Solving Schedulability as a Search Space Problem with Decision Diagrams

Dimitri Racordon, Aurélien Coet, Emmanouela Stachtiari, and Didier Buchs

Computer Science Department
Faculty of Science,
University of Geneva, Switzerland
first.last@unige.ch

Abstract. Real-time system design involves proving the schedulability of a set of tasks with hard timing and other constraints that should run on one or several cores. When those requirements are known at design time, it is possible to compute a fixed scheduling of tasks before deployment. This approach avoids the overhead induced by an online scheduler and allows the designer to verify the schedulability of the taskset design under normal and degraded conditions, such as core failures. In this context, we propose to solve the schedulability problem as a state space exploration problem. We represent the schedulings as partial functions that map each task to a core and a point in time. Partial functions can be efficiently encoded using a new variant of decision diagrams, called Map-Family Decision Diagrams (MFDDs). Our setting allows first to create the MFDD of all possible schedulings and then apply homomorphic operations directly on it, in order to obtain the schedulings that respect the constraints of the taskset.

Keywords: Search Problems · Decision Diagrams · Schedulability · Real-Time Systems · Resilient Systems · Multi-Core Architectures

1 Introduction

Multi-core architectures have become ubiquitous, as an answer for the exponential growth in computer performance required by modern applications. This observation obviously applies to small-scale real-time and cyber-physical systems as well. Unlike more general applications, these systems often have to run tasks with hard deadlines, to interact with their hardware components. It follows that one of their essential requirements is to guarantee that they are able to perform all their tasks on time, by providing a scheduling that assigns each task to a specific core at a specific point in time. Such a scheduling can be built along with the execution of the system, according to some heuristics [11], or pre-computed statically to avoid the overhead induced by an online scheduler. An additional advantage of the latter approach is that it allows to study the system's performance under various scenarios, not only to make sure it is actually capable of running its workload, but also to check whether it can be resilient to hardware faults (e.g. the failure of one of its cores).

Typically, each task has timing constraints, such as a *release time*, after which it can be executed, a *worst-case execution time*, which is the most pessimistic assumption for the time it takes to complete, and a *deadline* for its completion. Other non-timing constraints may exist and should also be taken into account in the scheduling. Common constraints include *precedence*, which indicates that a predecessor task has to be completed before a successor task starts. Schedulability analysis verifies that a *feasible* scheduling exists for a given taskset respecting all the timing, precedence, and other constraints. Roughly speaking, it suffices to check that all tasks will complete before their deadline.

Several analysis tools have been used to solve the multicore schedulability problem. For example, *utilization bound checks* [12] have been proposed for testing the schedulability of a taskset analytically. Though these checks are efficient, they inherently are pessimistic, often rejecting valid tasksets. Furthermore, they cannot handle multiple constraints over tasks. Other tools have been developed to analyze tasksets with complex constraints [14], and rely on *simulation* to check schedulability. However, simulation is known to cover only a sample of possible scenarios, thus it can lead to falsely feasible schedulings, which is not acceptable for critical real-time systems.

The aforementioned approaches in the literature target a specific variation of the problem, such as the existence of task parallelism or interference among tasks that cause delays. In this paper, we opt for a different, more generic approach, which relies on model checking. Unlike simulation, model checking explores the entire space of possible states. In our case, this translates into an enumeration of all possible schedulings, that we can filter to remove specific instances for which the constraints are not satisfied. In practice, such an approach is often intractable due to the *state-space explosion problem*. However, we mitigate this issue with decision diagrams, a data structure that encodes large sets of data into a memory-efficient representation by exploiting the similarities between each element. Our work is related to the technique proposed in [17]. The authors solve the schedulability problem as a state-space exploration using Data Decision Diagrams [6], which encode sets of variable assignments. We use a slightly different flavor of decision diagrams, called Map-Family Decision Diagrams (MFDDs), that encode sets of partial functions, and allow for a more direct translation of the problem. Each constraint is represented as a homomorphic operation that is applied directly on the encoded form, in a fashion reminiscent to Fourier filters.

This paper offers the following contributions:

1. An exhaustive search methodology for the multi-core schedulability problem, based on inductive homomorphisms, i.e. structure-preserving transformations, that compute all solutions at once.
2. A compact, human-readable representation of the solution set, which can be easily inspected during design and efficiently stored as a database of precomputed schedulings in production.
3. A refinement of the seminal work proposed in [17] that fixes a flaw in the authors' method, offers a simpler way to define filters and supports additional constraints, such as transient core failures.

2 Related Work

In this section, we include works related to the offline computation of global multicore scheduling for time-triggered tasks. In particular, we discuss approaches that employ model checking, linear programming and decision diagrams.

