Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis

Dominic Oehlert
Hamburg University of Technology, Hamburg, Germany
dominic.oehlert@tuhh.de

Selma Saidi
Hamburg University of Technology, Hamburg, Germany
selma.saidi@tuhh.de

Heiko Falk
Hamburg University of Technology, Hamburg, Germany
heiko.falk@tuhh.de

Abstract

Event arrival functions are commonly required in real-time systems analysis. Yet, event arrival functions are often either modeled based on specifications or generated by using potentially unsafe captured traces. To overcome this shortcoming, we present a compiler-based approach to safely extract event arrival functions. The extraction takes place at the code-level considering a complete coverage of all possible paths in the program and resulting in a cycle accurate event arrival curve. In order to reduce the runtime overhead of the proposed algorithm, we extend our approach with an adjustable level of granularity always providing a safe approximation of the tightest possible event arrival curve. In an evaluation, we demonstrate that the required extraction time can be heavily reduced while maintaining a high precision.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Compilers, Mathematics of computing → Integer programming

Keywords and phrases compiler, real-time, event arrival functions, extraction

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.4

1 Introduction and Motivation

The design of safety-critical real-time systems often requires an effective analysis of the worst-case timing behavior in order to determine the compliance of the system to the timing constraints. This usually involves a traditional two-steps approach [3] which consists of a first low-level code analysis to determine the worst-case execution time of every task based on its program structure, followed by a system-level timing analysis to determine the worst-case response time of interfering tasks based on abstract models of the tasks activations and the scheduling policy.

In particular, system-level analysis often makes use of event arrival functions [18, 20, 23, 1] in order to bound the number of accesses to the shared resources and analyze the amount of induced interference. Event streams abstract the notion of traces and describe the possible I/O timing of interfering tasks sharing resources in the system under analysis. System properties are then computed in a compositional way using algebraic techniques where event streams are used to connect components' analyses according to the system's application and communication structure.

The code-level and system-level analysis steps are complementary, however in practice they are often considered separately. Some existing approaches, such as [19, 4] extend

code-level analysis to system-level analysis by considering in a multicore system shared cache preemption delays to bound tasks' response times. These methods result in tight upper bounds on the response times. However, they consider a holistic approach for the evaluation of the worst-case execution time which cannot capture all the timing dependencies of interfering tasks. Furthermore, they are hard to scale with the complexity of the system and therefore they cannot be applied to complex hardware architectures involving on-chip interconnects and multi-level memory hierarchies.

On the other hand, system-level performance analysis approaches such as [23, 10] are more scalable and can be applied to analyze in a compositional way complex hardware structures [15]. They take as input for every task the worst-case execution time resulting from the code-level analysis, and *abstract models* of the arrival curves corresponding for instance to a known (periodic) activation pattern but which are very seldom derived using appropriate tools. This leads to harsh overapproximations in terms of timing, as the detailed event arrival curves are not known. Furthermore, the system-level results may even be unsafe due to unsafe event arrival curves resulting from e.g., traces which do not capture the worst-case behavior.

In this paper we present a compiler-based extraction of event arrival curves. Our goal is to bridge the gap between abstract system-level analysis and low-level code analysis. This is in particular very relevant for the analysis of multicore systems where there is a strong correlation between the individual timing of tasks and their cross-core interference [21]. Memory accesses constitute one main example where compared to existing approaches, such as [6, 24] that only consider a maximum total number of memory accesses for each program, arrival curves give a more precise information about the distribution of data accesses during the program execution. This allows to provide a more detailed and accurate analysis of the timing behavior of the system and to ease the integration between the worst-case execution and response time analysis steps. Yet, our proposed approach is not limited to memory accesses as it takes abstract events as an input, enabling various actions to be defined as an event (e.g., function calls).

Several existing work have investigated deriving access patterns by exploiting low-level informations. Li et al. [16] presented a mode-controlled data-flow model of real-time memory controllers. It is capable of deriving a tight worst-case bandwidth (WCBW) estimation for shared SDRAM memories. For this analysis, it is required to describe the dynamic command scheduling used by the memory controller and transaction sequence of the applications via so-called mode sequences.

Jacobs et al. [11] presented an approach for extracting safe upper event arrival curves at the code-level using compiler-based techniques. They proposed a modified version of the *implicit path enumeration technique* (IPET) [17] to find the maximum number of events potentially occurring in a given time interval on any path of the program. This approach is used to model all potential sub-paths implicitly by formulating an integer linear program (ILP). Yet, the presented approach lacks formalisms for critical aspects to ensure safeness (i.e., the resulting arrival curve should not be underapproximated) and tightness (i.e., the level of overapproximation due to the model should be minimal). The modification of the IPET approach is required since the standard approach only covers complete paths through a program. However, it is necessary to explore all possible sub-paths in a program starting and ending at any arbitrary node when deriving an event arrival curve. We extend this approach to also support lower event arrival curves and introduce a variable granularity during the extraction to find a compromise between extraction time and overapproximation.

Beside Jacobs et al. [11], only few existing work have considered IPET-based approaches exploring sub-paths in a program. Altmeyer et al. [2] presented an approach where sub-paths are defined by introducing additional preemption nodes. However, these sub-paths are restricted to the preemption nodes and are forced to terminate there which reduces the number of explored sub-paths. Kleinsorge et al. [14] presented an explicit path analysis which is capable of evaluating arbitrary partial worst-case execution paths. However, due to its nature all existing loops have to be unrolled during the analysis.

Contribution

We present a formal description of a compiler-based extraction of event arrival functions. It builds on the primary approach of Jacobs et al. and extends it in several aspects, e.g., the non-trivial extraction of lower arrival curves and increasing tightness of the arrival curves. The extraction of lower arrival curves is introduced since system-level analyses, such as Real-Time Calculus [23] or SymTA/S [10], partially rely on them as well. Tightness of the event arrival curves is increased by differentiation of loop control types and consideration of minimum loop bounds. For this, we introduce a complete formalized set of equations of the model. As the essential benefit of a compiler-based extraction of event arrival functions lies in its safe- and tightness, it is relevant to formulate the description well to ensure these characteristics. Besides, we provide an algorithm in order to derive a bound on the number of events for all possible time intervals of a program's runtime. The algorithm considers a complete coverage of all possible paths and therefore builds a safe upper bound on the number of events. The execution time of the proposed algorithm depends on the structure of the program but also on the granularity of the considered time intervals and the clustering of events per basic block. Therefore, we relate the extraction time of the proposed algorithm to the granularity and discuss the duality between the considered granularity level and the precision of the derived arrival curves.

