Direct Handling of Infeasible Paths in the Event Dependency Analysis

Kilian Kempf and Frank Slomka
Institute of Embedded Systems/Real-Time Systems
Ulm University, Germany
{kilian.kempf,frank.slomka}@uni-ulm.de

***Abstract*—While common task models in real-time analysis assume a task as being activated by a single event and producing a single outgoing event after its full completion, the event dependency analysis extended the model to allow for multiple outgoing events to occur already during the run-time of a single job. It uses the structure of a timing-annotated control flow graph to calculate the maximal densities of outgoing events. Previous work has shown that knowledge about infeasible paths in the control flow graph can be used to relax the event densities by modifying the graph prior to performing the event dependency analysis. However, that approach leads to a nearly explicit path enumeration. In this paper, we show how the event dependency analysis can be modified to directly exclude infeasible paths in order to relax the densities of outgoing events.**

I. Introduction

The integration of functions into a system usually involves mapping those functions onto software tasks and ensuring their inter-communication to allow for data exchange. The common models allow for activation of the tasks either by (periodic) timers or more importantly by external events emanating from the context the system is embedded in. During their computation, tasks can generate data that other tasks use as input. This dependency is predominantly modeled as events that one task can generate to activate another. The result is a dependency graph of tasks we call the task graph of a system. In a distributed system, this task graph stretches across multiple resources, including communication resources.

The common task models consider a task as a unit that is activated by an event, spawning a new job that can be scheduled for execution and interrupted and therefore delayed, but that creates a single event at the time of its full completion which then activates one or more other tasks. Quite a while ago, Bodmann et al. have proposed an addition called event dependency analysis [1] that allows for multiple outgoing events to be created during the run-time of a job. They consider the internal structure of a task in order to know when an event can be generated and are able to calculate the worst-case density of outgoing events that results from a density of incoming events activating the task. An extension of a control flow graph (CFG) is used to model the internal structure of a task. Bodmann et al. extended the notion of basic blocks by allowing a block to generate an event, calling the resulting structure an event flow graph. As the run-time of each basic block can be obtained from worst-case execution time (WCET) analysis (or more accurately best-case execution time in this case), the event dependency analysis is able to calculate the smallest time interval in which a certain number of events may be generated. This time interval may stretch across multiple subsequent activations of a task.

The event dependency analysis as proposed by Bodmann et al. always tries to find the highest density of any number of events in any part of a task's control flow. This is conceptually not too far from the implicit path enumeration technique (IPET) that is used in the WCET analysis, although in this case the origin of the implicit enumeration is different. Unfortunately, it leads to the inclusion of all possible paths through the control flow graph, even those that will never be executed due to data dependencies and mutual exclusion of sub-paths, a concept commonly referred to as infeasible paths. In order to mitigate the effect of infeasible paths on the event dependency analysis, Kempf et al. have proposed a work flow [2] that leverages the progress made in WCET analysis in order to identify infeasible paths and exclude them from the event dependency analysis. They present a transformation of the event flow graph and show how a WECT analysis tool can be used to strip any infeasible paths from the graph. It then can be handled by the unmodified event dependency analysis.

The approach Kempf et al. have taken demonstrated how the consideration of infeasible paths can lead to a relaxation of event densities without modifying the original event dependency analysis. Unfortunately, this comes with a price: In order to strip infeasible paths from the control flow, the proposed transformation steps lead to a nearly full *explicit* path enumeration. While this is sufficient for a proof of concept showing the influence of infeasible paths on event densities, it leads to significant increase of run-time and memory complexity. We feel that a more direct inclusion of the knowledge about infeasible path in a task is necessary to achieve a practicable analysis.

Similar to the approach in [2], we use the flow facts generated by a WCET analysis tool that handles the identification of infeasible paths. In fact, all the necessary information is contained in the flow facts. We modify and extend the event dependency analysis of Bodmann et al. to incorporate this information and directly exclude infeasible paths from the analysis.

II. Related work

There is a long list of contributions in real-time analysis, most of which can be classified into one of two levels of a system design: the (higher) system level and the (lower) task level. On the system level, the response-time analysis takes place which considers task activation and interdependency,

scheduling algorithms, and often communication resource arbitration in the case of a distributed system. All that the analysis on this level needs to know about a task is its best-case and worst-case execution time, its activation patterns, and a relative deadline, if applicable. External activation patterns and in large part the deadlines originate from the system's context. The execution times are estimated on the task level, which is the domain of the WCET analysis. Here, the control flow inside a task, data dependencies and a bulk of hardware dependent properties have to be taken into consideration, for example caches and pipelines in the case of a processing resource.

Examples for system level analyses go back as far as Tindell and Clark [3], newer approaches that are better at handling complex distributed systems include the SymTA/S approach [4] and the Real-Time Calculus [5], which both have been continuously improved and extended.

The WCET analysis is an entirely different field of research and has evolved rapidly since the implicit task enumeration technique [6] was conceived. Current methods are able to consider infeasible paths, e. g. [7] and [8].

Unfortunately, the two levels are generally not connected, more specifically the structure and execution of a task does not directly influence the resulting events. As described in the introduction, Bodmann et al. have developed the event dependency analysis [1] in order to allow the generation of multiple events by a single job during its run-time. This has been revisited by Kempf et al. [2] who use information from the task level, in this case the flow facts of the WCET analysis, to relax the event densities on the system level.

There are two other approaches that may seem related to the idea of multiple outgoing events emanating from a single task. One is the family of multi-frame models, the most general currently being the extended digraph real-time task model [9] by Stigge et al. This model allows for different types of jobs for a task, which all may have different execution times, deadlines and minimal separation times. While this can provide for different amounts of computation time from the activation of a task until its release of an outgoing event, it cannot handle arbitrary activation patterns and the emulation of multiple outgoing events at the same time. More importantly, a great part of the event dependency analysis is the extraction of event densities from the control flow inside a task, which has no analogy to the idea behind the multi-frame models.

The other possible relation might be to the superblock concept as used by Pellizzoni et al. [10]. In order to model the interaction with a shared resource during the run-time of a task, they structure the control flow inside a task into several sequential blocks of basis blocks. These superblocks are divided into phases of communication and computation, so that they interact with the system only during the first and the final phase. The approach was originally conceived to analyze the access on a shared memory but has since then been extended to shared communication resources.

III. Model

This section presents the model we use for the tasks themselves. The system level model, i. e. task graph, deadlines, etc. adheres to the common conventions. The following definitions are taken from [2].