Model checking has been a reliable tool for schedulability verification. Its inputs are a model of the system (e.g., tasks, scheduler) with finite reachable states and a set of properties that characterize the valid reachable states, e.g., states where no task has missed its deadline. Properties may be expressed as temporal logic formulae or be designated as *error states* in the model. An exhaustive search explores the reachable states of the system model and checks if properties hold. Violating states are returned as *counter-example*, which can be used to refine the model.

In [1], periodic tasksets are modeled as timed automata and schedulability holds if a certain state can be reached within an expected time window. In a similar approach [8], each task and its constraints are modeled as a timed automaton, on which it is verified that the task can meet its deadline. Interactions between tasks are modeled by composing their corresponding automata. Another work [18] for self-suspending tasks models each task as as a set of segments, the end of which corresponds to a suspension point. Generally, model checking can model complex factors on schedulability, such as stochastic execution times. However, it is computationally expensive and therefore only able to handle small tasksets. Moreover, the generated solutions are independent to each other and can not be narrowed down efficiently to account for additional task constraints or be compactly stored. In [7], statistical model checking is proposed to reduce the undecidability of symbolic (i.e., more efficient but over-approximating) model checking for the scheduleability of tasks with uncertain response and blocking times. In this setting, the system is simulated for a finite number of runs to test the satisfaction probability of a given property. Tasksets that were found unschedulable by symbolic model checking can be probed for useful information, such as the probability of a certain violation, or bounds on blocking times. Other timed automata extensions, such as Priced Automata [2], have been suggested to efficiently identify subsets of feasible schedulings.

Other approaches to schedulability rely on linear programming. In [16], schedulability for multi/many-core architectures is studied for three different platform architectures using integer linear programming (ILP). Cache conscious real-time schedulability using ILP has also been the target in [15]. In their setting, they assume that all tasks are connected by the dependency relation, which reduces candidate solutions, compared to our setting that does not assume that. Linear programming uses heuristics algorithms to efficiently compute schedulings close to the optimal ones, but it does not return all schedulings.

Decision diagrams have been proposed for representing and solving schedulability In [10] the authors find an optimal schedule for tasks with arbitrary execution times using a breath firth search on a Binary Decision Diagram. However, their representation considers uniform cores, while our model can be also applied to non-uniform cores, e.g., when not all cores are suitable for every task.

In another work [5], the authors proposed searching for the optimal single-core scheduling, using a Multi-Valued Decision Diagram that represents an overapproximated set of possible schedulings. As opposed to theirs, our method handles many cores and computes all feasible solutions.

3 Background

Decision diagrams were originally proposed as a data structure to represent and manipulate Boolean functions [4]. Since then, numerous variants have been developed that suit other domains (see [13] for a survey). Nonetheless, all of them share the same principle; they encode each element as a path in a finite directed acyclic graph (DAG), from its root to a terminal node (i.e. a node without successors). Non-terminal nodes are labeled with a variable, and arcs are labeled with the value assigned to this variable. Our approach uses one specific variant of decision diagrams, named MFDDs, which we developed to encode sets of partial functions $f : A \to B$, where A and B are any sets and $dom(f) \subseteq A$ is finite. Partial functions typically correspond to dictionaries or mappings in regular programming languages, and are well-suited to encode various kinds of data structures. For instance, a list can be seen as a partial function that maps numerical indices to the list's elements.

In a MFDD that encodes a set of partial functions $A \to B$, all non-terminal nodes are labeled with a value from A, while arcs are labeled with a value from $B \cup \{\epsilon\}$, where ϵ represents the absence of any value. A diagram may have up to two terminal nodes, labeled with $\top$ and $\bot$, representing the acceptance of a path from the root and its rejection, respectively. Hence, a function is encoded as a path from the root to the accepting terminal node.

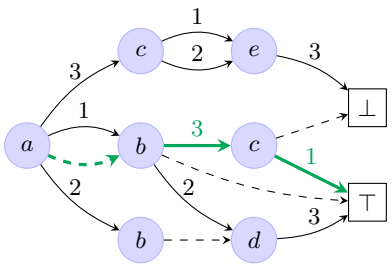


Fig. 1: Example of a (non-canonical) MFDD

Example 1. Figure 1 depicts a decision diagram that encodes a set of seven different partial functions from a domain $A = \{a, b, c, d, e\}$ to a codomain $B \subseteq \mathbb{N}$. Circles denote domain values, while squares denote terminal nodes. Dashed edges represent ϵ-arcs, while solid ones are labeled with a number. The path highlighted

with thick, green arrows encodes a function f such that $dom(f) = \{b, c\}$ (i.e., it is undefined for any other value in A) and $f(b) = 3, f(c) = 1$.

Any path that leads to the rejecting terminal (i.e., $\bot$) is said *rejected* and denotes a function that is not present in the encoded set. If some domain value v is absent from a path, then it is assumed that the corresponding function is not defined for v. All functions that cannot be associated with any accepted path in the diagram are considered absent from the encoded set. It follows that a given set of functions can have multiple representations[1].