The remainder of the paper is structured as follows. In Section 2 we present the system model and how the IPET approach is extended to extract the event arrival curves while providing a full coverage of all execution paths in a program. Section 3 presents our proposed algorithm for the extraction of the event arrival curves and its extension to consider different granularities. Section 4 evaluates our algorithm and confirms our findings. Section 5 concludes the paper.

2 System Model

2.1 Context and Prerequisites

Event arrival functions allow to model the dynamics of a real-time system, even for arbitrarily triggered events. They are generally defined as follows,

▶ **Definition 1** (Event Arrival Functions). Let $\eta_i^+(\Delta t)$ and $\eta_i^-(\Delta t)$ denote for each task i the maximum and minimum number of events issued within a time window of size Δt. Their pseudo-inverse counterparts $\delta^+(n)$ and $\delta^-(n)$, return the maximum/minimum time interval between the first and the last event in any sequence of n event arrivals. The conversion between η and δ functions is straightforward and can be easily derived as explained in [5].

In order to extract event arrival curves using code-level analysis, we consider as input the low-level representation of the program implementing a task annotated with loop bounds and timings. The low-level representation of a program is close to its actual assembly

representation, yet still represented by certain data structures to ease the handling. Loop bounds are annotations which indicate the maximum or minimum number of possible iterations of a loop and can be inserted by the user or automatically. Prior to the extraction of the arrival curve, a *worst-case execution time* (WCET) analysis is performed considering no interference from other cores or tasks. The WCET of a program is the worst possible time it needs when it runs in isolation from its start until its termination. Subsequently, a *best-case execution time* (BCET) analysis is also performed. We are not discussing these analyses in further detail, since existing methods are used.

We denote events as actions triggered by an instruction or a sequence of instructions. Most notably this can be a memory access to a shared memory region or an access to an I/O device. However, the model is not restricted to this, since it solely takes as input the maximum and minimum of occurring events per basic block (BB).

2.2 Path Analysis and Event Arrival Functions

We base the extraction of the event arrival curves of a program on the control flow graph (CFG) extracted from its low-level representation. In order to determine the maximum (resp., minimum) number of events in a specific time interval Δt, all possible paths in this CFG have to be considered. Since the number of existing paths grows exponentially with the depth of conditional statements and variable loop bounds, considering all existing paths individually easily becomes infeasible. Jacobs et al. [11] proposed to exploit the so-called implicit path enumeration technique (IPET) as presented by Li and Malik [17]. This technique is typically used to locate the worst-case execution path (WCEP) of a program, over which its WCET occurs.

Using the IPET, a set of integer linear programming (ILP) *flow constraints* is generated to describe the CFG. All possible paths through the task's CFG are then implicitly described by the relation of its basic blocks in the constraints. By setting distinctive conditions, e.g., the first and last basic blocks have to be executed exactly once while maximizing the accumulated time, the WCEP can be found.

Yet, the classical IPET formulation can not be directly applied to the problem of finding maximum number of events during a given time window. This originates from the fact that we do not enforce one full path through the CFG, since we are only interested in sub-paths. Such a sub-path does not need to start at an entrypoint, nor end at an exit block. This way all possible sub-paths, which can be executed in a given time window, need to be considered.

Jacobs et al. introduced a modified IPET-based approach, in which all possible sub-paths are considered. Therefore any basic block can act as a source, whereas any reachable block can be a sink. This way any consecutive path, starting and ending at an arbitrary basic block, can be chosen by the ILP solver in order to find the sub-path over which the maximum number of events with respect to a given time interval are present.

In the sequel, we present the underlying basic model based on the previous work. A set of linear inequations is set up to describe the CFG of a program. The objective function is set to maximize the number of events on a to be chosen sub-path of the CFG, whereas the timing of this sub-path is not allowed to exceed a user-given constant.

For the upcoming we use the following notational conventions. Lower case italic Latin letters like a will be used for ILP variables. Upper case italic Latin letters like A represent constants inside the ILP model. Table 1 contains all ILP variables used in the paper. Unless otherwise stated, all ILP variables have a lower bound of 0. Lower case Latin letters as a subscript represent an index. Table 2 contains further miscellaneous symbols used.

Table 1 ILP decision variables.

Symbol	Description
a_i^+ (a_i^-)	Maximum (minimum) number of events contributed by basic block i on the sub-path
a_{Total}^+ (a_{Total}^-)	Maximum (minimum) number of events occurring along the sub-path
b_i	Reduction factor for basic block i if it is used as a starting and/or ending block
e_i	Basic block i is used as an end of the sub-path
f	Binary variable indicating if the chosen path covers a complete path through the program
g_ℓ	Number of flows at the loop entrance of loop ℓ
h_ℓ	Number of flows exiting the loop ℓ
n_ℓ^T	Maximum number of flows through the back edge of tail-controlled loop ℓ
n_ℓ^H	Maximum number of flows into the body of head-controlled loop ℓ
o_ℓ	Binary variable indicating the if the start of the sub-path was placed inside the loop ℓ
$p_{i,j}$	Total number of flows from basic block i to j
r_s (r_e)	Binary variable indicating if a timing reduction is applied at the starting (ending) block
$s_{i,j}$	Edge from basic block i to j is used as a starting edge
s_j	Any incoming edge at basic block j is used as starting edge
w_i^+ (w_i^-)	Total number of cycles contributed by basic block i on the sub-path when generating an upper (lower) arrival function

Table 2 Miscellaneous symbols.