The structure of tasks is modeled by a control flow graph, which is a directed possibly cyclic block-graph. The vertices are the blocks while the edges form a possible path for the flow of control.

Definition 1. Basic block: *A basic block (BB) is a sequence of instructions in a task that can be processed unconditionally, i. e. it contains no conditional jumps. The whole basic block can be the target of a jump. At the end of a basic block there may be a conditional jump to another basic block. All blocks are annotated with a minimal and maximal execution time which denotes the range of times needed for the uninterrupted execution of the block on a specific resource.*

For this paper, only the best-case execution times are needed and annotated, as the event dependency in its current state calculates the maximal event densities.

Definition 2. Control flow graph*: A control flow graph (CFG) is a directed graph $G = (V, E, s, x)$, where V is a set of basic blocks and $E \subseteq V \times V$ the set of edges between them that model the transfer of control. $s \in V$ denotes the single start node ($\forall v \in V : (v, s) \notin E$) and $x \in V$ denotes the single exit node ($\forall v \in V : (x, v) \notin E$). We demand that for every block $v \in V$ there are at most two edges to other blocks:*

$$\forall v \in V, O = \{v\} \times V : |E \cap O| \leq 2$$

Definition 3. Predecessors and successors: *For each node $v \in V$ of a CFG $G = (V, E, s, x)$ there is a set of predecessors pred(v) and successors succ(v):*

$$k \in \textit{pred}(v) \quad \Leftrightarrow \quad (k, v) \in E$$
$$k \in \textit{succ}(v) \quad \Leftrightarrow \quad (v, k) \in E$$

The control flow during the execution of a task is modeled as a path through the CFG of that task that starts at the start block and ends at the exit block.

Definition 4. Path: *A path π through a CFG $G = (V, E, s, x)$ is a sequence of nodes from the start block to the exit block $\pi = (s, v_1, ..., v_n, x)$ with $s, v_{1..n}, x \in V$ and $(s, v_1), (v_1, v_2), ..., (v_n, x) \in E$.*

Definition 5. Dominance: *Let $G = (V, E, s, x)$ be a CFG. A node $v \in V$ of that graph is dominated by a node $k \in V$ when every path from the start node s to v passes through k, that is if all of those paths are of the form $(s, ..., k, ..., v)$. We then write k dom v. Every node dominates itself: $\forall v \in V : v$ dom v.*

IV. Event dependency analysis

This section will explain the event dependency analysis as far as necessary for this paper. Nevertheless, readers are encouraged to study the original publication [1] in order to gain a deeper understanding of the context. Additionally, in the paper at hand we will leave out the exact definitions of some elemental functions, as they are transparent to our extension.

A. General idea

The event dependency analysis as originally conceived by Bodmann et al. in [1] establishes an extension of the traditional task model. Instead of a task being activated by a single event and then releasing a single outgoing event at the end of its

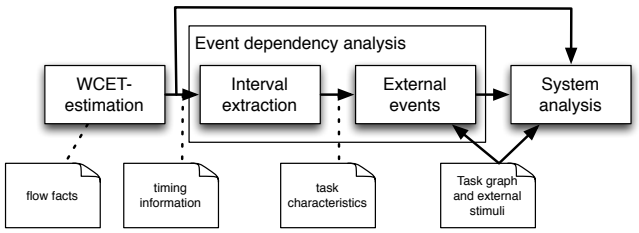

Fig. 1. Workflow of the real-time analysis

computation, the event dependency analysis allows for multiple outgoing events of the same type to be released during the run-time of a singe job. As with the traditional model, the outgoing events activate other tasks according to the system's task graph. Of course, the density or minimal distance between outgoing events is of major concern for the feasibility of the schedule. The event dependency therefore analyzes the control flow inside the tasks in order to determine where exactly the outgoing events are generated and what the minimal distance between multiple events may be. In [1], Bodmann et al. presented an algorithm that traverses the control flow graph of a task, visiting every node only once, and computes the maximal density of outgoing events in the form of the common interval domain.

The general workflow in which the analysis is embedded is depicted in Figure 1. The event dependency analysis has a control flow graph with annotated timings generated by a WCET estimation tool as an input. The analysis itself consists of two steps. The first one operates on single tasks and extracts the timing characteristics of each task. It determines the interval functions that describe the maximal density of outgoing events as a response to a single activation. A second step then considers multiple activations of a task in the form of incoming event densities. The propagation of event densities throughout the system can be calculated by using knowledge about the task dependencies obtained from the task graph. Combining this with the external stimuli originating from the system's context, the activation patterns for each task can be obtained. This is then used as an input to the real-time analysis of the system level, which analyzes scheduling feasibility, response times, resource utilization, etc.

B. Timing characteristics

One of the main concepts is the extension of a basic block to an event block. Please note that the use of basic blocks does not necessarily imply a granularity level equal to the actual machine code after compilation and various optimizations like loop unrolling, function inlining, or emulating a hardware-dependent function like division. The notion of basic blocks is perfectly possible on any level, and there are methods to back-annotate the execution times determined on a lower level back to the higher levels.

The introduction of event blocks allows us to model the events at the times they actually occur. Event blocks are basic blocks that generate an event at the end of their execution. The event blocks trigger task-global events, which means that all subsequent dependent tasks are activated by the event.

Definition 6. Event block: *An event block v is a basic block inside a task τ_i for which event(v) = true, meaning that it generates an event at the end of its execution which activates another task τ_j.*

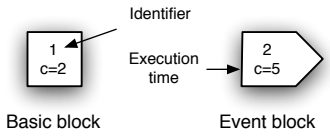

Fig. 2. Graphical representation of basic block and event block

The graphical representation for an event block used by Bodmann et al. in [1] is a block arrow that points to the right and resembles the output symbol of the Specification and Description Language (SDL). Figure 2 depicts such an event block along a normal basic block.

Definition 7. Event flow graph: *An event flow graph (EFG) is a modified control flow graph $G = (V, E, s, x)$ with $\exists v \in V$: event(v) = true.*

In order to determine the minimal intervals for a number of events, the structure of the control flow graph has to be taken into consideration. Bodmann et al. used an incremental representation of the graph, consisting of the basic functions cat for concatenation of sub-graphs, and mrg, denoting the merge of the control flow of sub-graphs. These two building blocks may then be used to describe any control flow graph. Please note that the event dependency analysis does not actually reconstruct the graph for the analysis. Those two functions are used to generalize the necessary definitions and to convey the concepts. Nevertheless, for the traversal of a flow graph one should think of cat as adding another node to the already handled portion, while mrg means that the corresponding node effectively merges possible paths through the graph, which means it is the target of a number of jump instructions.