Definition 1 (Map-Family Decision Diagram). *Let A and B be a domain and a codomain set, respectively. The set of Map-Family Decision Diagrams $\mathbb{M}_{A,B}$ that encode families of partial functions $A \to B$ is inductively defined as the minimum set, such that:*

- $\{\bot, \top\} \in \mathbb{M}_{A,B}$ *are terminal nodes,*
- $\langle a, s\rangle \in \mathbb{M}_{A,B}$ *is a non-terminal node labeled with $a \in A$, whose successors are given by the partial function $s : B \cup \{\epsilon\} \to \mathbb{M}_{A,B}$.*

Efficient implementations of decision diagrams rely on representation uniqueness to share identical sub-graphs, in order to reduce the memory footprint and cache the result of each homomorphic operation. We provide a canonical representation of MFDDs by applying some constraints. Firstly, we require that domain A be associated with a total order, and that all successors of a node be either a terminal node or a node labeled with a greater value. This is the case in the MFDD of Figure 1, assuming that the members in A are ordered lexicographically. Secondly, we require that all non-terminal nodes have at least one arc not labeled with ϵ. Nodes that do not satisfy this constraint can be safely removed, as they do not carry any information. For instance, node b at the bottom path of Figure 1 is redundant. Finally, we require that all rejected paths be removed. Recall that functions which cannot be associated with an accepted path are assumed to be rejected (i.e. absent from the encoded set). Hence, rejected paths, like the two top paths of Figure 1, do not add any information. Note that we cannot get rid of the rejecting terminal itself, as it is necessary to represent the empty set of functions. The canonical form of the the MFDD from Figure 1 is shown in Figure 2a.

Definition 2 (Canonicity). *Let A be a domain set with a total order $< \subseteq A \times A$ and B be a codomain set. A MFDD $d \in \mathbb{M}_{A,B}$ is canonical if and only if:*

- $d \in \{\bot, \top\}$ *is a terminal node, or*
- $d = \langle a, s\rangle$ *is a non-terminal node such that $\exists b \in dom(s), b \neq \epsilon$ and $\forall b \in dom(s), s(b) \in \mathbb{M}_{\{x \in A | a < x\},B} - \{\bot\}$.*

[1] Incidentally, as MFDDs are *finite* graphs, it follows that all encoded functions f have a finite domain $dom(f) \subseteq A$, represented by the non-terminal nodes along an accepting path, even if A is infinite.

Let $\langle a, s \rangle$ be a non-terminal node. The first canonicity constraint is enforced by $s(b) \in \mathbb{M}_{\{x \in A | a < x\}, B}$, which prescribes that successor nodes be labeled with greater domain values. The second constraint is enforced by $\exists b \in dom(s), b \neq \epsilon$. Finally, although the third constraint is not enforced explicitly, it is easy to show that requiring $s(b) \neq \bot$ inductively prevents rejected paths to be encoded.

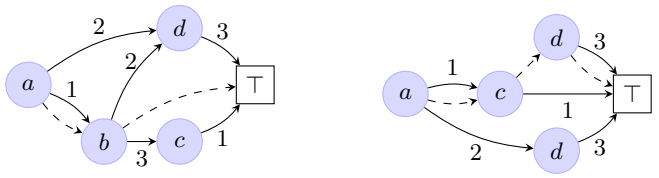


Fig. 2: Two examples of canonical MFDDs

Example 2. Consider the MFDD shown in Figure 2a. The root node is a tuple $\langle a, s_a \rangle$, where s_a is a function such that $dom(s_a) = \{\epsilon, 1, 2\}$ and $s_a(\epsilon) = s_a(1) = \langle b, s_b \rangle$ and $s_a(2) = \langle d, s_d \rangle$. The function s_b is defined such that $dom(s_b) = \{\epsilon, 3\}$ and $s_b(\epsilon) = \top$ and $s_b(3) = \langle c, s_c \rangle$. Finally, s_c (resp. s_d) is defined such that $s_c(1) = \top$ (resp. $s_d(3) = \top$) and is undefined for any other value in $B \cup \{\epsilon\}$.

In addition to their compact representation, another advantage of MFDDs is that they can be manipulated by the means of homomorphisms. These operations can modify multiple elements at once, as any alteration of a prefix has a direct impact on all elements encoded by its suffixes.

Example 3. Suppose we were to remove b from the domain of all partial functions encoded by the MFDD in Figure 2a. Rather than enumerating all seven instances to apply the filter, we could define a homomophism that simply removes the node corresponding to b's bindings, and rewires its incoming arcs to existing or new nodes, as shown in Figure 2b. In other words, a homomophism can modify the domain of all encoded functions by rewriting the decision diagram.

