as StH_1 and finishes executing the 4 units of execution pending for J_1 in only 2 time units.



(A) Running Example

(B) General Case

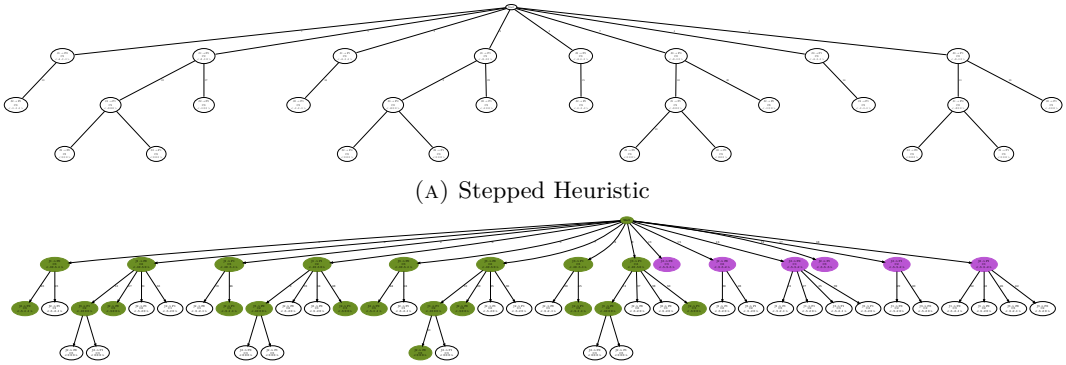
FIGURE 4.5: Stepped Heuristic

As can be seen in Figure 4.6a, with this method, only 28 different scheduling decisions are generated and 21 of them need to be evaluated before the optimal schedule is identified. Figure 4.6b shows the 21 different decisions that had to be evaluated highlighted in green. The nodes highlighted in purple indicate the scheduling decisions that were evaluated for the Simulation-EDF Heuristic, but not the Stepped Heuristic.

Figure 4.5b shows this heuristic being applied in a more general case. For a system with K cores, the Stepped Heuristic modulates its core frequency from $1\times$ to $K\times$ in K steps, lending the name of this algorithm. Such a computation also leads to cases where under EDF, a job is scheduled to run on a hypothetical core, but mid-way through, the core must ramp up its frequency. This case is handled naturally and the completion time of the job is scaled proportionally to the amount of time it executed on each clock frequency.

4.6 Dynamic Graph Expansion

The standard A* graph search algorithm expects the entire graph along with all the edge weights to be provided to the algorithm. However, in the case of a scheduling tree, the resulting graph is too large to completely expand and store in memory. Each potential *scheduling decision* incurs a memory overhead that is directly proportional to the number of jobs in the jobset. And the number of potential scheduling decisions



(A) Stepped Heuristic

(B) Overlay on Full Scheduling Tree

FIGURE 4.6: Scheduling Tree with Stepped Heuristic

increases exponentially with the number of jobs in the jobset. Hence, the proposed algorithm expands the scheduling tree for only those partial schedules that it intends to explore.

In this section we discuss a simple, naïve approach to expanding the graph and another that optimises the method for better performance.

4.6.1 Fully Exhaustive Matcher

The *Fully Exhaustive Matcher* (FEM), is a simple and naïve approach to expanding the scheduling graph. Given a scheduling state σ, the FEM will generate a new scheduling state by exhaustively matching every pending job with every CPU core for all possible cache slice allocations. A high-level overview can be seen in Algorithm 3. As can be seen in Algorithm 3, the only case where a scheduling action is not considered is when the number of cache slices(s) used does not improve the WCET(C_i) of the job compared to using fewer slices

This algorithm seems reasonable at first sight. It is the obvious way to expand the scheduling tree on demand. However, this technique has two major drawbacks which affects the performance of the overall A* algorithm.

1. Generation of (potentially) bad schedules

   When the arrival time of the job being considered is *after* all cores finish their scheduled executions, one of the cores must idle waiting for the job to arrive. However, if the arrival time is significantly later in the future, it is feasible that the core will be forced into an idle state longer than the maximum execution time of a ready job.

Algorithm 3 Fully Exhaustive Matcher

```
1: procedure FEMExpansion
2:     NextSchedules ← list()
3:     for all j_i ∈ PendingJobs do
4:         for all k_j ∈ K do
5:             EarliestReleaseTime ← max(a_i, cores[k_j])
6:             for all s ≤ S do
7:                 if C_i(s) ≥ C_i(s − 1) then
8:                     continue
9:                 end if
10:                ReleaseTime ← SlicesAvailableAt(s, EarliestReleaseTime)
11:                SchedAction ← CreateSchedulingAction(j_i, k_j, s, ReleaseTime)
12:                NextSchedules.add(SchedAction)
13:            end for
14:        end for
15:    end for
16:    return NextSchedules
17: end procedure
```

Even if no deadlines are missed, such idling causes a loss of optimality since the ready job could have been executed during that idle period. Hence, generating such schedules only wastes time and memory, and should be avoided when possible.

2. Generation of redundant schedules

A *fully exhaustive matching* also causes generation of redundant schedules. For example, the partial schedule depicted in Figure 4.7 can be reached from 2 different paths after j_0 has been scheduled. Either the Job j_1 is scheduled first on processor P_0 and then Job j_2 on P_1, or vice versa. However, both these paths will lead to precisely the same complete schedule in the future. Such redundant partial schedules are also captured and eliminated by the state-space pruning approaches described in Chapter 5. However, those approaches incur a runtime cost and memory overhead and hence it is useful to simply not generate such redundant schedules when possible.

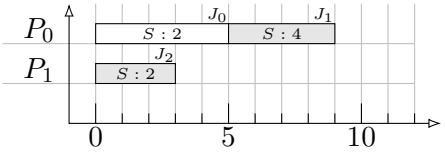


Figure 4.7: Sample Schedule

4.6.2 Chronological Match Generator

The drawbacks mentioned in the previous section add up to a significant performance hit for most jobsets. In order to alleviate these issues, we implemented another graph extension algorithm called, the *Chronological Match Generator* (CMG). A high level overview of the CMG can be seen in Algorithm 4, with the changes highlighted.

Algorithm 4 Chronological Match Generator

```
1:  procedure SkipSchedulingJob(j)
2:      for all j_p ∈ PendingJobs do
3:          if D_p < a_j then
4:              return True
5:          end if
6:      end for
7:  end procedure

8:  procedure CMGExpansion(CurrentSchedule)
9:      NextSchedules ← list()
10:     for all j_i ∈ PendingJobs do
11:         if SkipSchedulingJob(j_i) then
12:             continue
13:         end if
14:         for all k_j ∈ K do
15:             EarliestReleaseTime ← max(a_i, cores[k_j])
16:             for all s ≤ S do
17:                 if C_i(s) ≥ C_i(s − 1) then
18:                     continue
19:                 end if
20:                 ReleaseTime ← SlicesAvailableAt(s, EarliestReleaseTime)
21:                 if ReleaseTime < CurrentSchedule.LastReleaseTime then
22:                     continue
23:                 end if
24:                 SchedAction ← CreateSchedulingAction(j_i, k_j, s, ReleaseTime)
25:                 NextSchedules.add(SchedAction)
26:             end for
27:         end for
28:     end for
29:     return NextSchedules
30: end procedure
```

Under the Chronological Match Generator, we attempt to skip the generation of partial schedules which are guaranteed to not lead to a unique complete schedule.

The call to `SkipSchedulingJob` on Line 11 handles the first drawback of the Fully Exhaustive Matcher. It prevents generating a scheduling decision where a job would idle-block a core for an amount of time that is long enough to completely schedule a ready job. Similarly, the check on Line 21 deals with the second drawback mentioned for the FEM. It will forces generation of scheduling decisions in a chronological order. That is the release times of the generated scheduling decisions are always monotonically increasing. This prevents generating redundant schedules from different paths.

It is important to note here that the schedules pruned in these checks would otherwise have been deemed infeasible or marked redundant by the state-space pruning approaches explained in Chapter 5. However, as mentioned earlier, each generated scheduling decision incurs a memory overhead proportional to the size of the jobset. Hence, avoiding the generation of these decisions helps to reduce the peak memory usage of the algorithm.

Chapter 5

State Space Pruning

The set of all possible schedules for a given job set is extremely large. Even navigating it with a good heuristic function can require large amounts of memory and time. In order to improve the overall performance of the A* algorithm, we propose another modification to it. These modifications are called *State Space Pruning* and work to reduce the size of the search space that the algorithm must navigate. In this section we talk about the different methods for state space pruning.

5.1 Isomorphic Schedules

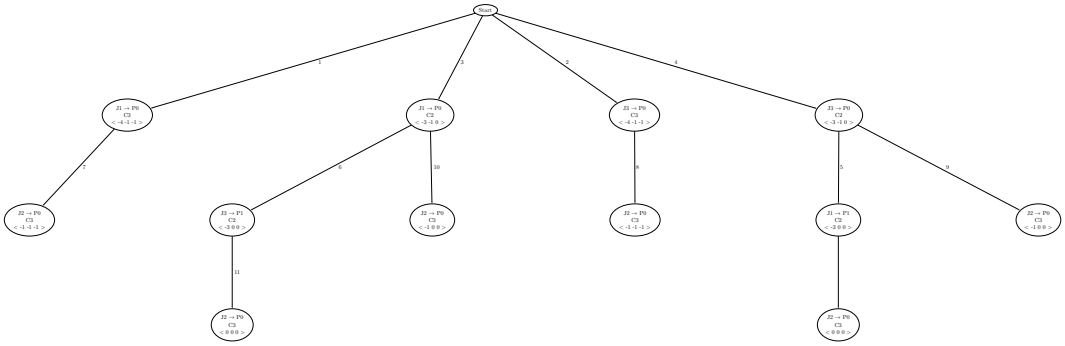
Multiple schedules may be equivalent to one another. Such schedules will always lead to the exact same result in terms of the optimization objective. For example, consider two states, σ_1 and σ_2, which represent the following partial schedules:

σ_1	σ_2
$j_0 \to k_0$	$j_0 \to k_1$
$j_1 \to k_1$	$j_1 \to k_0$

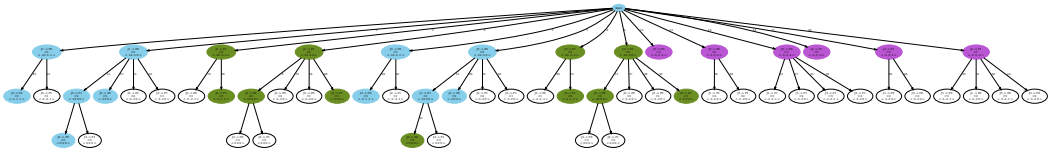
In this case, it is trivial to see that for every complete schedule in $\Psi(\sigma_1)$, there exists a complete schedule in $\Psi(\sigma_2)$ with the exact same value of the optimization objective.