Bodmann et al. introduced several interval functions, mapping numbers of events to intervals. Fortunately, for our approach to work, the original functions do not have to be

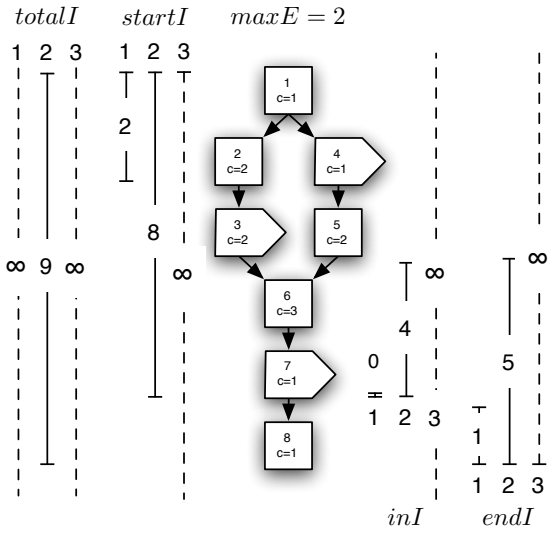

Fig. 3. Elementary interval functions in an event flow graph

changed except for a minimal formal extension. Nevertheless, we feel that a good understanding of the concept behind the interval functions is required in order to follow our contribution. The definitions given in [1] make use of an iterative construction of the control flow graph using the elemental functions cat and mrg. We will abstract from those formal definitions by giving the appropriate informal explanation. An example of an event flow graph with annotated intervals is depicted in Figure 3.

The most important interval function is the one describing the intervals of events inside a task. It is called inI_n and returns the minimal distance between n outgoing events on any path through the control flow graph. The corresponding interval may reside anywhere inside the task. Incidentally, also $inI_1 = 0$, as the minimal interval in which one event can occur is of infinitesimal length. In principle, this is the only function concerning the system analysis, as it describes for any activation of a task the worst case densities of the outgoing event stream of a task, which in turn may activate other dependent tasks. Unfortunately, this is not sufficient when considering multiple activations of the task, i. e. multiple jobs, in which case the minimal intervals of a greater amount of events than one job may generate need to be calculated. Three other interval functions are used during the event dependency analysis. They help for the consideration of multiple jobs but are already necessary for the calculation of inI_n.

Instead of allowing the corresponding interval to reside anywhere inside the task's control flow graph, function $startI_n$ demands that the interval is bound to the start node and stretches inside the flow graph while spanning n outgoing events. In other words, it describes the minimal interval from the start of the CFG to the n-th event. Interval function $endI_n$ denotes the analogon for intervals bound to the end node of the CFG, stretching backwards into the flow graph. The last of the interval functions is $totalI_n$, denoting the minimal amount of time that is necessary for traversing the the control flow graph from start to end while generating exactly n events. If there is no path through the task that generates the exact amount of requested events, the interval returned is infinite. Another function denotes the maximal number of of outgoing events that a single job may generate. $maxE$ returns the maximal amount of outgoing events that can be generated by any path through the control flow graph.

C. Analysis workflow

For the actual analysis, a stack-based algorithm is used to traverse the control flow graph and calculate the interval functions for every node. For any node, its functions denote the minimal intervals for all possible paths though the flow graph up until the node. This subgraph then is incrementally extended until all nodes and therefore all possible paths have been accounted for. Different situations may be encountered during the traversal:

1) The currently analyzed node has only a single predecessor. In this case, the intervals valid for the preceding node need to be updated and extended to include the current node. The definitions of the interval functions as given in [1] make use of the cat function in this case. A $cat(G, k)$ denotes that the preceding subgraph G (including all paths from the start node to the current node) is concatenated with the current node k.

2) The current node has multiple predecessors, which equates to a merge of control flows. Each preceding node and their corresponding subgraphs allow for multiple possible paths, which then all culminate in the currently analyzed node. Concerning the interval functions, this is originally defined as $cat(mrg(G, H), k)$ with G and H being the subgraphs corresponding to each of two preceding nodes and k being the current node. Such a merge of control flows is commonly found in the domain of WCET analysis, for example in value and cache analysis. In order to calculate the minimal intervals for the occurrence of events, the subgraphs first merged by keeping the minimum of the intervals for every number of events. This is where we need a little extension to the original definitions in order to allow for more than two predecessors for a node. Of course, an n-way merge can be represented as $n-1$ binary merges, but as all interval functions simply return the minimum for any number of events, we redefine

$$f_i(mrg(N, ..., M)) = min\{f_i(N), ..., f_i(M)\}$$

for $f_i \in \{totalI_i, startI_i, endI_i, inI_i\}$. The function $maxE$ is accordingly redefined as

$$maxE(N, ..., M) = max\{maxE(N), ..., maxE(M)\}.$$

For a merging node to be analyzed, all of its predecessors must have been handled. This is easily accomplished by the stack-based nature of the approach. After a node has been processed, all its successor nodes are pushed onto the stack. In case a merging node has unhandled predecessors, it is skipped an will automatically be pushed on the stack by the unhandled preceding nodes. This corresponds to a depth-first traversal that is bounded at the merge points.

After the control flow graph of a task has been processed, all the interval functions are known, representing for any number of outgoing events the minimal intervals in which they can occur. The functions are bounded by $maxE$, which denotes the maximal amount of events that a single activation of a task may generate.

As mentioned before, the consideration of multiple jobs requires additional steps and functions. For any task activated more than once, more than $maxE$ outgoing events have to be accounted for. However, this affects the scheduling and other aspects on the system level. The temporal characteristics of any task are fully defined by the functions presented up to this point. They can be regarded as part of the interface between the task level and the system level. All work following in this paper aims at a relaxation of those (interval) functions. Higher levels of the event dependency analysis do not need to be changed in order for our approach to work.

V. Infeasible paths and their detection

The original event dependency analysis as reproduced in the previous section calculates the minimal intervals in which a number of events can occur, while considering every path through the control flow graph. However, during the execution of a task, not all paths may actually be taken.