Symbol	Description
A_i^+ (A_i^-)	Maximum (minimum) number of events of basic block i
$\mathcal{B}$	A set containing all basic blocks of the program
$\mathrm{B}_\ell^{\mathrm{Up}}$ ($\mathrm{B}_\ell^{\mathrm{Low}}$)	The upper (lower) loop bound of loop ℓ
C_i^+ (C_i^-)	WCET (BCET) of basic block i
$\mathcal{C}_z$	A set containing all calling edges to the function z
$\mathcal{E}_\ell$	A set containing all entry basic blocks of loop ℓ
$\mathcal{E}_\ell^r$ ($\mathcal{E}_\ell^i$)	A set containing all regular (irregular) entry basic blocks of loop ℓ
$\mathcal{F}$	A set containing all functions of the program
$\mathcal{L}$	A set containing all loops of the program
$\mathcal{L}_T$ ($\mathcal{L}_H$)	A set containing all tail-controlled (head-controlled) loops
$\mathcal{M}_\ell$	A set containing all basic blocks belonging to loop ℓ
$\mathcal{N}_\ell$	A set containing all back edges of loop ℓ
$\mathcal{P}_i$	A set containing all direct predecessors of basic block i
$\mathcal{R}_z$	A set containing all possible return edges of the function z
$\mathcal{R}_{z,(i,j)}$	A set containing all possible return edges of the function z when called using edge (i,j)
$\mathcal{S}_i$	A set containing all direct successors of basic block i
T_ℓ	Equals 1 if loop ℓ is tail-controlled, otherwise 0
$\mathcal{X}_\ell$	A set containing all exit basic blocks of loop ℓ

As mentioned previously, a WCET (resp., BCET) analysis is first executed where all accesses to a shared memory are assumed with a minimal (resp., maximum) latency. Additionally, variable timings (which may be influenced by the event-type under focus or caches) have to be considered carefully, such that they do not thwart a safe WCET (resp., BCET) estimation. Note that, the system is evaluated in isolation, without considering interference from other cores. We consider the timing of a basic block in terms of cycles. Therefore integer variables are suitable to represent the timing of a basic block. Subsequently, the CFG is synthetically modified, such that every basic block has a successor and predecessor. These additional blocks are not inserted into the actual program code and are only present in our analysis. A virtual source $\ominus$ is created for the entrypoint and inserted as a predecessor to the first basic block. In the same fashion, virtual sinks $\perp$ are created for all possible exits and inserted as a successor to the last basic blocks. Therefore, for every basic block i in the CFG, a flow constraint is generated as follows,

$$\sum_{j \in \mathcal{P}_i} p_{j,i} - e_i = \sum_{k \in \mathcal{S}_i} (p_{i,k} - s_{i,k}) \tag{1}$$

The integer variable $p_{i,k}$ describes the number of times the control flow (subsequently simply called *flow*) enters basic block k from basic block i. Each input flow of a basic block represents one execution of the basic block. The variable e_i is bound to a binary value and is set to 1 when the basic block i is used as the last basic block in the chosen sub-path. It represents a "movable" sink. In a similar manner, the variable $s_{i,k}$ is bound to a binary value and is set to 1 when the basic block k is the first basic block in the chosen sub-path. In particular, the edge from basic block i to basic block k is used as the initial flow. The set $\mathcal{P}_i$ contains all directly preceding basic blocks of i. Similarly, the set $\mathcal{S}_i$ contains all directly succeeding basic blocks of i. This way, Equation (1) functions as a node law, assuring that the amount of flow into a node is equal to the amount of flow leaving it. Additionally, one initial flow can be inserted into a basic block without violating the constraint. In the same fashion, a path can end at a particular node if the corresponding e variable is set to 1.

The flows originating from (resp., directed to) the virtual sources (resp., sinks) are defined as follows,

$$p_{\ominus,i} = s_{\ominus,i} \tag{2}$$

$$p_{i,\perp} = 0 \tag{3}$$

Since only one consecutive path is allowed, the sum over all starting (ending) points is limited to be smaller or equal to one. Additionally, if a starting point is existing, there has to be an ending point as well:

$$\sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{P}_i} s_{j,i} = \sum_{i \in \mathcal{B}} e_i \leq 1 \tag{4}$$

where, $\mathcal{B}$ is the set holding all basic blocks of the current task.

The ILP variable s_i is set to 1 if any of the ingoing edges of basic block i is used as an initial flow. It is defined as follows:

$$s_i = \bigvee_{j \in \mathcal{P}_i} s_{j,i} \tag{5}$$

Logical operators like $\vee$ or $\wedge$ can be easily described inside the ILP formulation as shown by, e.g., Johannes [12].

We assume that all instructions which may cause an event are known. It is possible to perform a value analysis for this purpose, although potential over- or underapproximations due to unknown values should be handled carefully. Architectures featuring out-of-order execution need a particularly careful micro-architectural analysis, as instruction order may change during the execution. We will not discuss these issues in detail since they exceed the scope of this paper. We define the maximum number of events per basic block i as A_i^+. This is used to calculate the amount of events happening on the chosen sub-path.

$$a_i^+ = A_i^+ \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \tag{6}$$

$$a_{\text{Total}}^+ = \sum_{i \in \mathcal{B}} a_i^+ \tag{7}$$

The ILP variable a_i^+ represents the maximum accumulated number of events of basic block i over all its executions which are part of the chosen sub-path. a_{Total}^+ defines the maximum number of events existing on the chosen sub-path.

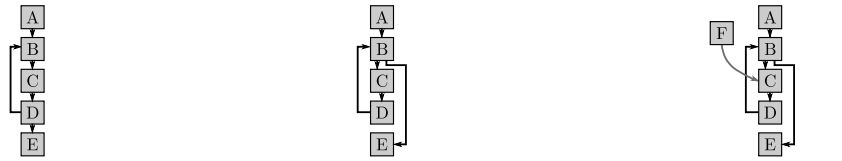
Besides the control flow and the events, also the timing has to be considered. We define w_i^+ as the number of cycles which basic block i contributes on the chosen sub-path.

$$w_i^+ = \left(C_i^- \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - (C_i^- - 1) \cdot b_i \tag{8}$$