We call such partial schedules, *isomorphic schedules*. The algorithm needs to follow only one schedule from a set of isomorphic schedules in order to ensure that the entire state space is searched.

The pruning of isomorphic schedules happens most frequently at the very early stages of the algorithm, and is most useful during the first iteration when it eliminates a significant chunk of the potential scheduling trees. Figure 5.1a shows the scheduling



(A) Pruning of Isomorphic Schedules



(B) Overlay on Full Scheduling Tree

FIGURE 5.1: Scheduling Tree with Pruning of Isomorphic Schedules

tree generated after applying pruning of isomorphic schedules while using the Stepped Heuristic. Only 12 potential scheduling decisions were generated out of which 11 had to be evaluated before the optimal schedule was identified. The 11 decisions that were evaluated by the algorithm are highlighted in light-blue on the cumulative scheduling tree shown in Figure 5.1b. As before, the purple and green nodes indicate the scheduling decisions that were evaluated when applying only the Simulation-EDF and Stepped Heuristics respectively, but not when also applying pruning of isomorphic schedules. This shows us how effective the pruning of isomorphic schedules is and how it can be used to significantly reduce the size of the search space.

5.2 Dominated Schedules

During the exploration of the entire state space, one may come across partial schedules which are objectively worse than another partial schedule. That is, there may exist partial schedules whose completions will never result in a lower value of the optimization objective than another schedule in the tree. Such partial schedules are called dominated schedules and it is trivial to see that they can never lead to the optimal path. Hence pruning them early can potentially prevent the algorithm from exploring parts of the graph which will not result in the optimal path.

Let $\sqsubseteq$ be an operator that defines the domination relation and σ_1 and σ_2 be two partial schedules. Then:

$$\begin{aligned} \sigma_2 \sqsubseteq \sigma_1 &\implies \min\{g(\sigma_1') \mid \sigma_1' \in \Psi(\sigma_1)\} \leq \min\{g(\sigma_2') \mid \sigma_2' \in \Psi(\sigma_2)\} \\ &\implies f^*(\sigma_1) \leq f^*(\sigma_2) \end{aligned} \tag{5.1}$$

In other words, if state σ_2 is dominated by σ_1 then for every possible completion of σ_2, there exists a completion of σ_1 with the same or lower value of the optimization objective. From this, we can see that the best complete schedule that is an extension of σ_2 will never be better than the best complete schedule extending from σ_1. By pruning all completions of the dominated state, σ_2, we can reduce the search space of the algorithm, allowing it to be faster and more memory efficient. The earlier a dominated schedule is identified, the larger the portion of the overall search space that can be pruned.

As can be seen in Algorithm 2, on line numbers 15 and 16, the pruning of dominated schedules is done in two steps:

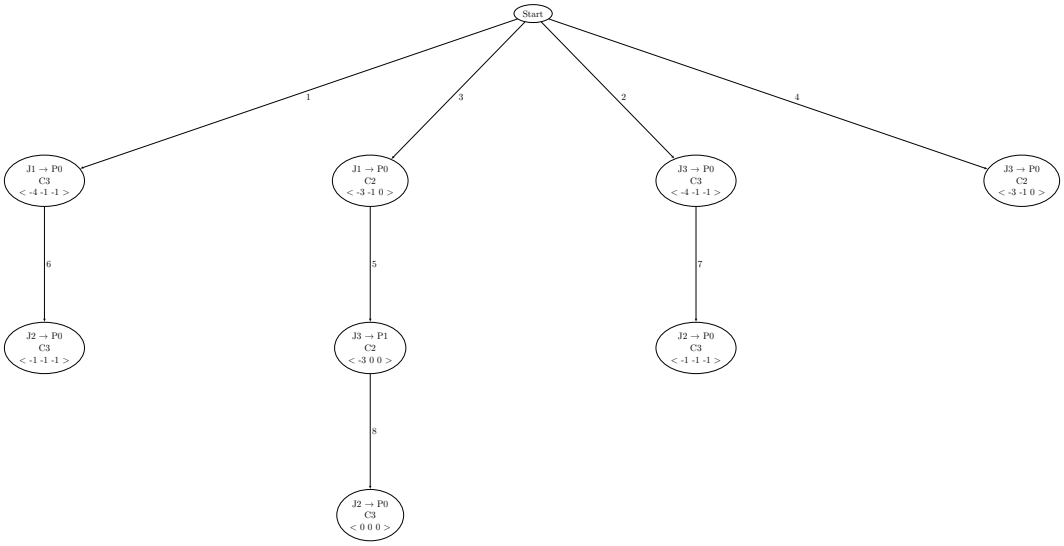
1. Remove any partial schedules that the algorithm has not yet explored which are dominated by the current schedule
2. Stop processing the current schedule if it is dominated by any schedule in $OPEN \cup CLOSED$

We can then define the domination relation $\sigma_2 \sqsubseteq \sigma_1$ as:

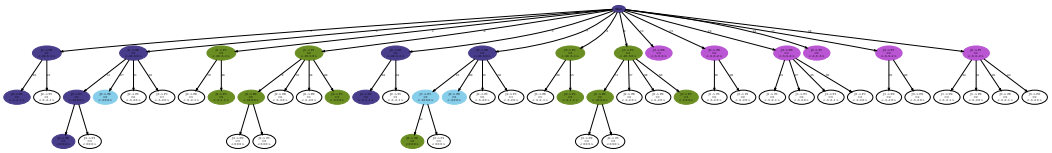
- σ_1 has finished scheduling a superset of the jobs scheduled in σ_2
- For each of the jobs scheduled in σ_2, the finishing time of the same job under σ_1 is the same or earlier
- σ_1 has required the same or fewer cache slices to schedule the jobs until now

Considering each of the above points together, we can see that for every schedule in $\Psi(\sigma_2)$, there will always exist a schedule in $\Psi(\sigma_1)$ where the jobs finish with a smaller lateness.

Below we describe two mechanisms for identifying and pruning dominated states.



(A) Pruning of Dominated Schedules



(B) Overlay on Full Scheduling Tree

FIGURE 5.2: Scheduling Tree with Pruning of Dominated Schedules

5.2.1 List-Based Domination Checker

After filtering the isomorphic schedules as seen in Section 5.1, all the remaining schedules are tested for *domination*. Our implementation maintains two doubly linked-lists, `Domination::OPEN` and `Domination::CLOSED`. Each of these lists contain elements composed of a `DominationState`. A `DominationState` consists only of information required for the domination checks. This includes, 1. The latenesses of every job scheduled in that partial schedule, 2. The times during which each of the CPU cores were in use, 3. The times when each of the cache slices were allocated to a job.

Each *scheduling state* that is removed from the top of the `OPEN` queue for dynamic graph expansion, is also removed from the `Domination::OPEN` list. This requires a linear search through the list. After, expansion, a `DominationState` is created for each of the states, σ_g, that are both feasible and not isomorphic. This `DominationState`

is compared, linearly, with each of the states in `Domination::OPEN` to see if it dominates any partial schedule that has not yet been explored. Any states found to be dominated are removed from both, the `Domination::OPEN` list and the `OPEN` queue.

Next, the `DominationState` is checked against all the elements in both the lists, `Domination::OPEN` and `Domination::CLOSED` to see if any of those elements dominate it. If they do, the state, σ_g is dropped since we have found a partial schedule that will always perform better than it. This check also entails a linear search across both the lists.

Since every new partial schedule is checked against every schedule that has been generated so far, the complexity of this search is $O(n^2)$ in the total number of partial schedules generated.

Partial schedules that are identified as dominated can prune away large parts of the search-space. This can be seen in Figure 5.2a, which shows that only 7 scheduling decisions had to be considered before finding the optimal schedule. The 7 decisions are also highlighted in dark-blue in the cumulative scheduling tree in Figure 5.2b.

5.2.2 Depth Domination Checker

Profiling of the implementation runtime shows that about 95% of the running time is spent in performing the domination check described above. This is a very large percentage of the total runtime, however, we expect it to be more efficient on the whole since it prevents exploration of large parts of the state space. Any optimisations made to the List-Based Domination Checker have a potential to be very effective since it dominates the overall runtime cost.

Let σ_1 and σ_2 be two scheduling states such that $\sigma_2 \sqsubseteq \sigma_1$. Then, under the List-Based Domination Checked described above, it is possible that the set of scheduled jobs in σ_1 is a strict superset of the set in σ_2. That is, it has not only scheduled all the jobs in σ_2 with a lower finishing time, but also scheduled some additional jobs on top of it. We call the number of scheduled jobs in a partial schedule, the *depth* of the schedule since it corresponds to the depth of the tree.

An analysis of the runtime logs shows this case indeed occurs and it is not just theoretical. However, we also noticed that it does not occur often enough. In an overwhelming majority of the domination relations identified, the depth of both the partial schedules is the same, i.e. $ScheduledJobs(\sigma_1) = ScheduledJobs(\sigma_2)$.

The cost of finding such domination relations seemed to be greater than the time saved by the pruned states resulting from them. Thus, we implemented the Depth Domination Checker. In this checker, separate `Domination::OPEN` and `Domination::CLOSED` lists are maintained for each *depth* of the tree. The depth here indicates the number of scheduled jobs. The checks are then performed exactly as described for the List-Based Domination Checker, with the caveat that only states with equal depths are compared.

Since, under this method, every partial schedule against every partial schedule generated so far that has the same number of scheduling decisions, the complexity of this method is, $O(n^2)$ in the total number of partial schedules generated at every *depth*. Hence, this algorithm is never worse than the *List-Based Domination Checker*. It may however result in missing some branches that could potentially be pruned. The effects of this are shown and discussed in more detail in Section 7.1.3.

For our simple running example, the change to Depth Domination Checker, does not change the number of states explored.

Chapter 6

Non-Optimal Scheduling

Throughout this thesis, our focus has been on searching for the optimal schedule. Finding the optimal schedule is an intractable problem that requires exploring most of the state space to solve. This is important when the target system is heavily resource constrained and efficient use of the available hardware is a necessity. In such cases, one must bear the high computational cost of a full state-space search in order to find the optimal schedule.