A. General idea

As the conditional branching relies on predicate expressions, it may be subject to data dependencies. This may result

in certain paths never being executed. Not only the input values of a task are responsible for this, but also static properties like the combination of conditions. Together with the structure of the control flow graph this can result in paths that may not be taken for any instance of the task. Those paths are commonly called infeasible paths. They may for example exist if a condition of a conditional branch is re-evaluated at a later point along the path. More formally in [2]:

Definition 8. Infeasible path:
A path $\pi = (v_1, ..., v_n, c, b, w_1, ..., w_m)$ through a control flow graph is said to be infeasible if c is a control block containing the predicate expression A which always evaluates to the same logical value when reached through the sub-path $\pi' = (v_1, ..., v_n, c)$ while $b \in succ(c)$ is the node following the control block that would be taken if A evaluated to the opposite value.

Clearly, there may be multiple infeasible paths originating from the same condition at a control block. In fact, if the conditions resulting in infeasibility reside anywhere inside the control flow graph, all paths that have those conditions as a sub-path are also infeasible. In order to pinpoint the origin of all those infeasible paths we adopt the concept of a shortest infeasible path from Bodík et al. [11]:

Definition 9. Shortest infeasible path: *An infeasible path $\pi = (v_1, v_2..., v_n)$ is a shortest infeasible path if both sub-paths $(v_2, ..., v_n)$ and $(v_1, ..., v_{n-1})$ are feasible.*

That way, the first and last node contributing to the existence of any infeasible path are known. Note that the shortest infeasible path may be of any length: If there is a branch that may never be taken due to its conditional expression, the shortest infeasible path consists of only a single node, namely the first node of the branch. We will revisit different types and origins of infeasibility in the next subsection.

Groups working on WCET analysis have presented several methods that can detect and handle infeasible paths, as this is necessary for tight execution time estimates. Stein and Martin have shown their approach in [8] which has been extended in [12]. A different method has been shown by Suhendra et al. in [13]. Another group working on WCET analysis is the one from Mälardalen University responsible for the analysis tool SWEET. They have presented their approach to infeasible paths in [7]. They use a variant of symbolic execution and are able to detect three different origins of infeasible paths. For our work, we chose the flow facts generated by SWEET as the basis. A more detailed description of the interface follows in the next subsection.

B. Infeasible paths in SWEET

The SWEdish Execution Time analysis tool SWEET uses abstract execution, which is a kind of symbolic execution, for their flow analysis. For our work, we use the flow facts generated by SWEET as an input. Therefore, we will now show the different types of infeasible paths that SWEET is able to detect.

SWEET distinguishes between three types or origins of infeasibility: infeasible *nodes*, infeasible *pairs*, and infeasible *paths*. The algorithms used to find those origins are presented in [7]. It further differentiates between the *scopes* in which such infeasibility may occur. An example for a scope is a function called with specific parameters, or the inside of a loop. In our current approach, we will restrict ourselves to infeasible paths that are always present. The handling of different scopes is beyond the aim of this paper. Figure 4 shows an example of control flow graphs with an infeasible node, an infeasible pair and an infeasible path.

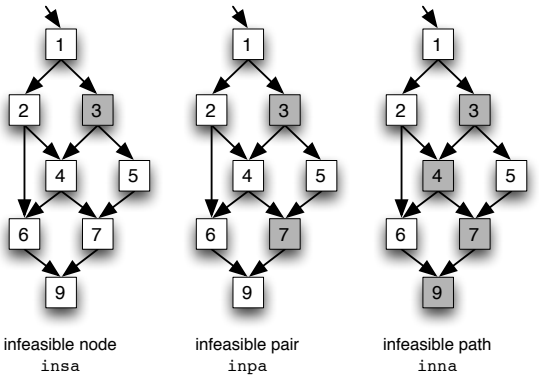

Fig. 4. Example of an infeasible node, pair and path

Listing 1. Shortened example of flow facts generated by SWEET

```
expint::if.then15 = 0 ;    %% insa
expint::if.then + expint::if.end54 < 2 ;    %% inpa
expint::for.end + expint::if.end54 < 2 ;    %% inpa
```

Nodes: An infeasible node is a node that may never be executed in a certain scope. Therefore, its shortest infeasible path is just the node itself. This is usually a node directly after a branching condition which will always be evaluated the same way. In SWEET's nomenclature, this is denoted as `insa`. There is no path $\pi = (v_1, ..., a, ..., v_n)$ with a being an infeasible node.

Pairs: An infeasible pair is a pair of nodes in the control flow graph that may never be executed together, so there is no path $\pi = (v_1, ..., a, ..., b, ..., v_n)$ with (a, b) being an infeasible pair. The corresponding nomenclature is `inpa`.

Paths: An infeasible path in SWEET is a tuple of nodes that may never be executed together. There is no feasible path $\pi = (v_1, ..., a, b, c, d, ..., v_n)$ with $I = (a, b, c, d)$ being an infeasible (sub-)path. In SWEET's nomenclature this is denoted as `inna`.

Listing 1 shows shortened typical examples of different types of infeasibility in the form of flow facts as generated by SWEET, for which the scope has been omitted.

VI. Handling infeasibility

In this section, we will show how our approach handles the different types of infeasibility. Please note that this is in no way a strict dependency on SWEET, as those types represent a generalization of the origins of infeasible paths and therefore can also be detected using other methods, algorithms, or tools.

A. Modifications of the analysis

For our approach we extend the original event dependency analysis to be able to cope with infeasibility in the control flow graph in order to relax the densities of outgoing events. As shown before, the original analysis consists of two steps, the first being the extraction of a task's timing characteristics so that its response to a single activation is known. In a second step, these characteristics are then used to calculate the propagation of maximal event densities through the system. At that point the task dependencies are considered.

Our extension only concerns the first step of the event dependency analysis. The influence of infeasible paths will be entirely reflected by changes to the timing characteristics. In particular, all it will effectively do is to relax the density of events in the interval functions and reduce $maxE$ of a task. This is completely transparent to the propagation step of the event dependency analysis which we will leave unchanged. As a result, the increase of computational complexity implicated by our approach is linear with the number of tasks in a system. Just as before, every task has to be analyzed only once. As the event dependency in its current form does not internally handle loops in the control flow, we restrict ourselves to the same for this paper.

The modifications to the interval functions themselves as defined in [1] will be negligible. We only allow for the merging of more that two predecessors of a node, as already shown in Section IV. The main concept is the preserving of information across merge points. We use annotations of interval functions to each node that may be part of a shortest infeasible path.