A key property of MFDD homomorphisms is that they preserve set-theoretic operations, such as union and intersection. More formally, let $d_1, d_2 \in \mathbb{M}_{A,B}$ be two MFDDs, and Φ be a homomorphism, then $\Phi(d_1 \cup d_2) = \Phi(d_1) \cup \Phi(d_2)$. This allows homomorphisms to be rearranged for efficient computations.

4 Methodology

In the context of search based problems, MFDDs present the advantage that they can be used to both store and compute sets of solutions efficiently. Their compact representation is able to encode large sets with minimal memory footprint, while

homomorphisms on map families allows the construction of solutions with a smaller computational overhead than traditional approaches.

There are two main approaches to use MFDDs in search based problems. The first consists of exploring the state space of a problem to build its solution set incrementally, augmenting it with new instances that satisfy the problem's constraints as they are found. This process ends when a fixpoint is reached or if the entire space has been visited. The second technique, more reminiscent of Answer Set Programming (ASP), proposes to start from a MFDD representing the entire state space of the problem before filtering out instances that do not satisfy the problem's constraints.

The *n-queens puzzle* is a simple example of a problem that can be solved with the second approach. The puzzle consists of finding all possible ways that n different queens can be placed on a $n \times n$ chessboard without being able to attack each other. More formally, let $Col_n = \{a, b, \ldots\}$ denote a set of column identifiers for some $n \times n$ chessboard. Similarly, let $Row_n = \{1, 2, \ldots, n\}$ denote row identifiers. Let $C_n = Col_n \times Row_n$ denote the set of coordinates on the board. Let $I \in \mathcal{P}(C_n)$ denote a set of coordinates at which queens are placed, and R denote a relation on coordinates which indicates whether a position can be reached from another by a queen, according to the rules of chess. $(d, 3) \in I$ indicates for example that a queen lies at row d and column 3 of the board, and $\langle (d, 3), (g, 6) \rangle \in R$, because the position $(g, 6)$ can be reached from $(d, 3)$ by a queen. A configuration I is said to be a solution to the n-queens puzzle if it contains exactly n coordinates, and if for all $a \in I$, there is no $b \in I$ such that $\langle a, b \rangle \in R$. Hence, the set of all solutions S is formally given as:

$$S = \{I \mid (\parallel I \parallel = n) \wedge (\forall a, b \in C_n, a \in I \wedge \langle a, b \rangle \in R \Rightarrow b \notin I\}$$

Using MFDDs, one can express the n-queens puzzle as the following algorithm:

1: $S_n \leftarrow \mathcal{P}(C_n)$
2: **for all** $a \in C_n$ **do**
3: $\quad S_n \leftarrow \{I \in S_n \mid a \in I \Rightarrow \nexists b \in I, \langle a, b \rangle \in R\}$
4: **end for**
5: $S_n \leftarrow \{I \in S_n \mid\mid\mid I \parallel = n\}$

At line 1, all possible configurations are computed, as the powerset of the coordinates C_n, and added to the set of candidate solutions S_n. Then, at line 3, the set of candidate solutions is refined by removing all configurations in which a queen can attack another. Finally, at line 5, all sets with cardinality other than n are filtered out, so as to keep in S_n only the actual solutions to the puzzle.

Implementing such an algorithm with MFDDs can be quite efficient, thanks to the use of homomoprhisms. As mentioned in the previous sections, they allow to perform filtering directly on the shared structure of a MFDD, therefore modifying large subsets of the encoded family at once. For instance, in the above algorithm, the actions performed on S_n operate on each map it contains in parallel, without the need to iterate over each one of them separately. In addition, MFDDs allow the maps in S_n to share nodes, so as to keep the overall represen-

tation compact. Although a 8×8 chessboard has roughly 10^{19} configurations, it takes only 2.5 KB to store in memory.

5 Multi-Core Schedulability

The multi-core schedulability problem [17] can be viewed as an assignment problem that consists of verifying whether a set of tasks can be executed on a multi-core system, with respect to task-specific timing constraints and dependencies between the tasks. We focus on the most common type of precedence dependencies, where the predecessor must finish before the successor can start.

We define tasks T with associated triples $\langle r, c, d \rangle$, where r denotes their release time, c denotes their worst-case execution time and d denotes their (absolute) deadline. We assume that all tasks can potentially meet their deadline, i.e., $r + c \leq d$. A *task model* $M = \langle T, \mu, D \rangle$ is a DAG consisting of a set of tasks T as its vertices, a function $\mu : T \to \mathbb{N}^3$ defining tasks characteristics and a relation $D \subseteq T \times T$ describing *direct* dependencies between tasks as its edges.

Consider the task model depicted in Figure 3, that features five tasks. Task t_0 can be scheduled at time 0, i.e., immediately when it is released, since it has no dependencies. Tasks t_2 and t_4 cannot be scheduled at time 0, even though they have no dependencies, because they are released at times 4 and 8, respectively. Task t_1 depends on both t_0 and t_2. Hence, it cannot start before they have completed, even though its release time is earlier. Finally, task t_3 depends on t_1, meaning that it also indirectly depends on t_0 and t_2 (i.e. t_1's dependencies).