However, the target system is not always extremely resource constrained. In such cases, one may be more interested in quickly finding a feasible schedule rather than the optimal one. Feasible schedules cause sub-optimal use of the hardware resources, but incur a lower computational cost compared to searching for the best possible schedule.

We would also like to know how a certain feasible schedule compares to the optimal schedule. It may also be useful to generate not just a feasible schedule, but one which is within a certain bounds of optimality. In this chapter, we present one technique that allows us to identify a feasible schedule with a bounded guarantee on how efficiently it uses the available hardware.

Section 4.5 first introduced the idea of assuming a faster single core for computing the heuristic cost of a partial schedule. The faster core was used to simulate a single core that performs an equivalent number of computations as a corresponding multi-core system. A scaling factor (s.f.), f, represents the factor by which the hypothetical core is sped-up.

Similarly, we can assume a core that is *slower* than the actual system model by setting $0 < f < 1$. We can use such a hypothetical core to simulate EDF during the

heuristic computations. That is, for the purpose of computing the heuristic latenesses of each of the pending jobs, we assume a core that is slower by a factor of f. Hence, we estimate a longer execution time than required for each of the pending jobs. This results in dropping partial schedules where the finishing time of a job is very close to its deadline.

Since the A* algorithm will pick the next scheduling decision based on the value of the heuristic lateness and actual latenesses of scheduled jobs, the final schedule generated by this method is guaranteed to perform no worse than the optimal schedule on a machine that is slower by a factor of f.

Consider the example introduced in Section 3.3.1 and the Simulated-EDF Heuristic as the heuristic cost function. Next, let us assume a scaling factor of $f = 0.5$. This implies that during the evaluation of the heuristic, we assume that the core runs at only half its speed. Hence for our example, the effective scaling factor, after considering the speedup due to number of cores is $2 \times \frac{1}{2} = 1$. We can see this in Figure 6.1, where the heuristic costs of both J_2 and J_1 are doubled as compared to the Simulated-EDF Heuristic (Figure 4.3a).

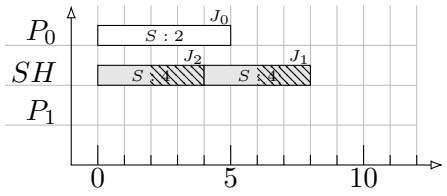


FIGURE 6.1: Effect of Scaling Factor, $f = 0.5$

Chapter 7

Experimental Evaluation

In this chapter, we will discuss the performance of the algorithm presented in Chapter 4. We will discuss the various factors that affect the runtime performance and also how it compares to standard static per-core partitioning. Section 7.4 also shows how the non-optimal scheduling strategy presented in Chapter 6 affects the performance.

Experimental Setup

The algorithm presented in Chapter 4, titled `AART`, was implemented in C++14. The implementation is single-threaded and is designed to be memory efficient, sometimes at the cost of performance. `AART` takes a JSON file containing the jobset as input and prints out an optimal schedule if it can find one. The input for the running example used throughout the thesis is shown in Listing 7.1.

For the evaluations presented here, the jobsets were generated by first generating a taskset and then unrolling them for the duration of their respective hyperperiods. The tasks of the tasksets were selected at random (without replacement) from the set of all tasks available in TACLeBench [FAH+16].

`AART` requires the partition size sensitivity of tasks as part of its input. The partition size sensitivity of a task represents the WCET of the task when it is executed in isolation on a single core with different amounts of cache. For the analysis of the WCETs, the low-level timing analysis tool `LLVMTA` [HJR16] was used. Each of the tasks was provided with a 4 KiB instruction cache and a data cache ranging from 128 bytes to 1 KiB in steps of 128 bytes.

```
{
  "J1": {
    "wcet": {"16": 7, "32": 5, "48": 4, "64": 4},
    "id": 1,
        "arrival": 0,
    "deadline": 5
  },
  "J2": {
    "wcet": {"16": 8, "32": 8, "48": 5, "64": 5},
    "id": 2,
        "arrival": 0,
    "deadline": 10
  },
  "J3": {
    "wcet": {"16": 6, "32": 5, "48": 4, "64": 4},
    "id": 3,
        "arrival": 0,
    "deadline": 5
  }
}
```

Listing 7.1: Running Example

A *WCET Profile* is then generated for each of the analysed tasks. The WCET Profile shows the partition size sensitivities of a task relative to the WCET at a fixed reference point (1 KiB data cache). An example of a WCET Profile with 4 slices is shown in Table 7.1 for the task `minver`. It shows the real WCET in the first column and the WCET relative to the case with the maximum cache in the second column. This information tells us how the size of available cache influences the WCET of a task.

The total utilization of tasksets is varied in the range [0.05, 2.00] with a step of 0.05.

Table 7.1: Example of WCET Profile for `minver`

Slices	Real WCET	WCET Profile
1	212831	1.24813
2	174653	1.02424
3	171309	1.00463
4	170519	1.00000

The number of tasks per taskset is varied from 2 to 10 tasks. For each combination of the total utilization and number of tasks, 100 tasksets are randomly generated, yielding a total of 36,000 tasksets.

Each of the tasks in a taskset is assigned a WCET Profile at random. The utilization of each of the tasks is computed using the `randFixedSum` algorithm [ESD10]. Periods are chosen uniformly at random from a fixed set of values ([1, 2, 5, 10, 20, 50, 100, 200, 1000] ms) that are indicative of periods in real workloads [vdBUCF17, KZH15]. The reference WCET of the task is computed as the product of its period and utilization. The WCETs at different cache slice availabilities are then computed using the WCET Profile.

The periods are chosen from a fixed set of values which are mostly harmonic. This allows us to bound the maximum length of the hyperperiod. Similarly, we ensure that all tasksets contain no more than 100 jobs per hyperperiod. This is done in order to avoid comparing jobsets that contain 10 jobs with jobsets containing 1000 jobs.

The evaluations are run by assuming a system with $K = 2$ cores. Each core has a private, non-shared, 4 KiB instruction cache. The system also contains a 1 KiB data cache that is shared among all the cores. Only the data cache is partitioned, and it can be partitioned into 8 slices of 128 bytes each. However, for the purpose of these evaluations, we consider the system to contain only 4 slices of 256 bytes each.

The cache sizes were so chosen to ensure that over 80% of the tasks the instructions fit entirely into the instruction cache and that at least 50% of the tasks do not fit within the provided data cache [Hah19].

All the evaluations presented here were executed on a machine with an Intel Xeon E3-1220 CPU, running at 3.10 Ghz, with 32 GiB of DDR3 RAM. Each execution of `AART` was bound to a single core to prevent task migrations. The evaluations were also bounded to a maximum execution time of 1 second, after which the jobset was classified as *feasible*, *infeasible* or *timeout*.

7.1 Effectiveness of Different Strategies

In this section, we will discuss the effectiveness of the different strategies for evaluating the cost function, dynamically expanding the graph and for pruning the state space through dominations.

For this evaluation, `AART` was used to schedule each of the 36,000 generated jobsets. In each of the graphs presented next, the X-Axis represents the total utilization of the jobset, while the Y-Axis represents the number of jobsets which timed out. Hence, lower curves indicate a better performance of the algorithm.

Only the curves for tasksets with 3, 5, and 8 tasks are shown here. The full evaluation results for all taskset sizes from 2 to 10 are available in Appendix A.

7.1.1 Dynamic Graph Expansion

In Section 4.6 we discussed the need for dynamically expanding the graph during the execution of the algorithm. Two different methods were introduced for such dynamic expansion: 1. Fully Exhaustive Matcher (FEM), and 2. Chronological Matcher (CM).

The Chronological Matcher improved upon the Fully Exhaustive Matcher by reducing the number of non-optimal and redundant schedules that are generated. As a result, we expect that with the Chronological Matcher, more branches of the tree can be explored in the same amount of time. This should lead to fewer timeouts as the entire graph can be searched in a shorter span of time. However, as the problems get more difficult, and the graphs larger, the effects of the Chronological Matcher are diminished. These effects can all be seen in Figure 7.1. In the figure we can see that the improvements are more stark in the $n = 3$ case compared to any others. Since the Chronological Matcher never performs worse than the Fully Exhaustive Matcher and has no drawbacks, we recommend always using it.

One interesting thing to note in Figure 7.1 is that the number of timeouts decrease with increasing utilization. That is, as the scheduling problem gets harder to solve, our technique keeps showing better performance. This is explained in Section 7.2.2 where we discuss the effect of utilization on the overall performance.

7.1.2 Cost Evaluation Functions

Section 2.3.1 introduced the idea of a cost evaluation function in the A* algorithm and some desired features and properties of such functions. Later, in Section 4.5 we present two different functions for computing the heuristic cost of a partial schedule. Both of these functions are based on the idea that uniprocessor P-EDF is an optimal scheduling algorithm. We claimed that using the Stepped Heuristic as a heuristic function is better since it is less optimistic about the execution times of the jobs.

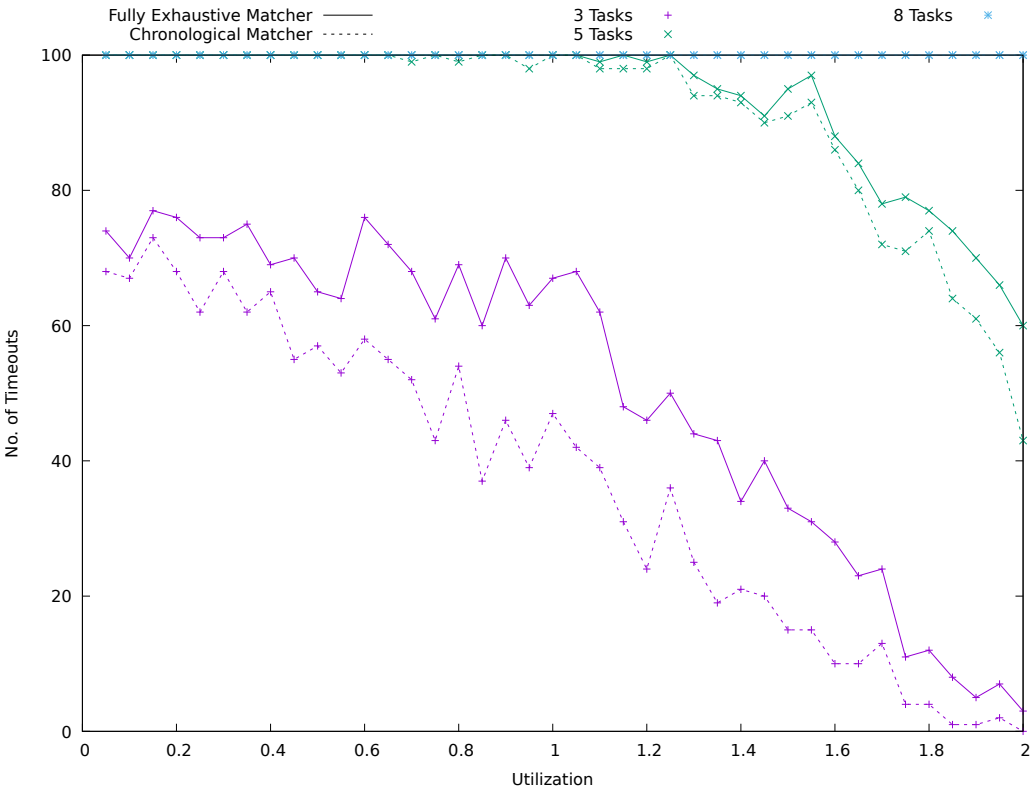


FIGURE 7.1: Performance of Dynamic Graph Expansion Strategies

Accounting for the period of time when a core is definitely busy and cannot be used for scheduling a new job should yield more realistic guesses for the finishing times of the jobs. Figure 7.2 shows a comparison between the two heuristic functions proposed in this thesis.