B. Infeasible nodes

Infeasible nodes allow to be handled rather straightforwardly in theory. As they may never be reached during the task's execution, they can be ignored for the event dependency analysis. This includes all successors that can only be reached on a path through the infeasible node, i. e. that are dominated by such node. An imaginable approach hence would be to prune the control flow graph by removing all nodes $\{v \mid i_{start} \text{ dom } v \wedge i \text{ is of type } \texttt{insa}\}$. This however would require an additional preprocessing step. As we aim to keep the general idea and structure of the event dependency analysis algorithm, we reason it would be better to integrate the handling of infeasible nodes into the traversal of the graph.

Traversal: When analyzing an infeasible node, discard all timing information for this node and mark it as dominated by a node of type `insa`.

Merging: During the merge, check whether the predecessors are infeasible nodes or are dominated by one. If they all are, the current one is also dominated and therefore marked and ignored, i. e. not analyzed. Otherwise proceed as normal. In that case, assume all intervals of the `insa`-dominated predecessors to be ∞ and $maxE$ to be 0. This ensures that all merge functions ignore the predecessor.

This works well with multiple infeasible nodes, even when the resulting infeasible paths are possibly overlapping. As all the infeasible nodes as well as all the nodes they dominate may never be taken, there is no need to distinguish between the different origins of domination. As long as all predecessors of a node are dominated by some node of the type `insa`, the node itself may never be reached. Thus we need only a single type of marking we call *insadom*.

C. Infeasible Pairs

The handling of infeasible pairs is completely different and way more complicated. The most challenging aspect is that the original event dependency analysis uses merging to bound the combinational complexity. Every time two or more control flows are merged, only the information about the highest possible density of events is kept, all other information is discarded. So, after a merge point, the origin of the timing information is not determinable any more. When reaching the second node of an infeasible pair, which is exactly the end of the corresponding shortest infeasible path, there is no way to determine whether the intervals contain an infeasible path and certainly no possibility to account for the infeasibility, as all other timing information has been lost.

In order to be able to calculate the maximal possible density of events at the end of an infeasible path, the information about the influence that any nodes residing on an infeasible path have on the intervals has to be preserved across the merge points. If one now considers that there are many possible paths through the control flow graph that may "cross" an infeasible path and on top of that there may be multiple infeasible paths crossing and partially overlapping each other, and all that information has to be preserved, one has to face a problem of exponential complexity.

In principle, all we need to do is to keep more than one set of the interval functions and $maxE$ for every node we visit during the traversal. The additional sets of functions store the minimal intervals and the maximal number of events for any paths excluding the ones through the nodes of an infeasible pair. Unfortunately, we cannot simply keep and propagate only the intervals that exclude the infeasible pairs. There may be a path that crosses nodes on the infeasible path, forming coincident sub-paths, but that itself is feasible and therefore has to be taken into consideration. At the time of the merging points along those sub-paths the impact of either path on the final minimal event density is undecidable.

Therefore, the functions for every infeasible pair, and even worse also for the power set of all infeasible pairs, have to be recalculated and stored for every node. Depending on the amount of infeasible pairs in the flow graph this may be worse than an *explicit* path enumeration.

In order to significantly reduce the complexity, our approach is to only calculate and store the functions along the shortest infeasible paths. This reduces the complexity for every node to the power set of all the shortest infeasible paths it is part of. We will see that for all nodes dominated by the start node of an infeasible pair the calculations of the corresponding functions may also be skipped.

Let us first try to convey the rough concept. We extend the traversal of the original event dependency analysis by annotating more than one set of functions to each node. This additional information begins its life whenever the start of a shortest infeasible path is encountered. The additional information is preserved across merges, but only as long as

necessary. It is no longer needed when the traversal of a shortest infeasible path ends, which happens when either the path is left or when its end is reached. The rest of this subsection will explain the necessary steps in detail and show how to handle the complexity of multiple infeasible paths "crossing" each other.

For the following definitions, we need the concept of reachability:

Definition 10. Reachability: *v reach k := $\exists \pi$ in the CFG* : $\pi = (v, ..., k)$.

The shortest infeasible path corresponding to an infeasible pair is not necessarily a single one but may be a set of paths that is in the transitive closure between the start node and the end note of the pair, consisting of all nodes that can both be reached by the start node and are able to reach the end node. Formally: Node k is on the shortest infeasible path of an infeasible pair i iff $(i_{start}$ reach $k) \wedge (k$ reach $i_{end})$.

In order to know when we are leaving an infeasible path, i. e. when there is no path from the current node to the end node of an infeasible pair, we need to pre-calculate the reachability for each of those nodes. This can be done using the Warshall-part of the Floyd-Warshall algorithm, which calculates the transitive closure and therefore reachability for the whole graph. If the total number of infeasible pairs is relatively small, other approaches like a bounded search might be used improve the performance.

For every node, we need to respect the power set of all the shortest infeasible paths caused by infeasible pairs it may be a part of. We do this by annotating multiple instances of the interval functions and $maxE$.

Annotations: In the original event dependency analysis, the functions $maxE$, $totalI$, $startI$, $endI$, and inI are calculated for the sub-graph of a flow graph leading to a node. Based on how the traversal of the graph works, this can be regarded as an annotation to every node containing a set of functions. This annotation represents the functions for the case where every possible path $\pi = (s, ..., k)$ from the start node up to the current node is accounted for. Based on this perception, we add annotations that represent the functions for certain subsets of all possible paths π, which exclude a certain infeasible path or a combination thereof. As one annotation for each element in the power set of infeasible pairs is necessary during the traversal, we use those elements to denote the annotations. Naturally, $\varnothing$ may be used to denote that no infeasible pair is excluded, i. e. it denotes the functions that take all possible paths into consideration.

Definition 11. *An annotation I to a node k is a set of infeasible pairs i for which k may be in the transitive closure. The maximal annotation I_k contains all of those infeasible pairs: $I_k := \{i \mid (i_{start}$ reach $k) \wedge (k$ reach $i_{end})\}$.*

Definition 12. *The set of annotations A_k to a node k contains the power set of I_k and therefore of all infeasible pairs relevant for k: $A_k := \mathcal{P}(I_k)$.*

Annotations are used to represent the interval functions for all paths from the start node of a graph to node k excluding the ones through the start nodes of the respective infeasible pairs. So we need to define how to determine the paths respected by an annotation.