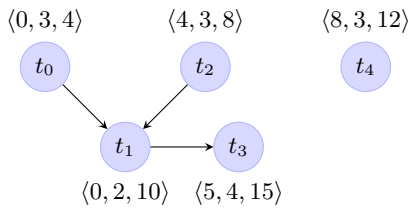


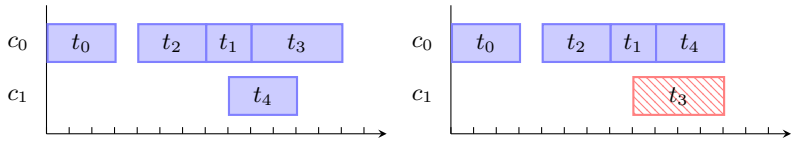
Fig. 3: Example of a task model consisting of 5 tasks.

Given a task model $M = \langle T, \mu, D \rangle$ and a set of cores C, a *scheduling* is a partial function $s : T \to C \times \mathbb{N}$, that assigns a task to a core at a specific time. Notice that this definition assumes that tasks cannot be preemptively suspended to execute another task on the same core, effectively meaning that all tasks are assumed to be executed from beginning to end. A scheduling is *feasible* if:

- all tasks are scheduled after their dependencies have finished;
- all tasks are scheduled after their release time; and
- the deadlines of all tasks are met.

Moreover, a scheduling is *consistent* if there are no tasks scheduled on the same core at the same time.

Figure 4 illustrates two examples of schedulings for the task model in Figure 3. Both schedulings are consistent, since tasks do not overlap. However, only the left one is feasible, since the right one does not satisfy t_3's dependencies.



(a) Feasible and consistent. (b) Non-feasible but consistent.

Fig. 4: Two examples of schedulings.

Definition 3 (Schedulability problem). *Given a task model and a set of cores, the schedulability problem consists of determining whether there exists at least one feasible and consistent scheduling.*

5.1 Schedulings as Decision Diagrams

Recall that MFDDs encode sets of partial applications. Since a scheduling is a partial function mapping each task to a core and a start time, encoding a set of schedulings with a MFDD is straightforward: tasks are represented by non-terminal nodes, whereas the start time and the core on which they are scheduled label the arcs. There are however two issues to address, which relate to the same limitation of decision diagrams. Because they are DAGs, extracting information from suffixes is usually challenging. Not only it initiates recursive explorations, which are costly on large diagrams, it also requires complex transformations to preserve the consistency of the prefix that leads to a particular node, when a parent is mutated. Such operations are usually avoided, in favor of homomorphisms that depend on values which are read on a prefix to a given node.

This limitation impacts the initial construction of the MFDD. Scheduling a task on a given core necessitates to know *when* the core is next available, which is an information that depends on the suffix of a given path. One way to tackle this issue is to lift the necessary information at the root of the MFDD. That way, we can first collect the next available time for a given core, before inserting a new mapping and schedule a new task on it.

The second issue relates to the handling of task dependencies. In order to decide if a scheduling is feasible, we need to determine whether each task is scheduled after all its dependencies. Consequently, the order in which tasks appear along a path in the MFDD must be chosen carefully, so that dependencies are laid out deeper. That way, we can first dive to the nodes representing a

specific task, and then remove all suffixes for which its dependencies are either not scheduled or finish too late.

Figure 5 represents the two schedulings depicted in Figure 4. The top path (resp. bottom path) corresponds to the left (resp. right) scheduling. The shared suffix shows how similarties between different schedulings can be exploited to compact the representation of a large set of possibilities.

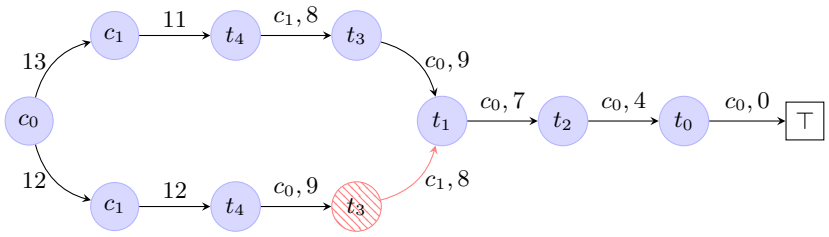


Fig. 5: Two schedulings represented as a MFDD.

Note that the order in which the tasks are laid out has a direct impact on sharing. Despite the constraint we mentioned earlier, namely that tasks should appear before their dependencies, there is a lot of freedom to pick this order. This is generally a difficult problem in the context of decision diagram optimizations [3, Chapter 7]. There are nonetheless some intuitions about what constitutes a good order. Indeed, sharing is most likely to occur on the bottom of the MFDD. Thus, tasks that are more loosely constrained, in terms of dependencies, release times and deadlines, should appear closer to the top. That way, more tightly constrained tasks, which will intuitively have less possible different assignments, will be found in the suffixes of more nodes.