As is expected, using the Stepped Heuristic lends a significant boost to the overall performance. This effect only grows larger as the scheduling problem becomes more difficult. At lower utilization values, the state space of the graph remains too large to process within the timeout of 1 second.

The Stepped Heuristic is indeed a more expensive function to evaluate. However, this expense is offset by the time it saves through guiding the search more accurately towards the optimal schedule. Hence we recommend always using the Stepped Heuristic due to the increased performance it provides across the board.

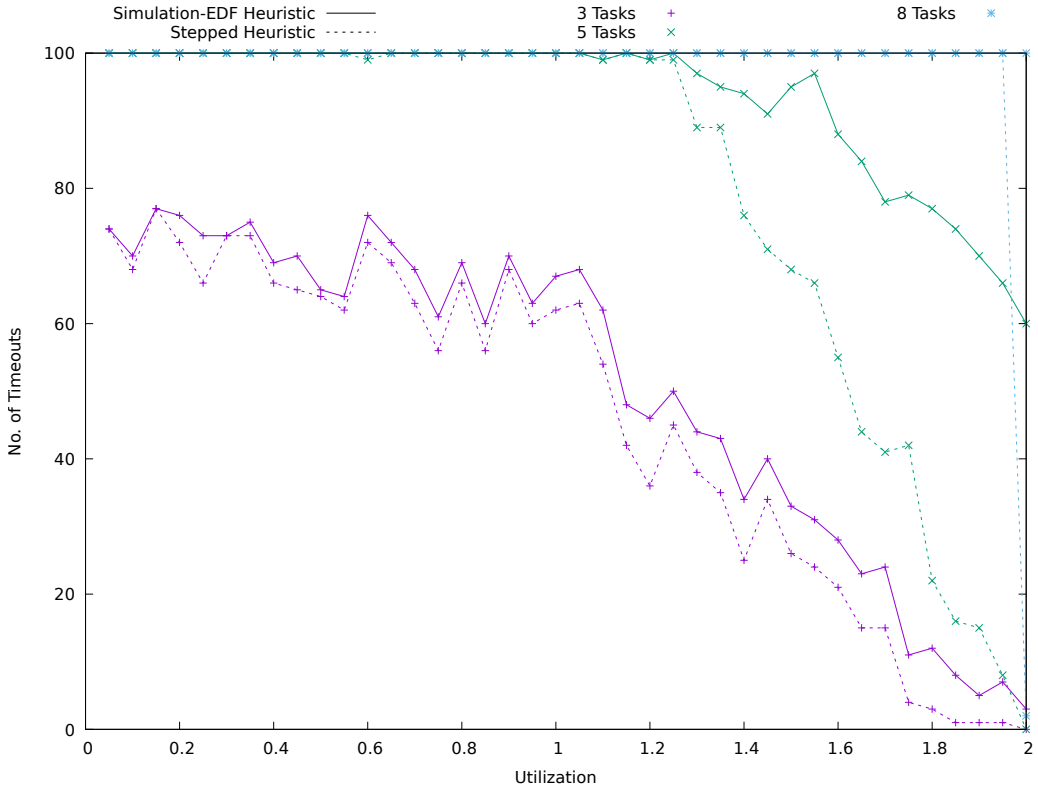


FIGURE 7.2: Comparison of Heuristic Functions

7.1.3 Domination Searching

Isomorphic schedules, discussed in Section 5.1, are very useful in reducing the search space, but they are most effective only for the first scheduling decision. This is also noted by Kwok and Ahmed [KA05] in their work which introduced the concept. The reason is plainly that the rigid requirements of isomorphic schedules are usually never met again during the later stages.

On the other hand, dominated schedules can be detected throughout the process of finding the optimal schedule. Checking for dominated states however is an expensive operation; in our analysis, we found that $\sim 95\%$ of the execution time is spent in looking for dominated states. Yet, we conjecture that the cost of searching for dominated states is offset by the time saved in not exploring those branches of the state space.

Figure 7.3 shows a comparison between the two different strategies proposed in Section 5.2 and the case when not pruning dominated states.

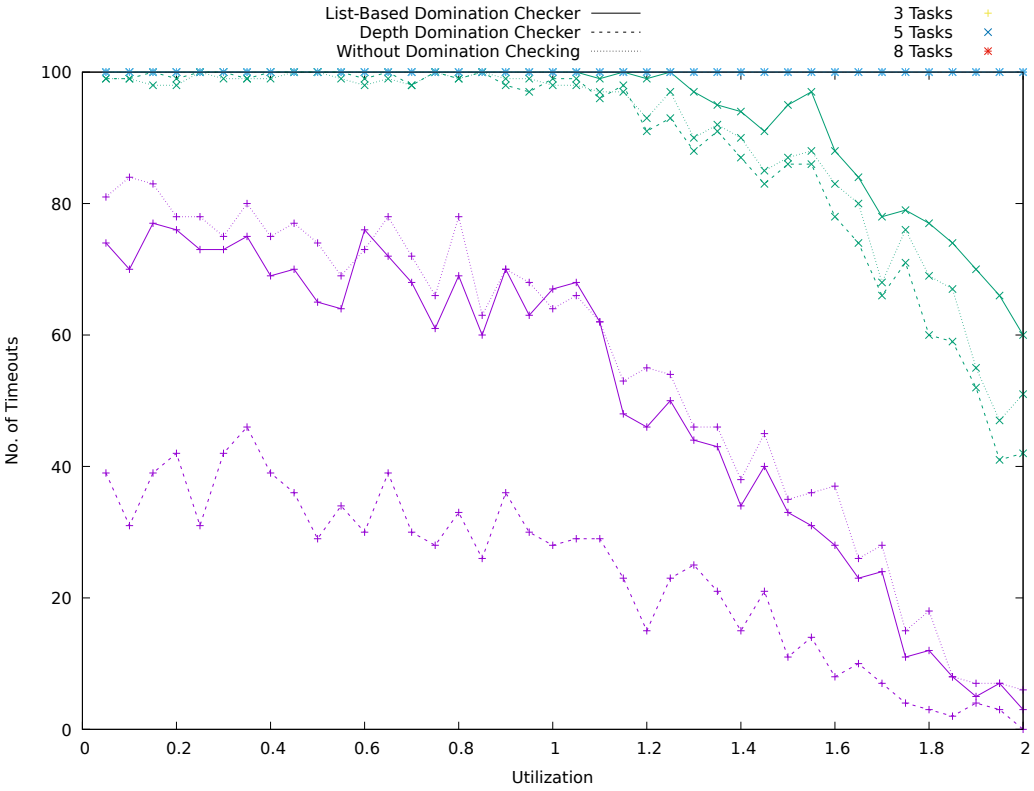


FIGURE 7.3: Comparison of Domination Checking Strategies

In Figure 7.3 we see the *List-Based Domination Checker* is not very effective. With a taskset with just 3 jobs, it performs about the same as if we performed no pruning of dominated states at all. The *Depth Domination Checker* performs better up to 5 tasks in the taskset. However, as can be seen in Appendix A, is also eventually yields worse performance than not pruning dominated states at all.

This occurs due to the fact that our implementations of domination checkers have a $O(n^2)$ complexity. Thus, as the problem size grows larger, the cost of checking for domination often grows faster than the costs of exploring the pruned branches of the tree.

The large improvement when $n = 3$ shows there is there is a lot of potential in improving the performance of the state-space search with a better domination checker algorithm.

Searching for dominated schedules is expected to be more effective when the total taskset utilization is lower. This is because, with lower utilizations, there are a lot

more opportunities for rearranging the jobs while looking for the optimal schedule. More such opportunities lead to more cases which are decidedly worse than another.

7.2 Performance Evaluation

We now discuss the effects of different parameters in our system model on the overall performance of the proposed algorithm. The same set of jobsets as generated in the previous section will be used for the evaluations here. The *Chronological Matcher* is used for dynamically expanding the graph, the *Stepped Heuristic* is used to compute the Cost Function and the *Depth Domination Checker* is used to find dominated schedules for pruning. Once again, the performance metric is considered to be the number of jobsets that did not finish within a 1-second timeout.

7.2.1 Number of Tasks

The time taken to find the optimal schedule is expected to be directly proportional to the number of jobs in the jobset. However, that is not the only quality of the input that affects the performance of the proposed algorithm.

As each task in a taskset can have only one job executing at any given time, the number of tasks should also affect the performance. The greater the number of tasks in the taskset, the more jobs are potentially available for scheduling at any instant of time, and hence a larger scheduling tree is generated. We expect that increasing the number of tasks should lead to a more time consuming search and hence more timeouts.

In Figure 7.4 we see the number of timeouts mapped against the size of the taskset at fixed total utilization values. Appendix B shows the same figure with the evaluation at all total utilization points.

As can be seen in the figure, for different values of the total utilization of the taskset, the effect of the number of tasks is the same. With an increasing number of tasks, the problem gets exponentially more difficult to solve resulting in more timeouts. This is exactly as we expect it. It is also interesting to see that the number of timeouts reduce with increasing total utilization. This is discussed in more detail in Section 7.2.2.