C_i^- is the BCET of the basic block i. The BCET is chosen instead of WCET here, since we are interested in the maximum amount of events in a given time interval. Hence, using the WCET would be too optimistic, as the accumulated time over the sub-path may require less time. The ILP variable b_i is bound to an integer value between [0,2] and is defined as follows:

$$b_i = \begin{cases} 0 & \text{if } s_i = e_i = 0, \\ 2 & \text{if } s_i \wedge e_i \wedge \left(\sum_{j \in \mathcal{P}_i} p_{j,i} > 1 \right), \\ 1 & \text{else.} \end{cases} \tag{9}$$

The variable b_i functions as a reduction factor to the timing contribution of basic block i. As it is not considered at which particular location inside the basic block its events are triggered, the first and last basic block of the sub-path need to be handled with special care in order to be safe: Since a sub-path through the program can in fact start (or end) at a specific instruction inside the basic block, assuming its full BCET for this case would be too pessimistic. For this particular case we assume that all events at this bounding block happen at the very last (or first if the ending block) cycle of the basic block. In case the chosen sub-path does neither start nor end at basic block i, the accumulated timing w_i^+ is not reduced, as the reduction factor b_i is set to 0. If basic block i is chosen as the start and end of the sub-path (and it does not solely consist of the basic block i), the reduction factor b_i is set to 2. Thereby the timing contribution of basic block i is reduced by $2 \cdot (C_i^- - 1)$. Finally, if the basic block i is chosen as the start or end block (or the sub-path only consists of BB i), the reduction factor is set to 1 as a safe overapproximation. We show in Section 3 a simplistic approach to increase the granularity to a single-event level with a minor preparation of the control flow graph to reduce the introduced pessimism.



(a) Tail-controlled. **(b)** Head-controlled. **(c)** Head-controlled (irr.).

Figure 1 Sample loop structures.

Finally, the sum of all timing contributions is limited to be smaller or equal to the chosen time interval Δt, while maximizing the number of events.

$$\Delta t \geq \sum_{i \in \mathcal{B}} w_i^+ \tag{10}$$

$$\max : a_{\text{Total}}^+ \tag{11}$$

2.3 Handling Loops and Function Calls

So far, the model does not limit loop iterations. It is assumed that all loops are annotated with loop bounds. The deriving of loop bounds or control type (head- or tail-controlled) is beyond the scope of this paper and well researched [22, 25]. The previous work by Jacobs et al. [11] covers the handling of loops only very briefly. It is stated that the original IPET formulation has to be extended for the case that a path is starting inside a loop, where the loop's back edge may be taken an additional loop bound-times. Yet, no formal description is given. In the sequel, we introduce a tight and accurate description of handling loops. Besides, we introduce how function calls can be handled which the previous work [11] lacks of.

We differentiate between head- and tail-controlled loops. For tail-controlled loops we limit the number of back edges taken:

$$\forall \ell \in \mathcal{L}_T : \sum_{(i,j) \in \mathcal{N}_\ell} p_{i,j} \leq n_\ell^T \tag{12}$$

$\mathcal{L}_T$ defines the set of all tail-controlled loops. The set $\mathcal{N}_\ell$ contains all back edges of the loop ℓ. A back edge of a loop originates from the loop tail to its head. In the exemplary loop in Figure 1a this is the edge from basic block D to B. The ILP variable n_ℓ^T denotes the maximum flow through all back edges of the loop ℓ.

$$n_\ell^T = (\mathrm{B}_\ell^{\mathrm{Up}} - 1) \cdot \left(\sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} + o_\ell \right) \tag{13}$$

$\mathrm{B}_\ell^{\mathrm{Up}}$ is defined as the upper loop bound of loop ℓ. The upper loop bound of a loop defines the maximum number of loop body iterations. The set $\mathcal{E}_\ell$ contains all basic blocks, which are entrances of loop ℓ, while $\mathcal{M}_\ell$ contains all members of this loop (including nested loop members). We define an entrance block of a loop as a basic block which belongs to the loop and has a predecessor which is not part of the loop. In the exemplary loop in Figure 1a basic block B is the entrance block. This implies that $p_{j,i}$ in Equation (13) covers all edges which are entering the loop from outside, which would be the edge $p_{A,B}$ in the sample loop.

The binary ILP variable o_ℓ is forced to 1 in case any of the basic blocks inside the loop is chosen as a starting point and is defined as follows:

$$o_\ell = \sum_{i \in \mathcal{M}_\ell} \sum_{j \in (\mathcal{P}_i \cap \mathcal{M}_\ell)} s_{j,i} \tag{14}$$

Thereby, Equation (13) permits the loop body to be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times for every time the loop is entered. Furthermore, if the starting point is chosen inside the loop, the loop body can be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times additionally. This is required, as the starting block can also be chosen inside a loop.

The constraints handling head-controlled loops are very similar, yet with a few modifications in order to tighten the resulting number of events. In contrast to the tail-controlled loops, for head-controlled loops we limit the number of times the loop is actually entered. Otherwise, one additional loop execution more than feasible by the CFG would be permitted.

Example: Assume the head-controlled loop in Figure 1b has an upper loop bound of 1. If the starting point is chosen at, e.g., basic block B, the number of executed back edges would be restricted to 1, since there is no flow entering the loop. Yet, without violating the constraints, the loop body could be executed twice according to the model (sequence $\{B, C, D, B, C, D\}$), since the back edge is only executed once. Especially in case of nested loops, an overapproximation of a single loop iteration can lead to a significant overapproximation of total number of events. We therefore introduce the following equations, which limit the number of times a head-controlled loop is entered.

$$\forall \ell \in \mathcal{L}_H : \sum_{i \in \mathcal{E}_\ell^r} \sum_{j \in (\mathcal{S}_i \cap \mathcal{M}_\ell)} p_{i,j} \leq n_\ell^H \tag{15}$$

The set $\mathcal{L}_H$ contains all head-controlled loops, while the set $\mathcal{E}_\ell^r$ contains the *regular* entrance block of the loop ℓ ($|\mathcal{E}_\ell^r| = 1$). The exemplary loop in Figure 1b only has a regular entry, while the exemplary loop in Figure 1c has two entries: One regular entry (B), and one *irregular* entry (C) (irregular entries arise due to, e.g., goto-statements into loops at the source code level). Equation (15) restricts the number of times the loop body is entered via its regular entry to a maximum of n_ℓ^H. Regarding the exemplary loop in Figure 1c this represents the edge from B to C. This limit is defined as follows:

$$n_\ell^H = \mathrm{B}_\ell^{\mathrm{Up}} \cdot \left(\sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} + o_\ell \right) - o_\ell - \sum_{i \in \mathcal{E}_\ell^i} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} \tag{16}$$