The reader will notice that our encoding does not handle sporadic tasks. Such tasks are common in real-time systems, and corresponds to actions that must be executed periodically. As we chose to formalize schedulings in the form of partial functions $T \to C \times N$, representing sporadic tasks would require T to be an infinite set. However, our definition of a task model is sufficient to describe a time window, in which occurences of sporadic tasks can be enumerated. As a result, any feasible and consistent scheduling can represent one time window in the unbounded behavior of the system, consisting of repeated time windows.

5.2 Computing Schedulings

We now describe the computation of the set of all possible schedulings by a MFDD of $\mathbb{M}_{T \cup C, \mathbb{N} \cup (C \times \mathbb{N})}$. In a nutshell, the method consists of iteratively scheduling one task, at one possible time slot, in all schedulings that have been computed so far. This produces a new subset of schedulings at each iteration, that is merged with the original set, until a fixed point is eventually reached. The process is guaranteed to terminate, as we can assume that all tasks have a

finite deadline. This limitation is consistent with the idea of using task models as a way to represent slices of a system's behavior. It follows that there is a finite number of time slots at which the algorithm should attempt to schedule a task.

The following pseudo-code describes our algorithm:

1: $S \leftarrow enc(\{[c \mapsto 0 \mid c \in C]\})$
2: $S' \leftarrow \bot$
3: **while** $S \not\equiv S'$ **do**
4: $\quad S' \leftarrow S$
5: $\quad$ **for all** $\langle t, c\rangle \in T \times C$ **do**
6: $\quad\quad$ **for all** $t_r \leq i \leq t_d - t_c$ **do**
7: $\quad\quad\quad S \leftarrow S \cup (sch(t, c, i) \circ \mathit{fltr}(t))(S)$
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end while**
11: $S \leftarrow check(S)$

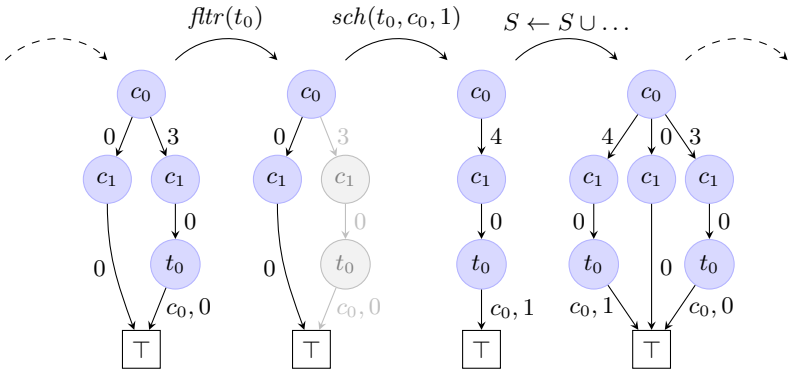


Fig. 6: One iteration of the scheduling construction algorithm's the inner loop

Line 1 creates a MFDD representing a singleton set which contains a function that maps all cores to 0 (i.e., their next available time). Then, most of the work is carried out at line 7, where a task's mapping (on a given core, at a specific time) is added to all schedulings. This process is illustrated for one single iteration in Figure 6. Assume S to be the MFDD depicted on the left, which encodes the empty scheduling, as well as one mapping in which a task t_0 is being scheduled on core c_0. The first step consists of filtering out the mappings for which the task is already defined, with the *fltr* homomorphism, producing the MFDD in which the corresponding paths have been grayed out. The remaining mappings are then fed into the *sch* morphism, that actually inserts the task at the correct position in all paths. Finally, we compute the union of the resulting MFDD with S, effectively merging all new schedulings into the previous ones.

Note that task dependencies are not taken into consideration until line 11, where a third morphism *check* removes all non-feasible schedulings. This final step boils down to another filter that removes the paths that do not schedule all tasks, as well as those in which task constraints are not satisfied.