Definition 13. *The set of all paths to a node k:* $\Pi(k) := \{\pi \mid \pi = (s, ..., k)\}$.

Definition 14. *The set of all paths corresponding to an annotation I of a node k:* $\Pi(I, k) := \{\pi \in \Pi(k) \mid \forall i \in I : i_{start} \notin \pi\}$.

For example, $\{i1\}$ contains the minimal intervals and maximal number of events along any paths from the start node of the graph to the current node that do not pass through $i1_{start}$, while $\{i1, i2\}$ is even more restrictive and excludes all paths through $i1_{start}$ or $i2_{start}$. The functions not accounting for any infeasible paths/pairs are denoted as $\varnothing$ and are always calculated and updated.

Start of an infeasible pair: When the start node i_{start} of an infeasible pair i is encountered, the new set of annotations is expanded to include all new combinations of infeasible pairs in order to satisfy Definition 12. However, all those new annotations $\{I \in I_k \mid \exists i \in I : i_{start} = k\}$ represent according to Definition 14 the interval functions for all paths not containing the current node. Therefore, it makes no sense to calculate them at this point and for all nodes dominated by i_{start}, so for now we just mark them as dominated by said infeasible pair. We also define the domination of a node by a set of infeasible pairs:

Definition 15. Set-dominance: *A node k is dominated by a set of infeasible pairs I, if it can only be reached on a path through a start node of one of the infeasible pairs:* I *dom* $k := \forall i \in I, \forall \pi(k) : i_{start} \notin \pi$.

Nodes with a single predecessor (concatenation): When processing a node with a single predecessor while traversing the graph, the changes to the original event dependency analysis are rather straight-forward: The functions of each annotation are updated according to their original definitions, as long as the annotation is not marked as dominated. If the latter one is the case, processing of the annotation is simply skipped. This is also true for all annotations for which any element is marked as dominated, as according to Definition 14 there would be no path to the current node that does not contain $i2_{start}$. For example, if the preceding node has annotations for $\{i1\}$ and is dominated by $i2$, interval functions for $\{i2\}$ and $\{i1, i2\}$ do not have to be calculated.

Nodes with more that one predecessor (merging): Whenever two or more flows of control are merged, the information in all the annotated functions has to be preserved. Therefore, the functions for all possible annotations (i. e. combinations of infeasible pairs) are calculated. In particular, all annotations of all predecessors are merged separately. As our approach is to only annotate the functions when necessary, there may be the case that not all of the predecessors have the annotations to be calculated. For example, the $\{i1, i2\}$s are being merged, but there is no path through $i2_{start}$ to one of the nodes being merged, therefore it has no annotation $\{i1, i2\}$. In that case, the "next best" constrained annotation concerning the set of functions is used for the merge. The rational behind this is the following: If for all combinations of infeasible pairs the corresponding interval functions had been annotated to every

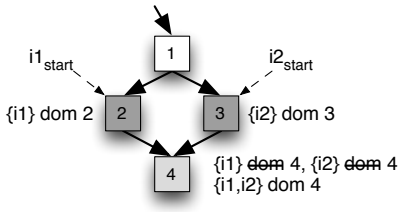

Fig. 5. Example of a node being dominated by a combination of start nodes

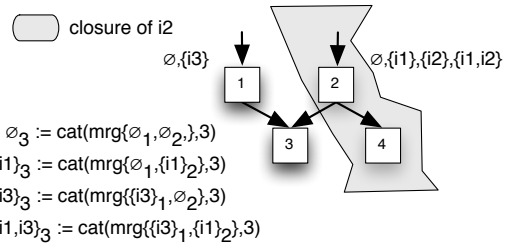

Fig. 6. Example of a leaving a closure together with a merge

node, the annotation $\{i1, i2\}$ would have denoted the functions for all paths except the ones containing $i1_{start}$ or $i2_{start}$. As in this case there is no path to the current node through $i2_{start}$, the functions annotated for $\{i1\}$ are "just as good" and can be used for the merge.

More formally, we use the the annotation I that is the intersection of I_k with the annotation M that is currently being merged, as it is a valid annotation of node k: $I = I_k \cap M$. We can do this, because in this case the set of paths corresponding to the annotations is identical, and therefore the annotated functions are as well. We show this in the following theorem:

Theorem 1. *Let $I_k \subseteq M$, then $\Pi(I_k \cap M, k) = \Pi(M, k)$.*

Proof: We first show $\Pi(I_k \cap M, k) \supseteq \Pi(M, k)$:
From $I_k \subseteq M \Rightarrow I_k \cap M \subseteq M$. and Definition 14 follows

$$\{\pi \in \Pi(k) \mid \forall i \in I_k \cap M : i_{start} \notin \pi\} \supseteq \{\pi \in \Pi(k) \mid \forall i \in M : i_{start} \notin \pi\}.$$

To show $\Pi(I_k \cap M, k) \subseteq \Pi(M, k)$ we show $\pi \in \Pi(I_k \cap M, k) \Rightarrow \pi \in \Pi(M, k))$:

$$\begin{aligned} &\pi \in \Pi(I_k \cap M, k) \\ \Rightarrow &\pi \in \{\pi \in \Pi(k) \mid \forall i \in I_k \cap M : i_{start} \notin \pi\} \quad (1) \\ \Rightarrow &\forall i \in I_k \cap M : i_{start} \notin \pi. \end{aligned}$$

$$\begin{aligned} i \in M \setminus I_k \Rightarrow i \notin I_k \Rightarrow \neg(i_{start} \text{ reach } k) \\ \Rightarrow \forall i \in M \setminus I_k : \neg(i_{start} \text{ reach } k). \end{aligned} \quad (2)$$

Clearly $\neg(i_{start}$ reach $k) \Rightarrow \forall \pi \in \Pi(k) : i_{start} \notin \pi$, therefore continuing from (2)

$$\Rightarrow \forall i \in M \setminus I_k, \forall \pi \in \Pi(k) : i_{start} \notin \pi. \quad (3)$$

With $(M \cap I_k) \cup (M \setminus I_k) = M$ we combine (1) and (3)

$$\begin{aligned} &\forall i \in M, \forall \pi \in \Pi(k) : i_{start} \notin \pi \\ &\Rightarrow \pi \in \{\pi \in \Pi(k) \mid \forall i \in M : i_{start} \notin \pi\} \\ &\Rightarrow \pi \in \Pi(M, k). \end{aligned}$$

■

What remains open is the case of predecessors being dominated by nodes of the type `inpa`-start. The following cases need to be distinguished:

1) All predecessors are dominated by the same `inpa`-start. In that case, the current node is also dominated and no calculation is necessary. Mark the current node as being dominated.