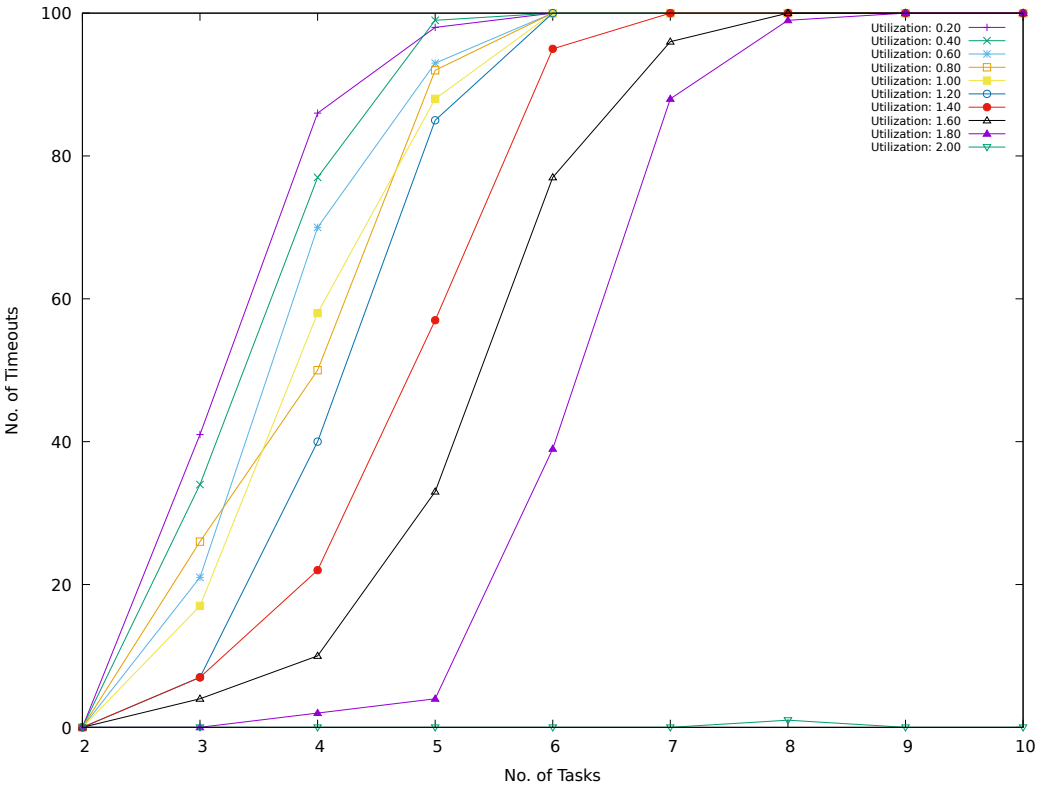


FIGURE 7.4: Effect of Taskset Size on Performance

7.2.2 Utilization

The total utilization of a taskset is bound to have an impact on the time it takes to search for the optimal schedule.

Figure 7.5 shows the effect of total utilization on the number of timeouts caused for 3 different taskset sizes. At first thought, one would intuitively expect that the number of timeouts increases with increasing utilization. As the total utilization increases, it increases the competition for the CPU time; this makes it a harder scheduling problem. However, for the purposes of finding the optimal solution this is not the case.

With a higher total utilization, there are fewer branches in the scheduling tree. Most potential re-arrangements of jobs lead to a deadline miss. In fact as the utilization approaches the number of available cores, the problem of finding the optimal schedule devolves into the problem of checking for feasibility of jobsets.

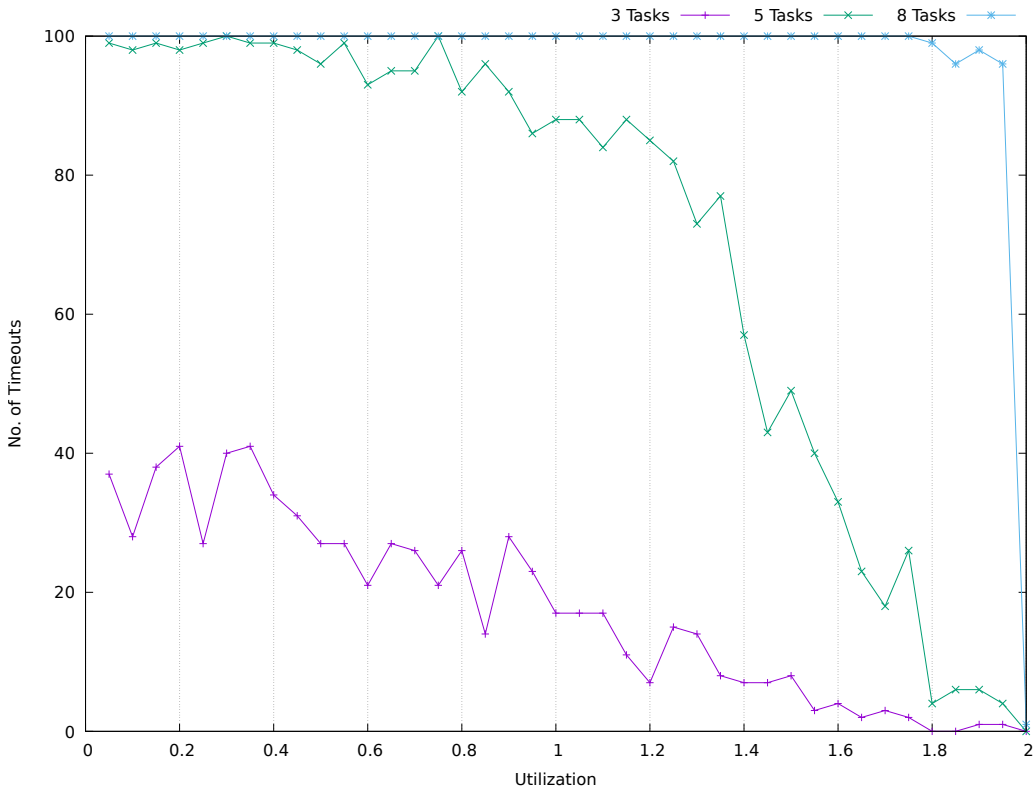


FIGURE 7.5: Effect of Total Utilization on Performance

7.2.3 Number of slices

Prior evaluations have considered a 1 KiB data cache split into 4 equal slices of 256 bytes each. In this section, we consider the effect of doubling the granularity of the cache slices from 4 slices to 8 slices. This makes each slice represent 128 bytes of the shared data cache.

With a finer cache slice granularity, we expect more jobsets to be schedulable in the system. However, with more slices, the scheduler also has more options of slice allocations that it must consider. This leads to a big growth in the size of the scheduling tree. As a result, with our one second timeout, we expect more jobs to be marked as timed-out and a few more jobs to be marked as feasible.

Figure 7.6 shows the effect of doubling the number of cache slices by reducing the granularity on each size. At three different values of the total taskset utilization, we see the same effect; fewer jobsets can be classified as feasible or infeasible. For cases with $n = 2$, we see a general increase in the number of feasible jobsets. In the

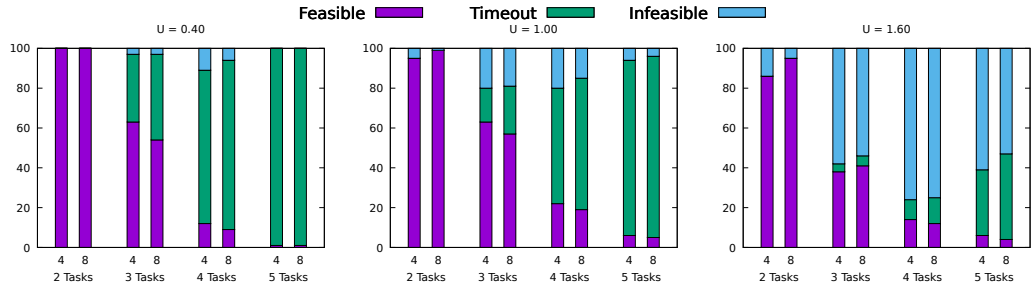


FIGURE 7.6: Effect of Cache Slices on Performance

other cases, the increased costs of evaluation lead to fewer scheduling trees being completely explored.

7.3 Schedulability

In Section 7.2.3, we saw that there exists potential for improving the number of feasible jobsets by decreasing the size of each cache slice. In this section, we discuss the case where the cache is statically partitioned per-core into equal slices. That is, each of the K cores always has exactly one slice of cache available, representing $\frac{1}{K}$ of the cache, and no repartitioning is allowed. We ran `AART` on the same jobsets as earlier to see how it performs with static, per-core partitioning of the cache.

Reducing the number of possible cache slice allocations to a unit value significantly trims down the size of the scheduling tree. Thus, we expect more jobsets to be classified as either schedulable or infeasible since the more scheduling trees can now be traversed within the 1s timeout. Figure 7.7 shows the case for $n = 3$, where static per-core partitioning seems to lead to a lot more feasible schedules.

The increase in schedulability compared to dynamic partitioning is explained by the fact that dynamic partitioning is a much more expensive algorithm. This can be seen in the fact that increasing the timeouts to one minute, brings the number of feasible schedules closer to that for static partitioning. With even longer analysis times, dynamically partitioned caches will eventually lead to more feasible schedules.

Figure 7.8 shows an overview of the results between static and dynamic partitioning at three different total utilization values. It can be clearly seen there that the reduction in feasible jobsets occurs due to the high volume of timeouts incurred during the analysis of dynamically partitioned caches.