Similar to tail-controlled loops, Equation (16) permits the loop body to be executed $\mathrm{B}_\ell^{\mathrm{Up}}$ times for every flow entering the loop. In case the starting point is chosen inside the loop (o_ℓ=1), the loop body can be entered an additional ($\mathrm{B}_\ell^{\mathrm{Up}}$-1) times. The deduction of 1 stems from the fact that if the starting point is chosen inside the loop, the loop is already entered once. The right-hand subtractive term is required for irregular loops. If the loop is entered via an irregular entry, the first loop iteration clearly does not include one entry from the regular entry into the loop body. As we limit the loop iterations via the number of times the loop body is entered via its regular entry, the upper limit n_ℓ^H has to be lowered for each time the loop is entered via an irregular entry.

In order to tighten the results, the ILP model also considers the minimum loop iterations.

$$\forall \ell \in \mathcal{L} : g_\ell = \sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} \tag{17}$$

$$h_\ell = \sum_{i \in \mathcal{X}_\ell} \sum_{j \in (\mathcal{S}_i \setminus \mathcal{M}_\ell)} p_{i,j} \tag{18}$$

$$\sum_{(i,j) \in \mathcal{N}_\ell} p_{i,j} \geq \min\left(g_\ell, h_\ell\right) \cdot (\mathrm{B}_\ell^{\mathrm{Low}} - T_\ell) \tag{19}$$

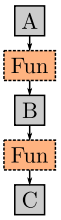


Figure 2 Exemplary CFG.

The set $\mathcal{L}$ contains all loops, whereas set $\mathcal{X}_\ell$ contains all exit blocks of the loop ℓ. Equations (17) and (18) are solely present for a better readability. Equation (17) defines the number of flows arriving at the loop head, while (18) defines the flows exiting the loop. Equation (19) sets a minimum number of loop iterations for each time the loop is entered and exited. $\mathrm{B}_\ell^{\mathrm{Low}}$ is the lower loop bound of loop ℓ, whereas T_ℓ equals 1 if ℓ is tail-controlled and 0 otherwise. The min()-Function in Equation (19) is described in the ILP as shown by Oehlert et al. [19].

Beside loops, our model is also capable of modeling function calls. It is sensitive to call edges and their corresponding return edges, i.e., for each calling edge, all valid return edges are evaluated. Calling contexts are currently not supported. To ensure tightness, we restrict the difference between ingoing and outgoing flows of functions. Since the start or end block may be chosen inside a called function, the in- and outgoing flows of a function may differ.

$$\forall \gamma \in \mathcal{F} : \forall (i,j) \in \mathcal{C}_\gamma : p_{i,j} \geq \min \left(\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n}, \sum_{(x,y) \in \mathcal{C}_\gamma} p_{x,y} \right) - s_\gamma \tag{20}$$

$$\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n} \geq \min \left(p_{i,j}, \sum_{(x,y) \in \mathcal{R}_\gamma} p_{x,y} \right) - e_\gamma \tag{21}$$

The set $\mathcal{F}$ consists of all functions inside the program, while $\mathcal{C}_\gamma$ contains all calling edges to the function γ. The set $\mathcal{R}_\gamma$ contains all possible return edges from the function γ. Furthermore, the set $\mathcal{R}_{\gamma,(i,j)}$ contains all possible return edges from the function γ when called via the edge (i,j) ($\mathcal{R}_{\gamma,(i,j)} \subseteq \mathcal{R}_\gamma$). s_γ is set to 1 if any basic block of function γ or a basic block contained by a function called by γ is used a starting block. e_γ is the corresponding counterpart for the ending points. Equation (20) sets up one constraint for every call inside the program. It sets a lower bound for the number of times the calling edge (i,j) is executed. Therefore, the minimum of flows entering the function γ and exiting via a return-edge belonging to the caller-edge (i,j) is determined. Equation (21) then sets a lower bound on the number of times a corresponding return-edge is executed. More generally speaking, the equations enforce that only call- and return-edges which belong together are allowed to be used. *Example:* Figure 2 depicts an exemplary CFG with 2 calls. The set of constraints modeling the function *Fun* contains two incoming edges, one from basic block A and one from B, as well as two corresponding exiting edges. Obviously, only the CFG-feasible paths $A \to \mathit{Fun} \to B$ and $B \to \mathit{Fun} \to C$ should be allowed, yet paths like $A \to \mathit{Fun} \to C$ not. The original IPET formulation can easily ensure this by forcing a calling edge's number of executions to be equal to the executions of its feasible return edges. As in our case sub-paths may also start (end) in a called function, the number of calls and returns may differ. Therefore the lower bound of a call-edge (return-edge) is decreased by one in case the starting (ending) point is chosen inside the called function. As recursive functions may be exited (called) multiple

times without being called (exited), this difference can also be greater than 1 (e.g., the sub-path is chosen to start in the deepest recursion level). Therefore the min()-function is used, such that minimum of overall executed calling edges and dedicated returns is evaluated in Equation (20), whereas Equation (21) handles the return-edges likewise.

This differentiation is done on one hand to ensure tightness (dedicated caller-return pairs) and on the other to enable starting and ending points to be chosen inside called functions.

2.4 Lower Bound on the Event Arrival Function

The previous approach by Jacobs et al. only focused on the extraction of an upper event arrival curve. In this section we present how lower event arrival curves can be extracted.