6 Experimental Results

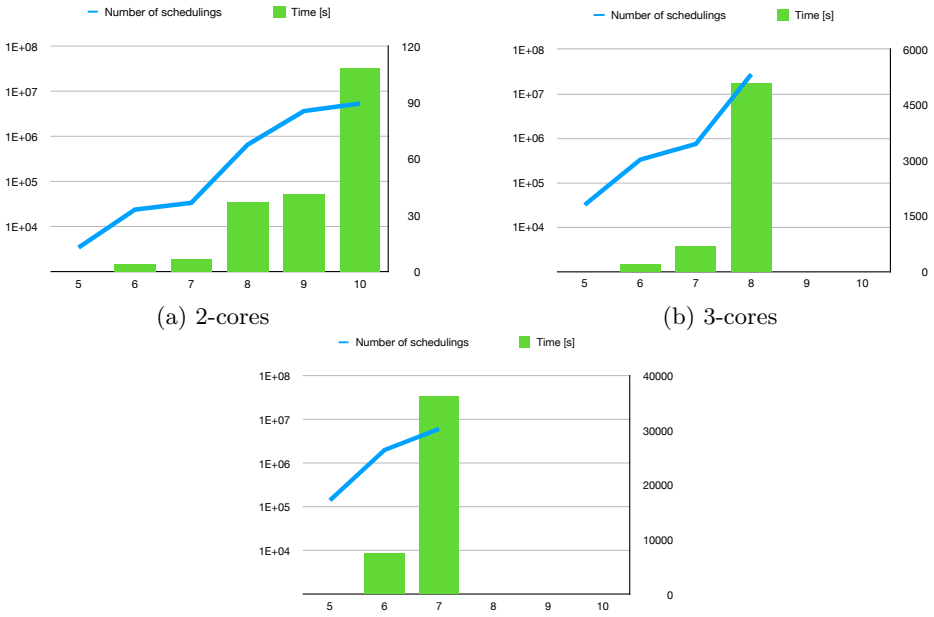
We implemented our schedulability analysis technique with DDKit, a library to manipulate MFDDs. We tested it on several randomly generated task models of various sizes, for a 2-cores, a 3-cores and a 4-cores architecture. Our implementation closely follows the algorithm presented in Section 5.2. All tests were ran on a intel i9 at 2.3 GHz. Sources as well as the randomly generated datasets we used are available on GitHub[2], and distributed under the MIT license.

Test runs showed that our technique can quickly determine if a given model is not schedulable due to timing constraints alone. This is because schedulings are built incrementally, by adding a new unscheduled task at each pass. Paths in which a task is not schedulable due to timing constraints (i.e. its deadline cannot be met) are cut from the MFDD as soon as they are detected, actually reducing its size. As a result, subgraphs that were only reachable from such paths need no longer to be explored for scheduling the remaining tasks. This is akin to pruning in a classical backtracking algorithm. Filtering out dependency constraints is also efficient, and amounts to about only 5% of the total execution time. This can be explained by our encoding strategy. As dependent tasks tend to appear closer to the MFDD's root, filtering out a path in which its dependency is not scheduled removes a lot of subgraphs at once, and with them, a lot of possible solutions that no longer need to be explored. Furthermore, this mechanism deals gracefully with chains of dependencies.

Results are summarized in Figure 7. As tasksets are generated randomly, computation times may vary from one task model to the other. Hence, we averaged all results on three different test cases, for each size of task model. We also removed models for which no schedulings can be found, as those are significantly faster to process. We measured the running time as well as the total memory consumed throughout the execution. Runs whose computation exceeded twelve hours were aborted, explaining missing results for both the 3-cores and 4-cores architectures.

As we can see, our algorithm scales much better with the number of tasks than with the number of cores. Although it took on average 4 minutes and a half to compute 150'000 solutions for a model consisting of 5 tasks on a 4-cores architecture, we were able to compute nearly 40 times more schedulings (i.e., roughly 5.5 millions) for a model of 10 tasks on a 2 core-architecture. This asymmetry can be explained by the way operations are optimized on decision diagrams. Recall that outgoing arcs of nodes that represent tasks are labeled with a pair denoting a core and a time. Increasing the number of cores increases

[2] https://github.com/kyouko-taiga/Schedulability



(a) 2-cores

(b) 3-cores

(c) 4-cores

Fig. 7: Experimental results

the size of the domain from which these labels will be picked, which leads to reducing the possibility of sharing identical subgraphs. As a result, applying a homomorphism on the diagram is more expensive, as caching gets less efficient.

This scaling problem could be tackled by the means of *anonymization* [9]. Anonymization is an optimization technique for decision diagrams which aims to reduce the properties that make two configurations distinguishable, so as to leverage caching more aggressively. In our case, the specific core on which a task is scheduled could be forgotten, so that arcs would be labeled only by the start time. Obviously, this loss of information would make the encoded schedulings less precise, as we could only know *when* a task is scheduled, and not *where*. However, this would be sufficient to determine the number of solutions (if any) there exists to schedule a task model on a given architecture.

Our approach is a refinement of another similar work [17], which purposes the use of Data Decision Diagrams for computing sets of schedulings. Our technique brings a number of improvements. Firstly, we are able to compute all possible schedulings for a given time window, including those in which some cores may be idling (i.e., inactive in a period of time), which was suggested as future work in [17]. Hence, we are able to check for schedulability in the presence of transient errors, whereas the original method can only model permanent core failures. This refinement also solves an issue related to the handling of dependencies. Since their technique is not able to model idling, it is not able to schedule a

task *after* its dependencies have been executed, on any core whose next available time precedes the completion of the dependency. Secondly, experimental results show that our approach is roughly twice as efficient in terms of computation time. We explain this disparity by the fact that we filter for dependency inconsistencies *after* having built the set of all possible schedulings, rather than after the addition of a new task. This alleviates the filtering effort and maintains a smaller MFDD during the task addition phase.