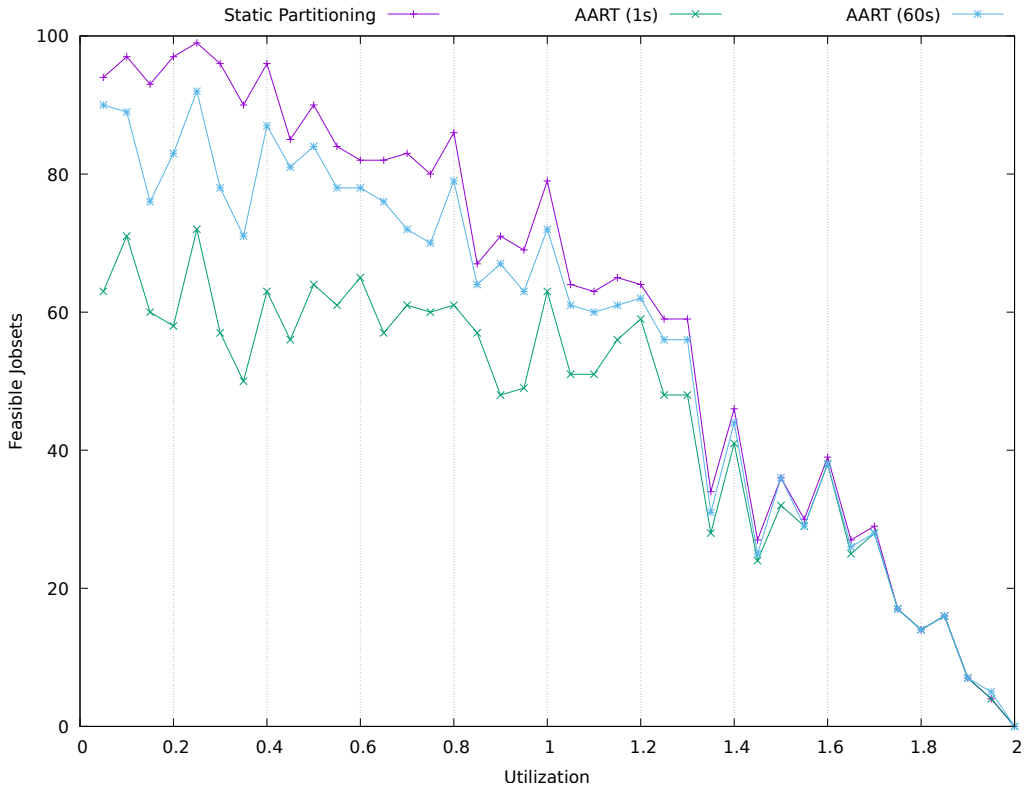


FIGURE 7.7: Schedulability Compared to Static Partitioning

7.4 Non-Optimal Scheduling

Chapter 6 introduced the idea of using a *scaling factor* to speed up the process of finding a schedule at the cost of optimality. Scaling factors work by eliminating potential schedules with finishing times that are too close to their deadline. This may lead to pruning the optimal schedule since one of its jobs misses the deadline during the computation of the heuristic.

We expect that scaling factors would display the largest benefits for tasksets with a low total utilization. As the utilization of the taskset increases, there is a greater chance that the only feasible schedules have finishing times close to their deadline and are hence deemed infeasible during the heuristic cost computation.

This can be seen in Figure 7.9, which shows the effects of using three different scaling factors: 0.5, 0.8 and 0.95. We can see that even high scaling factors of 0.95 result in more jobsets being classified as feasible. The guarantee with a scaling factor of 0.95 is that the resulting schedule will be no worse than the optimal schedule on a machine

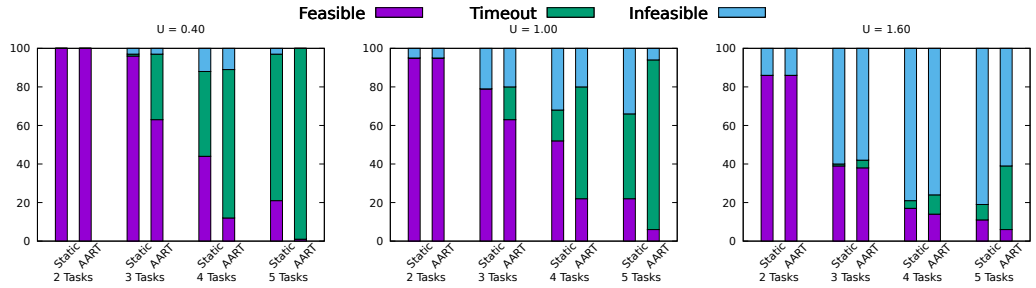
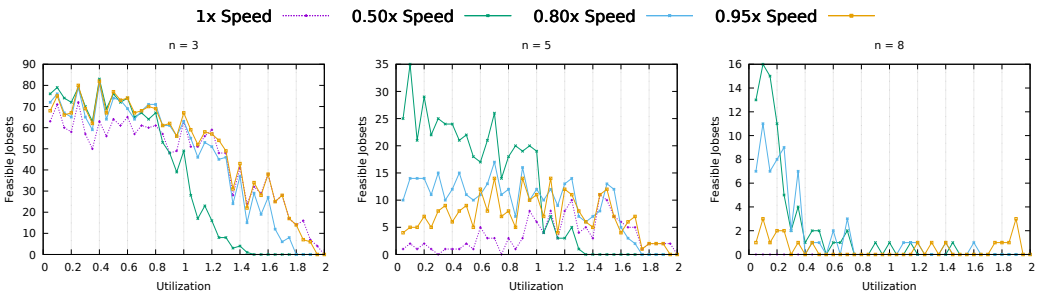


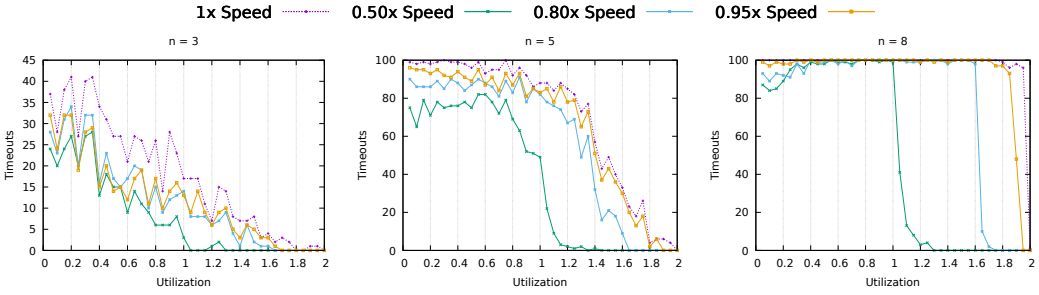
FIGURE 7.8: Schedulability Overview with Static Partitioning

running at 95% of the speed. At lower utilizations, it is always advantageous to use a scaling factor if the optimal schedule is not a hard requirement.

Figure 7.9 also shows that the significantly fewer jobsets cannot be classified within the 1s timeout. The sharp drops in number of timeouts in the graphs in Figure 7.9b is because around the $k \times s.f.$ utilization, the jobsets become completely infeasible.



(A) Feasible Jobsets



(B) Timeouts

FIGURE 7.9: Effect of Scaling Factor on Schedulability

Chapter 8

Conclusions and Future Work

Safety-critical hard real-time systems are often deployed in harsh conditions and are constrained in both size and power. They are also often over-provisioned to account for the worst-case scenarios. This makes it essential that we extract every last bit of efficiency from such systems.

Pre-computing an efficient schedule for each of their tasks is low-hanging fruit that can be leveraged in order to make more efficient use of the available hardware resources. Section 3.3 lays out an example that shows how existing solutions to the problem of cache interference in multi-core systems are not optimal. We show that dynamically partitioning the cache on a per-job basis provides more flexibility than static, per-core partitioning, and that this flexibility can be used to schedule tasks more efficiently on existing hardware. Allowing the cache to be repartitioned at any instant in time rather than only at context switches would be even more flexible. However, we do not consider such cases due to the inherent difficulty it poses in estimating the remaining WCET after the cache has been repartitioned.

In this thesis we present a novel method, based on the A* graph search algorithm to solve the NP-Hard problem of optimal cache partitioning and task to core assignment for multi-core schedules. Chapter 2 and Chapter 3 lay the groundwork for the rest of the thesis. They explain the ideas behind all the concepts such as cache partitioning, A* algorithm, etc. that are crucial to understanding the rest of the work. Chapter 3 also introduces the motivating example for this thesis.

Later, in Chapter 4 we discussed in-depth the modifications to the A* algorithm required for adapting it to the problem of real-time scheduling. Several techniques for computing the cost function and expanding the graph dynamically were also

presented. Since the complete scheduling tree remains very large, several methods for pruning non-optimal branches were then presented in Chapter 5.

An extensive evaluation of the algorithm was also conducted comparing the effects of each of the methodologies presented in this thesis. We find that it is always beneficial to use the Chronological Matcher as a graph expansion technique and the Stepped Heuristic for evaluating the cost function. Among the state-space pruning approaches, the List-Based Domination Checker shows potential but tends to lead to worse performance in most cases. The Depth Domination Checker simply delays the issue, but eventually also leads to a worse performance when compared against not pruning at all. However, the domination checkers do provide a space-time trade-off since using the domination checking techniques leads to lower peak memory usage when compared to no using such techniques.

Chapter 6 also introduces one method for efficiently computing a feasible schedule with a guarantee that the schedule will perform no worse than the optimal schedule on a slower processor. Such methods also allow us to trade-off optimality for faster results. The corresponding evaluation of this technique in Section 7.4 shows that even scaling the processor speed to 0.95x often leads to a 10% or more reductions in the number of timeouts encountered while improving the number of feasible jobsets by $\sim 10\%$.

8.1 Future Work

While these results are promising, they also betray a fatal flaw in the current approach. Searching the entire state space for an optimal schedule is expensive and requires a lot of time and power to compute. There are a few ideas we have to improve on the methods presented in this thesis.

Performance of State-Space Pruning Approaches

In particular, we see that the domination checking techniques consume 95% of the cycles at runtime. For larger jobset sizes, they also lead to a degradation in the overall performance. However, they do show that there exists potential for improving the performance through more efficient pruning approaches.

We believe that a better choice of a data structure for storing the internal state will help in improving its performance. Data structures such that R-Trees and KD-Trees seem very promising for this approach.

Early Identification of Feasible Schedules

Often, it may be beneficial to identify a feasible schedule quickly before trying to search for the optimal schedule. We showed one method that can be used for identifying feasible schedules in Chapter 6.

Other techniques such as using two separate heuristic evaluation functions, one admissible and another non-admissible may be an interesting idea that leads to early detection of feasible schedules. Such a method can then fall-back to searching for the optimal schedule once feasibility has been established.