A lower bound on the event arrival function $\eta_i^-(\Delta t)$ can be similarly derived using the introduced ILP model, yet with several modifications and additions. Since we want to determine the minimum amount of events in a given time window, we use the WCET of a basic block instead of the BCET used for the upper bound. Therefore Equation (8) is replaced with the following one:

$$w_i^- = \left(C_i^+ \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot ((s_i \wedge r_s) \vee (e_i \wedge r_e)) \tag{22}$$

Instead of the BCET C_i^- of a basic block i, its WCET C_i^+ is used. In order to derive a safe lower event arrival curve, C_i^+ has to include all potential interferences, stalls or similar. If C_i^+ is depending on the event-type under focus, it is possible to derive an upper event arrival curve up-front and use system analysis tools [5, 23] to determine a safe WCET. In case the basic block i is used as a starting and/or ending block and the corresponding binary variable r is set to 1, the block's timing is reduced by b_i (c.f. Equation (9)). This again is done as a safe overapproximation, since we are not considering at which particular locations the event triggering instructions are located in a basic block. Although the multiplication term does not appear to be linear, it can be expressed using a simple case-structure since $((s_i \wedge r_s) \vee (e_i \wedge r_e))$ is restricted to Boolean values. In a similar manner Equation (6) is replaced:

$$a_i^- = \left(A_i^- \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot ((s_i \wedge r_s) \vee (e_i \wedge r_e)) \cdot A_i^- \tag{23}$$

A_i^- represents the minimum number of events in basic block i. The first term remains the same while a second subtractive term is introduced. Similar to Equation (22), in case basic block i is the start and/or end block of the path, its number of events can be reduced by $b_i \cdot A_i^-$. The modifications of (23) is done since we do not account for the location of events inside the basic blocks, similarly as in Equation (22). By this overapproximation we assume, that all events happen at the very first cycle (very last cycle) of a starting (ending) node. Therefore, if r_s (or respectively r_e for an end block) is set to 1, a basic block's timing is reduced and $b_i \cdot A_i^-$ events are subtracted. The variables r_s and r_e are used in order to apply a safe overapproximation for the first and last basic block of a sub-path, yet still cover all occurring events when a full path through the program is found.

We insert additional constraints to detect the case that a complete path through the program (starting at the entrypoint and ending at a sink) is chosen.

$$s_{\ominus} = s_{\ominus,j} \tag{24}$$

$$e_{\perp} = \bigvee_{i \in \mathcal{T}} e_i \tag{25}$$

$$f = s_{\ominus} \wedge e_{\perp} \tag{26}$$

In Equation (24) the basic block j is the entry basic block of the program (c.f. block A in Figure 4). The set $\mathcal{T}$ contains all possible exiting basic blocks. Therefore f is set to 1 in case the chosen path starts at the entrypoint and ends at an exiting block, resulting in a complete path through the program.

Finally, Equations (10) and (11) are replaced by the following two:

$$\Delta t \leq \left(\sum_{i \in \mathcal{B}} w_i^- \right) + (f \wedge (\overline{r_s \vee r_e})) \cdot M \tag{27}$$

$$\min : a_{\text{Total}}^- \tag{28}$$

Most notably the direction of the comparison operator in Equation (27) is flipped and the objective is changed to minimize. Again, Δt is given as a constant, representing the time interval for which the minimum number of events should be determined. Therefore the solver is forced to find a (sub-)path in the CFG which takes at least Δt cycles and the minimum amount of events. M is a sufficiently large constant. A trivial sufficient value is the WCET of the analyzed program.

In case a complete program path is covered and no reductions in terms of cycles and events are applied, Equation (27) is always satisfied. Therefore the arrival function converges at a complete path with the minimum number of total events.

3 Event Arrival Extraction Over All Existing Paths

In the following, we present how event arrival curves can be obtained with an adjustable level of precision while still resulting in a safe overapproximation. This subject is not part of the scope of the previous work [11].

3.1 Extraction Algorithms

As described previously, in order to derive an upper (resp., lower) bound on the event arrival curves by a given task, we need to explore different time intervals and extract for each duration the maximum (resp., minimum) number of events during this interval. For this, the IPET approach is customized to consider all sub-paths of duration Δt and maximing (resp., minimizing) the number of events. This procedure has to be repeated multiple times to cover all possible values of Δt. In the following we present two algorithms to explore the space of all possible values of time intervals.

Algorithm 1 is used to generate an arrival curve with an adjustable level of *time granularity* I . Here the value of Δt is bound to increasing values with a fixed increment I while solving the ILP for every value of Δt. The WCET of the program is used as an upper bound, since by definition no path can result in a higher timing than the WCEP. Note that, the smaller the value of I, the more fine-grained the generated arrival curve is. This comes with a linear increase in the number of ILP variants to be solved, one for every possible new value of Δt.

Algorithm 1 Fixed granularity extraction.

Input: I - Time granularity
Output: m - Map with the max. number of arrivals with Δt as a key

```
1: Map m
2: for (Δt=0; Δt ≤ WCET; Δt += I) do
3:    m[ Δt ] = solveILP(Δt)
4: end for
```

Algorithm 2 Binary search.

Input: -
Output: m - Map with the max. number of arrivals with Δt as a key

```
1: Map m, List w // w contains all windows to be analyzed
2: w.push({ 0, WCET } )
3: while !(w.empty()) do
4:    Pair curWindow = w.pop()
5:    if !(m[curWindow.lower] exists) then
6:       Δt = curWindow.lower
7:       m[curWindow.lower] = solveILP(Δt)
8:    end if
9:    if !(m[curWindow.upper] exists) then
10:      Δt = curWindow.upper
11:      m[curWindow.upper] = solveILP(Δt)
12:   end if
13:   if m[curWindow.lower] ≠ m[curWindow.upper] and (curWindow.upper -
      curWindow.lower ) > 1 then
14:      x = ⌊(curWindow.lower + curWindow.upper)/2⌋
15:      w.push({curWindow.lower, x})
16:      w.push({x, curWindow.upper})
17:   end if
18: end while
```

It is noteworthy that this approach still results in a safe (overapproximated) arrival curve where a coarse-grain arrival curve always dominates a fine-grain arrival curve. A further discussion will be presented in the evaluation Section 4.