7 Conclusion

We presented a transformation of the multi-core schedulability problem as a state space exploration problem, in a style reminiscent to model checking. We showed how to build the set of all possible schedulings of a given taskset with Map-Family Decision Diagrams, a variant of decision diagrams that we designed to encode large sets of partial functions into a compact representation. We used homomoprhic operations, in particular filters, to manipulate the set of schedulings and illustrated how to use these operations to analyze solution sets.

We envision future works along two main axes. The first one is to refine our encoding, so as to improve on the performance of our algorithm. A promising lead in this direction is to find better heuristic for task ordering, to maximize sharing. Another idea would be to exploit symmetries between schedulings, so as to prune the state space exploration. Finally, as mentioned in Section 6, anonymization could also be leveraged for specific analysis, for instance, to more quickly identify non-schedulable models [9]. The second axis for futre works relates to the development of a framework to analyze scheduling sets. We already demonstrated that filtering homomorphisms can be efficiently leveraged to exclude schedulings based on dependency constraints. Other kind of constraints could be translated into filters as well, and applied on solution sets to check schedulability under more elaborate constraints. For instance, one could exclude schedulings in which two given tasks are not executed at the same time, on different cores, so as to account for possible communications.

References

1. Baro, J., Boniol, F., Cordovilla, M., Noulard, E., Pagetti, C.: Off-line (optimal) multiprocessor scheduling of dependent periodic tasks. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. pp. 1815–1820 (2012)
2. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Perform. Evaluation Rev. **32**(4), 34–40 (2005)
3. Bergman, D., Ciré, A.A., van Hoeve, W., Hooker, J.N.: Decision Diagrams for Optimization. Artificial Intelligence: Foundations, Theory, and Algorithms, Springer, Berlin (2016)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986)
5. Cire, A.A., van Hoeve, W.J.: Multivalued decision diagrams for sequencing problems. Operations Research **61**(6), 1411–1428 (2013)

6. Couvreur, J., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.: Data decision diagrams for petri net analysis. In: Esparza, J., Lakos, C. (eds.) Applications and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 2360, pp. 101–120. Springer (2002)
7. David, A., Larsen, K.G., Legay, A., Mikucionis, M.: Schedulability of herschel revisited using statistical model checking. International Journal on Software Tools for Technology Transfer **17**(2), 187–199 (2015)
8. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G.: Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In: IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems. pp. 263–272. Springer (2007)
9. Hong, S., Kordon, F., Paviot-Adet, E., Evangelista, S.: Computing a hierarchical static order for decision diagram-based representation from P/T nets. Transactions on Petri Nets and Other Models of Concurrency **5**, 121–140 (2012)
10. Jensen, A.R., Lauritzen, L.B., Laursen, O.: Optimal task graph scheduling with binary decision diagrams (2004)
11. Korousic-Seljak, B.: Task scheduling policies for real-time systems. Microprocessors and Microsystems **18**(9), 501–511 (1994)
12. Lehoczky, J.P., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In: Real-Time Systems Symposium. pp. 166–171. IEEE Computer Society (1989)
13. Linard, A., Paviot-Adet, E., Kordon, F., Buchs, D., Charron, S.: polydd: Towards a framework generalizing decision diagrams. In: Gomes, L., Khomenko, V., Fernandes, J.M. (eds.) International Conference on Application of Concurrency to System Design. pp. 124–133. IEEE Computer Society, New York (2010)
14. Mahadevan, S., Storgaard, M., Madsen, J., Virk, K.: ARTS: A system-level framework for modeling mpsoc components and analysis of their causality. In: International Symposium on Modeling. pp. 480–483. IEEE Computer Society (2005)
15. Nguyen, V.A., Hardy, D., Puaut, I.: Cache-conscious offline real-time task scheduling for multi-core processors. In: 29th Euromicro conference on real-time systems (ECRTS 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
16. Puffitsch, W., Noulard, E., Pagetti, C.: Off-line mapping of multi-rate dependent task sets to many-core platforms. Real-Time Systems **51**(5), 526–565 (2015)
17. Racordon, D., Buchs, D.: Verifying multi-core schedulability with data decision diagrams. In: Crnkovic, I., Troubitsyna, E. (eds.) International Workshop on Software Engineering for Resilient Systems. Lecture Notes in Computer Science, vol. 9823, pp. 45–61. Springer, Berlin (2016)
18. Yalcinkaya, B., Nasri, M., Brandenburg, B.B.: An exact schedulability test for non-preemptive self-suspending real-time tasks. In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe. pp. 1228–1233. IEEE (2019)