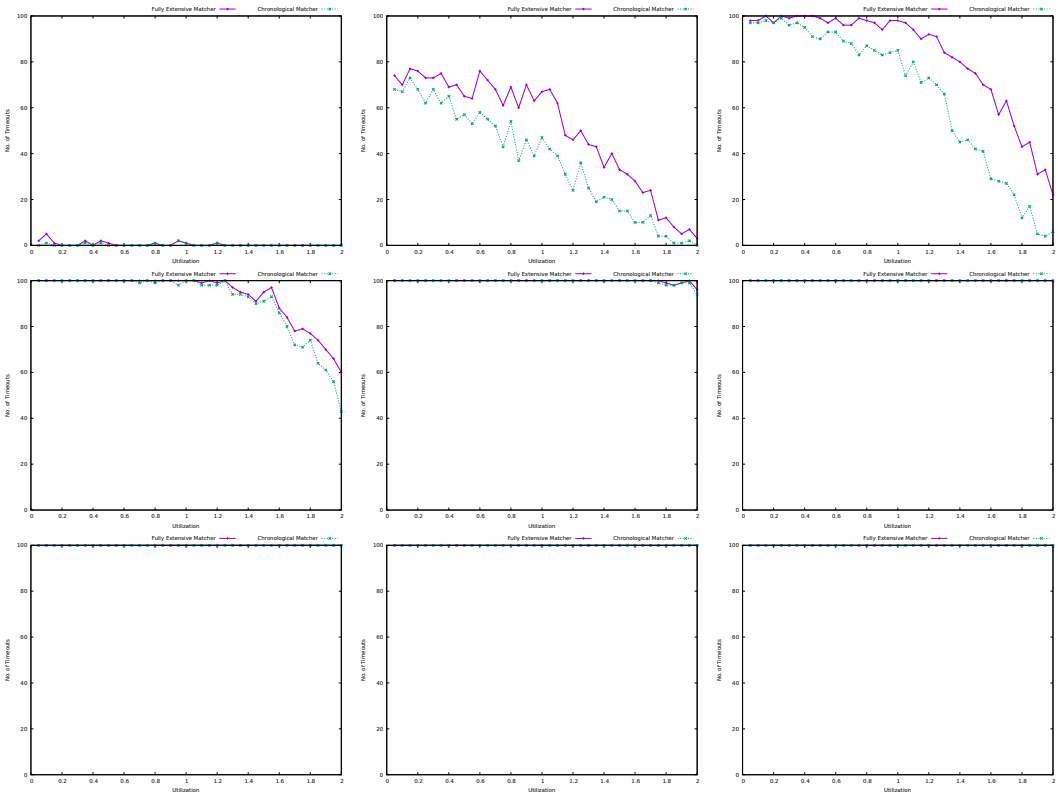
Different Heuristic Algorithms

We presented two different heuristic cost evaluation functions based on the intuition that Preemptible-EDF is an optimal scheduling algorithm on single-core machines. One issue with the presented functions is that they are relatively expensive to compute. The heuristic cost functions need to be very cheap since they are evaluated once for every node in the scheduling tree. Hence, finding another, more efficient heuristic function is also an important topic for future work in this area.