While this approach is reasonable for a limited amount of sample points over the arrival curve, it is not applicable for generating an arrival curve covering all potential intervals (i.e $I = 1$). For this circumstance, we present in Algorithm 2 another procedure based on a binary search. We exploit two facts regarding the event arrival curves: i) they are monotonically increasing, ii) they are piecewise step functions (i.e., we will not necessarily have for instance a memory access at every cycle of execution). Therefore, for a given interval of Δt, we first examine the maximum number of events at the outer boundaries of the interval. If this number is equal at the boundaries, then no new event has occurred during this interval and thereby no further analysis is required inside the current time interval, since all intermediate values will result in the same maximum (resp., minimum) number of events at the interval boundaries. Otherwise, the interval (initially set to [0,WCET]) is split in half and the procedure is further repeated until all intervals in the curve are covered. Note that, both algorithms can be used to generate either an upper event arrival curve or a lower event arrival curve.

Figure 3 A sample basic block before and after splitting.

3.2 Refining the Basic Block Granularity

The number of events is analyzed on a basic block level. In case a basic block i has A_i events, this number is accounted for the whole block, leading to another overapproximation since we do not consider where these events are located during the execution of the basic block. In order to refine the level of granularity we partially re-structure the basic blocks which contain potential event triggering instructions. These basic blocks are transformed into multiple "sub basic blocks" as shown in [19]. Therefore, all basic blocks containing instructions which potentially trigger an event are split up into so-called sub basic blocks to isolate the event. Such sub basic blocks solely consist of the event's single instruction.

Consider the example depicted in Figure 3. After refining the granularity the basic block is split up into 3 sub basic blocks, where the second sub basic block only consists of the potentially data accessing instruction. This is shown in Figure 3b. This technique can be applied prior to the ILP generation. Besides, the ILP model with the refined sub basic blocks can be set up using the same constraints as presented.

Therefore, combining this refining technique and the presented extraction algorithms, the granularity can be adjusted at two levels: 1) Calculating a fixed number of sample points versus a complete curve coverage. 2) Considering a clustered number of events per basic block versus isolating each event in a separate sub basic block.

4 Evaluation

All experiments are performed on an Intel Xeon Server (20 cores at 2.3 GHz, 94 GB RAM) and the ILPs were solved using Gurobi 7.5.0. For evaluation purposes the MRTC benchmark suite [9] with annotated loop bounds from the TACLeBench project [7] are used. All benchmarks are compiled with the WCET-aware C compiler (WCC) [8] and the `-O2` flag activated which enables several ACET-oriented optimizations. As an exemplary evaluation platform the ARM7TDMI architecture (without caches) is chosen. Timing analyses are performed using methods described by Kelter [13]. The benchmark `duff` is excluded from the evaluation, as it is not supported by the currently used timing analysis tool.

For all our experiments, we focus on extracting event-arrival functions for *data accesses*. We therefore assume each access of a data object to generate an event.

4.1 An Illustrative Example

In the following, we illustrate the approach considering the control flow graph example depicted in Figure 4. We show how to derive the arrival curve $\eta^+(\Delta t)$ representing an upper bound on the number of data accesses. For this, each basic block i is annotated with its execution time C_i (for this particular example we assume that the BCET of a basic block is equal to its WCET) and its number of events A_i. Note that we do not consider any

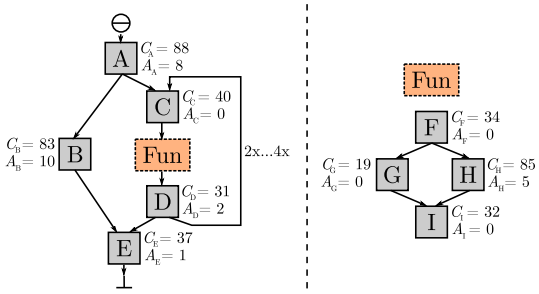


Figure 4 Sample control flow graph.

distribution of events inside a basic block. The considered example contains a tail-controlled loop with a minimum and maximum number of executed back edges of 2 and 4 and therefore loop bounds of [3,5].

We first derive the ILP model of the given CFG example. A virtual source is inserted as a predecessor of the task's entrypoint basic block A and a virtual sink as a successor to its exiting basic block E. We start by setting up the node equation for the basic block A (cf. Equations (1) - (3)).

$$s_{\ominus,A} - e_A = p_{A,B} - s_{A,B} + p_{A,C} - s_{A,C} \tag{29}$$

We continue with the rest of the basic blocks of our main function.

$$p_{A,B} - e_B = p_{B,E} - s_{B,E} \tag{30}$$
$$p_{A,C} + p_{D,C} - e_C = p_{C,F} - s_{C,F} \tag{31}$$
$$p_{I,D} - e_D = p_{D,E} - s_{D,E} + p_{D,C} - s_{D,C} \tag{32}$$
$$p_{B,E} + p_{D,E} - e_E = 0 \tag{33}$$

As all node constraints for the main functions are set up, additional node constraints for the function `fun` are inserted in the same fashion.

$$p_{C,F} - e_F = p_{F,G} - s_{F,G} + p_{F,H} - s_{F,H} \tag{34}$$
$$p_{F,G} - e_G = p_{G,I} - s_{G,I} \tag{35}$$
$$p_{F,H} - e_H = p_{H,I} - s_{H,I} \tag{36}$$
$$p_{G,I} + p_{H,I} - e_I = p_{I,D} - s_{I,D} \tag{37}$$

After all basic node constraints have been inserted, additional constraints concerning the loop are also inserted (cf. Equations (12)-(14)).

$$p_{D,C} \leq n^T_{L1} \tag{38}$$
$$n^T_{L1} = 4 \cdot (p_{A,C} + o_{L1}) \tag{39}$$
$$o_{L1} = s_{D,C} + s_{C,F} + s_{F,G} + s_{F,H} + s_{G,I} + s_{H,I} + s_{I,D} \tag{40}$$

As shown in Figure 4, the loop is tail-controlled. Therefore Equation (38) limits the number of back edges executed to a maximum of n^T_{L1}. In case a chosen path starts inside the loop body o_{L1} is set to 1. Since the loop is not nested, $p_{A,C}$ can be at most 1, which bounds the number of back edges executed to be at most 4 in any case. In case the loop would be nested, for each flow entering the loop an additional 4 flows through the back edge would be permitted.

In order to tighten the number of possible events, we also consider the minimum number of loop iterations (cf. Equations (17)-(19)).

$$p_{D,C} \geq \min(p_{A,C}, p_{D,E}) \cdot 2 \tag{41}$$

Equation (41) sets a minimum number of loop iterations in case the chosen path enters and exits the loop.