2) Concerning the combination of infeasible pairs currently to be calculated, all of the predecessors are dominated by a combination of infeasible pairs of which the super set will contain the currently processed combination, then at least the current combination is infeasible. An example for this is depicted in Figure 5. Node 4 is not dominated by $i1$ or $i2$, but by their combination. This is handled by the set-domination (Definition 15).

3) At least one predecessor is not annotated as being dominated. In this case, the merge will use the intersection with I_k as shown in Theorem 1. For the purpose of merging, i. e. finding the minimal intervals, a predecessor being marked as dominated by a set of `inpa`-starts is assumed to have an annotation for this set but the intervals are assumed to be ∞ and $maxE$ is assumed to be 0. That way, the predecessor is ignored for the calculation of the respective combination of infeasible pairs.

Leaving a shortest infeasible path: When leaving a shortest infeasible path of an infeasible pair i, the number of additional annotations is reduced (see Definition 12). There are two ways to leave such a path:

The first one is by leaving the transitive closure of i_{start} and i_{end} not but passing through i_{end}, i. e. reaching a node unequal to i_{end} from which i_{end} can no longer be reached. In this case all the additional information relating to the respective infeasible pair can simply be discarded. This is also valid for all elements in the set of annotations that contain the respective infeasible pairs. If the closures of multiple infeasible pairs are being left at the same time, this procedure is applied for all of those infeasible pairs:

Discard $\{I \in A_k \mid \exists i \in I : \neg(k \text{ reach } i_{end})\}\}$.

Example 1. *An example for leaving the closure of an `inpa` and performing a merge is presented in Figure 6. Let node 1 have annotation $\{i3\}$ and node 2 have annotation $\{i1\}, \{i2\}, \{i1, i2\}$. Traversing from node 2 to node 3 leaves the closure of `inpa` $i2$, therefore $\{i2\}_2$ and $\{i1, i2\}_2$ will be discarded. The merge of $\{i1\}$ fails to have an appropriate annotation in node 1 (no $\{i1\}_1$) so $\{i3\} \cap \{i1\} = \varnothing$ will be used for the merge. After the merge, node 3 is concatenated to the result. For the newly created $\{i1, i3\}$, intersections with the respective I_k have to be used on both sides.*

The second way of leaving a shortest infeasible path is by reaching i_{end}. In that case, all paths up to the current node that go through the corresponding i_{start} are definitely infeasible, so the annotations not accounting for this (whose paths contain i_{start} have to be discarded. In order to comply with Definition 12 (and to enable further merging), the annotations not containing the infeasible pair are not discarded, instead

they are mapped to the annotations A_k of the current node. For example, $\{i1, i2, i3\}$ becomes $\{i1, i3\}$ and $\{i2\}$ becomes $\varnothing$ when reaching $i2_{end}$. We define a mapping function e, that maps each of the annotations of the current node k to the annotations of the predecessor p:

$$e : A_k \mapsto A_p.\ e(I) = I \cup \{i \mid i_{end} = k\}.$$

When reaching a node that is the end of multiple `inpas`, the assignments of the annotations directly assign the remaining combinations of `inpas`. For example $\{i1, i2, i3\}$ becomes $\{i3\}$ and $\{i1, i2\}$ becomes $\varnothing$ when reaching $i1_{end}$ and $i2_{end}$ at the same time.

The amount of additional information in the form of functions annotated to the nodes will shrink with every i_{end} encountered. After traversing the whole control flow graph, only the interval functions of $\varnothing$ will remain. Those functions then contain the minimal intervals for all paths that do not pass through both nodes of any of the infeasible pairs.

D. Infeasible paths

The handling of infeasible paths as detected by SWEET is not too different form the handling of infeasible pairs. Infeasible Paths have a start node and an end node that are processed just as the ones of the infeasible pairs. The only difference is the detection of leaving the closure of all possible shortest infeasible paths. Instead of checking the reachability from the current node to the end of the path, the reachability to the next node along the path as given by SWEET has to be considered. The handling and interval calculation is the same as with infeasible pairs.

E. Combining the ideas

The handling of infeasible pairs and infeasible paths can be easily combined. The only difference is the detection of leaving the set of possible shortest infeasible paths originating from `inpas` on the one hand and from `innas` on the other. In order to be able to distinguish the two and choose the appropriate check for leaving of the closure, the annotations to the nodes have to specify the type of infeasibility. Concerning the power set, `inpas` and `innas` make an equal contribution.

The approach for handling infeasible nodes as given in subsection VI-B fits well with the handling of the other types. As it only decides whether to discard all timing information or analyze the current node, it does not interfere. Again, for the merging it assumes all intervals to be ∞ and $maxE$ to be 0.

The pseudo-code for the modified traversal of the event flow graph resembling our approach is given in Listing 2. We follow the style of the pseudo-code by Bodmann et al. The procedure `calculateIntervals` resembles the core of the original but has been extended for the handling of infeasible nodes. Only nodes that are not dominated by an infeasible node need to by analyzed. Procedure `analyze` first determines all annotations to be processed and for each that is not in a dominance relation updates the interval functions by merging or concatenating the current node. The function `allMarkAnyPredDom` checks that by trying to satisfy

$$\forall i \in I \exists p \in \text{pred}(k) : i \text{ dom } p.$$

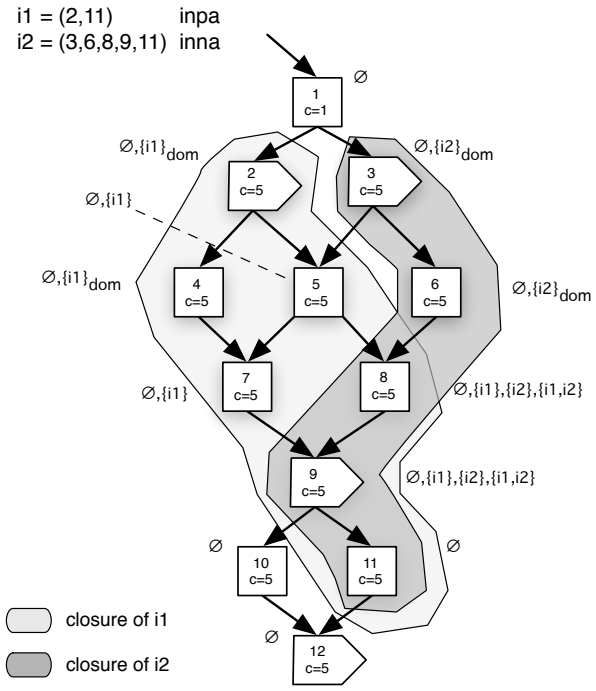

Fig. 7. Example of an event flow graph with infeasible paths

Afterwards, the case of a new shortest infeasible path starting in the current node is handled. The function `intersect` returns the intervals of an appropriate annotation by intersection of I_k with the annotation currently being merged as defined in Subsection VI-C. Accordingly, function `mapAnnotations` maps the annotations and removes the references to a shortest infeasible path when reaching its end node.