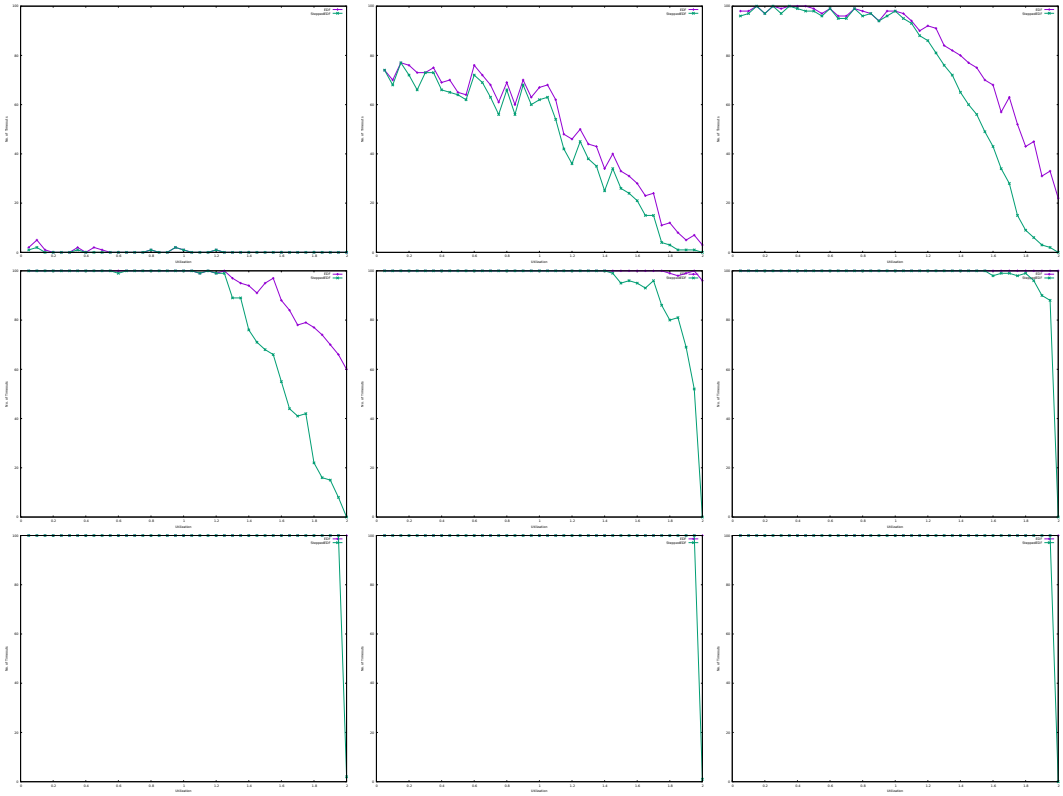
Appendix A

Effectiveness of Strategies

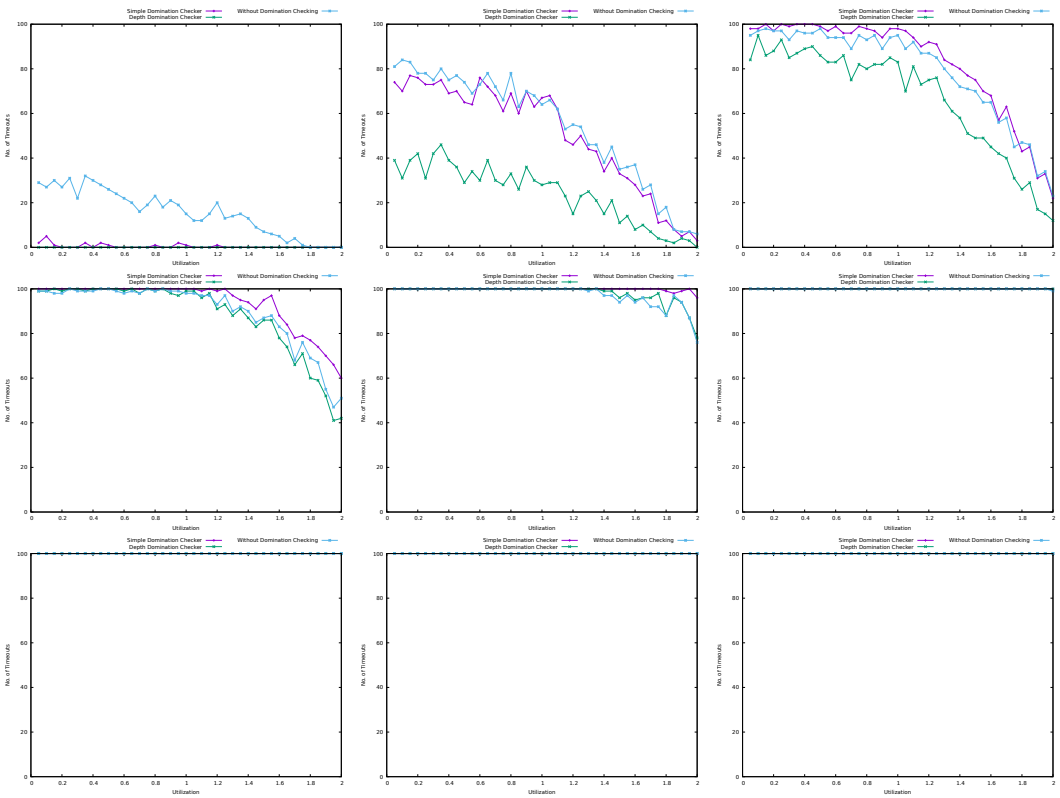
Dynamic Graph Expansion



Heuristic Functions



Domination Checkers



Appendix B

Effect of Taskset Parameters

Size of Taskset

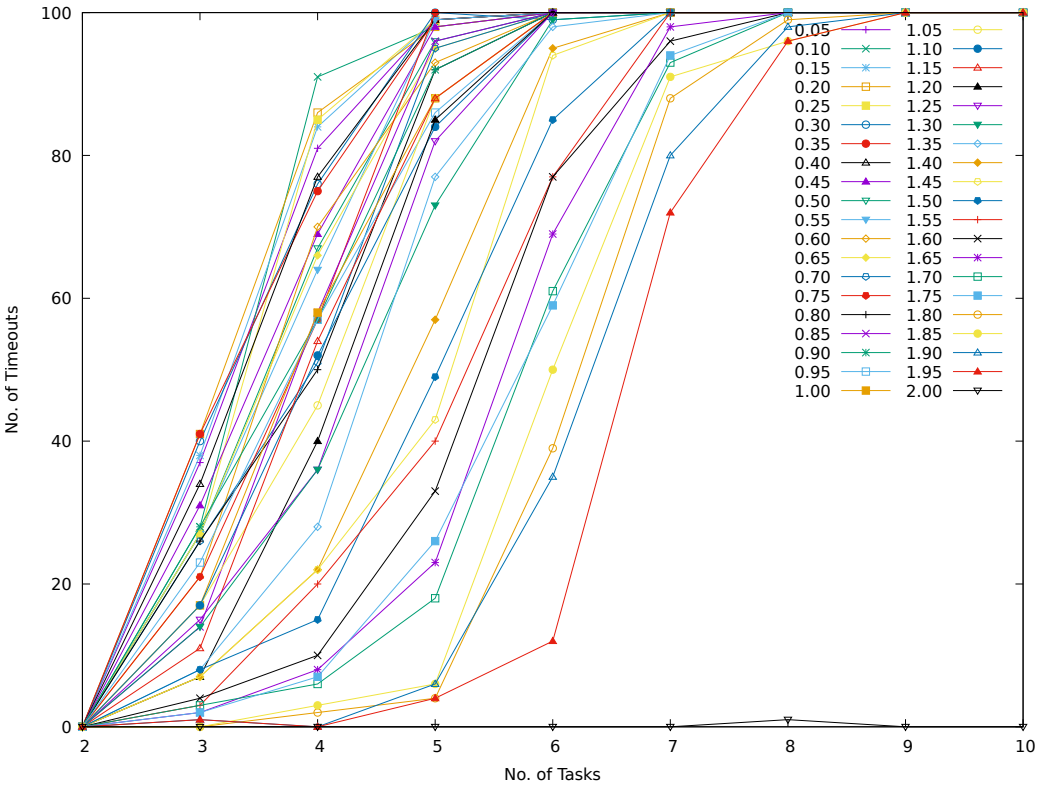


FIGURE B.1: Effect of Taskset Size on Performance

Total Utilization

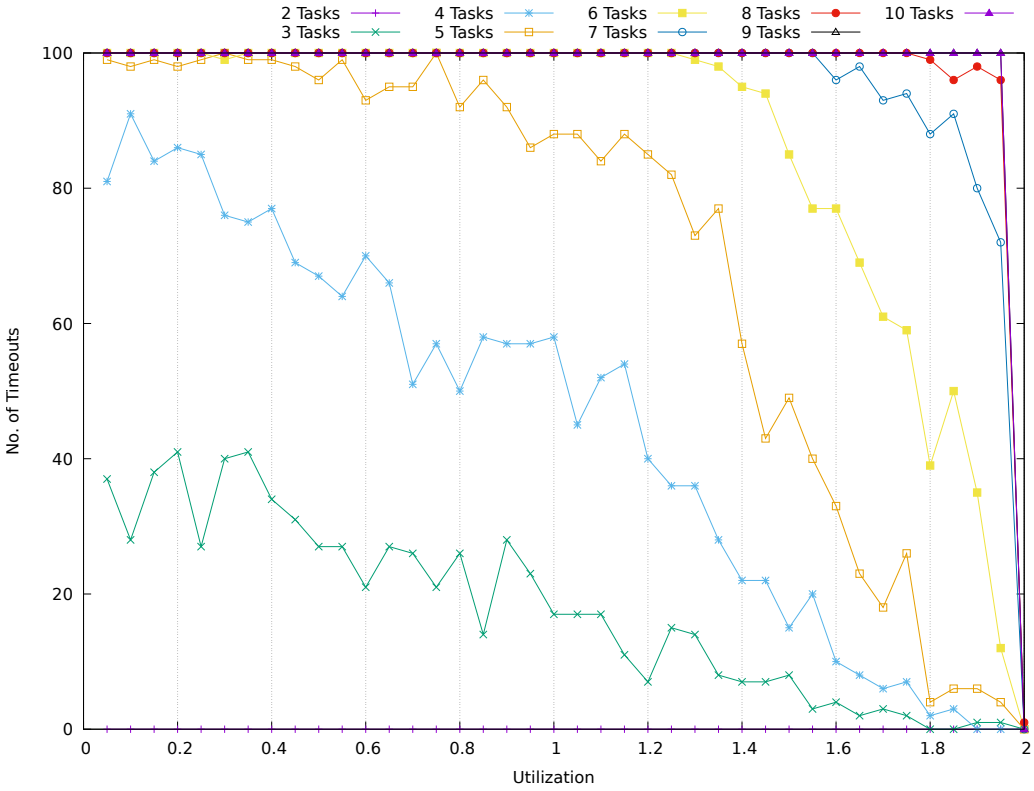


FIGURE B.2: Effect of Total Utilization on Performance

Bibliography

[ACD06] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms. In *IEEE Real Time Technology and Applications Symposium*, pages 179–190. IEEE Computer Society, 2006.

[Bak10] Theodore P. Baker. What to make of multicore processors for reliable real-time systems? In Jorge Real and Tullio Vardanega, editors, *Reliable Software Technologiey – Ada-Europe 2010*, pages 1–18, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BCSM08] Bach Duy Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*, pages 101–110. IEEE Computer Society, 2008.

[CC94] Hong Chich Chou and Chung-Ping Chung. Optimal multiprocessor task scheduling using dominance and equivalence relations. *Computers & OR*, 21(4):463–475, 1994.

[Cer16] Certification Authorities Software Team (CAST). Position Paper CAST-32A Multi-core Processors, November 2016.

[CJ94] Pei Chann Chang and Yen Shean Jiang. A state-space search approach for parallel processor scheduling problems with arbitrary precedence relations. *European Journal of Operational Research*, 77(2):208 – 223, 1994.

[CM05] A. Chousein and R. N. Mahapatra. Fully Associative Cache Partitioning with Don ' t Care Bits for Real-Time Applications Ali Chousein 3 . Fully Associative Cache for Partitioning. *ACM SIGBED Rev.*, 2(2):35–38, 2005.

[De 59] Rene De La Briandais. File searching using variable length keys. *ACM West. Jt. Comput. Conf.*, 1:295–298, 1959.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[DM89] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec 1989.

[DZ13] Yiqiang Ding and Wei Zhang. Multicore real-time scheduling to reduce inter-thread cache interferences. *JCSE*, 7(1):67–80, 2013.

[ESD10] P Emberson, R Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS*, 01 2010.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *WCET*, volume 55 of *OASICS*, pages 2:1–2:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[FAK+12] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48. ACM, 2012.

[FBN60] Edward Fredkin, Bolt Beranek, and Newman. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[GA83] Malik Ghallab and Dennis G. Allard. $A_{epsilon}$ - an efficient near admissible heuristic search algorithm. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pages 789–791, 1983.

[GSYY09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 245–254, 2009.

[Hah19] Sebastian Hahn. *On Static Execution-Time Analysis — Compositionality, Pipeline Abstraction, and Predictable Hardware*. PhD thesis, Saarland University, 2019.

[HJR16] Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *RTNS*, pages 299–308. ACM, 2016.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[Hor74] W. A. Horn. Some simple scheduling algorithms. *Nav. Res. Logist. Q.*, 21(1):177–185, 1974.

[HVA+16] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. *Proc. - Int. Symp. High-Performance Comput. Archit.*, 2016-April:657–668, 2016.

[Int16] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3b: edition, 2016.

[KA05] Yu-Kwong Kwok and Ishfaq Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *J. Parallel Distrib. Comput.*, 65(12):1515–1532, 2005.

[Kir] D Kirk. Smart (strategic memory allocation for real-time) cache design. *Rtss '89*.

[KK83] Vipin Kumar and Laveen N. Kanal. A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artif. Intell.*, 21(1-2):179–198, 1983.

[KKR13] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. *Proc. - Euromicro Conf. Real-Time Syst.*, pages 80–89, 2013.

[KP82] J H Kim and J Pearl. Studies in semi-admissible heuristics. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(4):392–399, 1982.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. *WATERS*, 01 2015.

[LCG+15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ISCA*, pages 450–462. ACM, 2015.

[Lee16] Marissa Lee. EMA, Singapore Power to test-bed energy storage system. *Straits Times*, pages 257–266, 2016.

[LGBS05] C. C. Liu, I. Ganusov, M. Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3d ic technology. *IEEE Design Test of Computers*, 22(6):556–564, Nov 2005.

[LHH97] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, June 1997.

[LLD+] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Soft-OLP - Improving Hardware Cache Performance Through Software-Controlled Object-Level Partitioning.pdf.

[LPM09] Paul Lokuciejewski, Sascha Plazar, and Peter Marwedel. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *WCET*, volume 10 of *OASICS*, pages 1–11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[LSK04] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. *IEEE High-Performance Comput. Archit. Symp. Proc.*, 10:176–185, 2004.

[MCHvE06] Anca Mariana Molnos, Sorin Dan Cotofana, Marc J. M. Heijligers, and Jos T. J. van Eijndhoven. Throughput optimization via cache partitioning for embedded multiprocessors. In *ICSAMOS*, pages 185–192. IEEE, 2006.

[MKR10] Sai Prashanth Muralidhara, Mahmut Kandemir, and Padma Raghavan. Intra-application cache partitioning. *Proc. 2010 IEEE Int. Symp. Parallel Distrib. Process. IPDPS 2010*, 2010.

[NS14] Kartik Nagar and Y. N. Srikant. Precise shared cache analysis using optimal interference placement. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 125–134. IEEE Computer Society, 2014.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Poh70] Ira Pohl. First results on the effect of error on heuristic search. *Machine Intelligence.*, 5, 01 1970.

[Poh73] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 12–17, 1973.

[PTH11] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling Cache Utilization of HPC Applications. *Ics'11*, page 295, 2011.

[RLT06] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. *Proc. 15th Int. Conf. Parallel Archit. Compil. Tech. - PACT '06*, page 2, 2006.

[RMMA15] Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. WCET analysis in shared resources real-time systems with TDMA buses. *Proc. 23rd Int. Conf. Real Time Networks Syst. - RTNS '15*, pages 183–192, 2015.

[RS09] Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl.*

Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[SG92] A. D. Stoyenko and L. Georgiadis. On optimal lateness and tardiness scheduling in real-time systems. *Computing*, 47(3-4):215–234, 1992.

[SHK14] H. Shah, K. Huang, and A. Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 708–713, Jan 2014.

[SKI08] Shekhar Srikantaiah, Mahmut Kandemir, and MJ Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ACM Sigplan Not.*, 42(2):135–144, 2008.

[SKS16] Prateek Sharma, Purushottam Kulkarni, and Prashant Shenoy. Per-VM page cache partitioning for cloud computing platforms. *2016 8th Int. Conf. Commun. Syst. Networks, COMSNETS 2016*, 2016.

[SM08] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. *Proc. - Des. Autom. Conf.*, pages 300–303, 2008.

[SR18] Darshit Shah and Jan Reineke. Towards optimal offline scheduling for multi-core systems with partitioned caches. In Martina Maggio, editor, *Proceedings Work-In-Progress Session of the 30th Euromicro Conference on Real-Time Systems*, pages 4–6, July 2018.

[SS16] Stefanos Skalistis and Alena Simalatsar. Worst-Case Execution Time Analysis for Many-Core Architectures with NoC. *Form. Model. Anal. Timed Syst.*, 6246(October 2017), 2016.

[vdBUCF17] Georg von der Brüggen, Niklas Ueter, Jian-Jia Chen, and Matthias Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 108–117, New York, NY, USA, 2017. ACM.

[WHKA13] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. *Euromicro Conf. Real-Time Syst.*, pages 157–167, 2013.

[XAP17] Jun Xiao, Sebastian Altmeyer, and Andy D. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *RTSS*, pages 199–208. IEEE Computer Society, 2017.

[YMWP14] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. *Real-Time Technol. Appl. - Proc.*, 2014-October(October):155–166, 2014.

[ZWN17] Wenguang Zheng, Hui Wu, and Chuanyao Nie. Integrating task scheduling and cache locking for multicore real-time embedded systems. In *LCTES*, pages 71–80. ACM, 2017.