VII. Example

A small example of an annotated event flow graph is shown in Figure 7. It contains two different sources of infeasibility, an infeasible pair of type `inpa` $i1 = (2, 11)$ and an infeasible path of type `inna` $i2 = (3, 6, 8, 9, 11)$. The respective closures, in which the shortest infeasible paths lie and the annotations are depicted as well. Table I shows the resulting intervals when accounting for different sources of infeasibility. In each case accounting for $i1$ and for $i2$ allows for a relaxation of the outgoing event densities, but accounting for both allows an even greater relaxation. This is the end result of the modified analysis considering all the inherent infeasible paths.

TABLE I. Event intervals of the example

Without infeasible paths:

Events	1	2	3
inI	0	5	11
startI	5	12	17
endI	0	5	11

Accounting for $i1$:

Events	1	2	3
inI	0	5	12
startI	5	12	17
endI	0	5	12

Accounting for $i2$:

Events	1	2	3
inI	0	5	11
startI	5	12	18
endI	0	5	11

Accounting for $i1$ and $i2$:

Events	1	2	3
inI	0	5	14
startI	5	12	20
endI	0	5	14

Listing 2. Pseudo code handling infeasibility

```
procedure calculateIntervals(FlowGraph G);
  Node k := start(G);
  k.preG := ∅;
  push(k);
  while (stack not empty) do
    k := pop();
    if (allPredVisited(k)) then
      if(k in infNodes) then
        markInsadom(k);
      else if(insadom(p) for all p in pred(k)) then
        markInsadom(k);
      else
        analyze(k);
      end if;
      markVisited(k);
      for k' in succ(k) do
        push (k');
      end for;
    end if;
  end while ;
end;

procedure analyze(Node k)
  for I in powerSet(pred(k).annotations) do
    if(allMarkAnyPredDom(I,k)) then
      markDom(I,k);
    else
      if(predCount(k) > 1) then
        merge(I,k);
      else
        concat(I,pred(k),k);
    end if;
  end for;
  for i in infPairsAndPaths do
    if(k == i.start) then
      markDom(i,k);
      for I in k.annotations do
        if(not I dom k) then
          k.join(I,i) := I;
        end if;
      end for;
    end if;
  end for;
end;

procedure merge(Annotation I, Node k)
  Node tmp := new(Node);
  for p in pred(k) do
    if(I in p.annotations) then
      if(dom(I,p)) then
        tmp := mrg(tmp,∞);
      else
        tmp := mrg(tmp,p.I);
      end if;
    else
      tmp := mrg(tmp,intersect(I,p));
    end if;
  end for;
  concat(I,tmp,k);
end;

procedure concat(Annotation I, Node p, Node k)
  if(notInClosure(I,k)) then
    discard(I);
  else for i in infPairsAndPaths do
      if(k == i.end) then
        mapAnnotations(I,k);
      end if;
    end for;
  end if;
  k.I := cat(p.I,k);
end;
```

VIII. Conclusion and future work

In this paper we have shown an approach to directly incorporate information about infeasible paths in a task into the event dependency analysis. Prior work had demonstrated how this can be accomplished without modification of the analysis by preprocessing the control flow graph but resulted in an nearly explicit path enumeration. Our approach in contrast is able to operate on an unmodified graph by modifying the analysis to directly utilize information about infeasiblities. We accomplish this by calculating additional timing information for all nodes that may be part of an infeasible path and preserving this information across merge points in the control flow. We have shown how to account for flow facts generated by the worst-case execution time tool SWEET which is able to identify three different origins of infeasible paths. In a next step, the handling of loops can be integrated directly into the event dependency analysis. The flow facts generated by SWEET already include estimations for the loop bounds.

References

[1] F. Bodmann, K. Albers, and F. Slomka, "Analyzing the timing characteristics of task activations," in *International Symposium on Industrial Embedded Systems 2006, IES'06*. IEEE, 2006, pp. 1–8.

[2] K. Kempf, S. Kollmann, V. Pollex, and F. Slomka, "Relaxing event densities by exploiting infeasible paths in control flow graphs." in *RTNS*, 2011, pp. 75–84.

[3] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and Microprogramming*, vol. 40, pp. 117–134, 1994.

[4] K. Richter, "Compositional scheduling analysis using standard event models - the symta/s approach," Ph.D. dissertation, University of Braunschweig, 2005.

[5] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister, "Performance evaluation of network processor architectures: combining simulation with analytical estimation," *Comput. Netw.*, vol. 41, no. 5, pp. 641–665, 2003.

[6] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *ACM SIGPLAN Notices*, vol. 30, no. 11, pp. 88–98, 1995.

[7] J. Gustafsson, A. Ermedahl, and B. Lisper, "Algorithms for Infeasible Path Calculation," in *Sixth International Workshop on Worst-Case Execution Time Analysis,(WCET'2006), Dresden, Germany*, 2006.

[8] I. Stein and F. Martin, "Analysis of path exclusion at the machine code level," in *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[9] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "On the tractability of digraph-based task models," in *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*. IEEE, 2011, pp. 162–171.

[10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 741–746.

[11] R. Bodík, R. Gupta, and M. Soffa, "Refining data flow information using infeasible paths," in *Software engineering-ESEC/FSE'97: 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 1301. Springer, September 1997, p. 361.

[12] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann, "New developments in wcet analysis," in *Program analysis and compilation, theory and practice*. Springer-Verlag, 2007, pp. 12–52.

[13] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Efficient detection and exploitation of infeasible paths for software timing analysis," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 358–363.