Furthermore, we restrict the number of in- and outgoing flows of the function `Fun` (cf. Equations (20),(21)):

$$p_{C,F} \geq \min\left(p_{I,D}, p_{C,F}\right) - s_{\texttt{Fun}} \tag{42}$$

$$p_{I,D} \geq \min\left(p_{C,F}, p_{I,D}\right) - e_{\texttt{Fun}} \tag{43}$$

Anyhow, since `Fun` is only called by one location and not recursive, Equations (42) and (43) can be omitted in this case.

Subsequently the constraints concerning the events are inserted (cf. Equations (6), (7)).

$$a_A^+ = 8 \cdot s_{\ominus,A} \tag{44}$$

$$a_B^+ = 10 \cdot p_{A,B} \tag{45}$$

$$\ldots$$

$$a_I^+ = 0 \tag{46}$$

$$a_{\text{Total}}^+ = a_A^+ + a_B^+ + \ldots + a_I^+ \tag{47}$$

As the timing contribution of a basic block is dependent on the subtracting factor b_i (cf. Equation (9)), the corresponding constraints are inserted:

$$b_A = \begin{cases} 0 & \text{if } s_A = e_A = 0, \\ 2 & \text{if } s_A \wedge e_A \wedge \left(p_{\ominus,A} > 1\right), \\ 1 & \text{else.} \end{cases} \tag{48}$$

$\ldots$

Finally, the timing constraints are added (cf. Equations (8)-(10)).

$$w_A^+ = 88 \cdot s_{\ominus,A} - 87 \cdot b_A \tag{49}$$

$$\ldots$$

$$w_I^+ = 32 \cdot (p_{G,I} + p_{H,I}) - 31 \cdot b_I \tag{50}$$

$$\Delta t \geq w_A^+ + w_B^+ + \ldots + w_I^+ \tag{51}$$

With Δt being a constant, representing the length of the current interval.

The resulting lower and upper arrival function are depicted in Figure 5 where the granularity of Δt for the algorithm of extraction was set to 1 cycle. In the following, we detail the results of the arrival curve $\eta^+(\Delta t)$. The very first step appears at Δt=1 to 10 events (basic block B). The subsequent second step to 18 events occurs at Δt=2, happening on the path from basic block A to B. At Δt=85 a step to 19 events occurs which happens on the path $\{A, B, E\}$. The next step to 20 events is at Δt=298. This is occurring on the path $\{A, C, F, H, I, D, C, F, H\}$. The next step up to 22 events happens at Δt=415, where the previous path is extended to include basic blocks I and D (forming two complete loop iterations). At Δt=520 the maximum number of events increases to 27, including additional executions of basic blocks C, F and H. Note that there is no intermediate step to 23 events via the loop exiting path $\{D, E\}$ due to the lower loop bound of 3. From this point on the arrival curve follows a repetitive pattern. The arrival curve converges at Δt=1112 with 44 events, which covers the whole right side of the CFG from Figure 4 with the maximum amount of loop iterations.

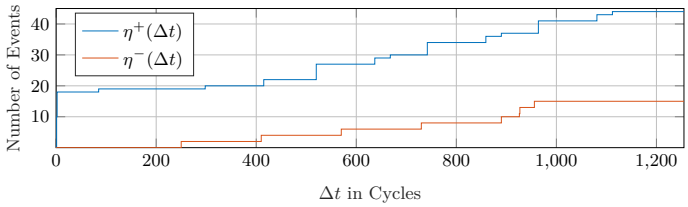
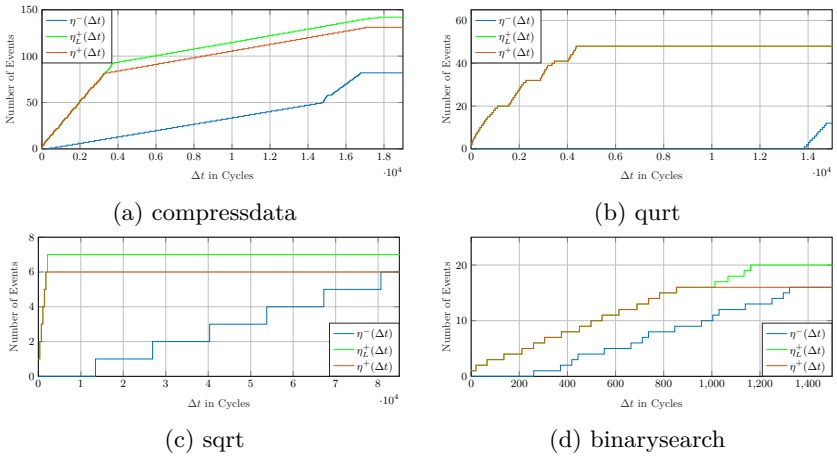


Figure 5 Extracted event arrival curves for CFG in Figure 4.



(a) compressdata (b) qurt

(c) sqrt (d) binarysearch

Figure 6 Extracted event arrival curves for benchmarks (refined BBs, binary search).

4.2 Benchmarks Evaluation

In the following, we present the event arrival curves of 4 selected benchmarks from the MRTC benchmark suite [9]. The benchmarks are chosen in order to investigate different program behaviors. Note that, an exhaustive evaluation of the benchmark suite follows in Section 4.3. All event arrival curves were extracted using Algorithm 2, while refining the event granularity to a single access. Additionally, an upper event arrival function $\eta_L^+(\Delta t)$ is generated using the same parameters, yet neglecting the loop differentiation and minimum iteration constraints introduced in Section 2.3. The sole purpose of this is to show the increased tightness due to these additional constraints in comparison to the previous work. In case of the benchmark `qurt`, $\eta_L^+(\Delta t)$ is identical to $\eta^+(\Delta t)$.

Figures 6a and 6b show $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ for the benchmarks `compressdata` and `qurt`. For both benchmarks the upper curve differs from the lower curve. This is caused by variable loop bounds, conditional statements and multiple program exits. E.g., the benchmark `qurt` can terminate with solely 12 data accesses in total or with up to 48.

Figures 6c and 6d depict the lower and upper arrival curve functions for the benchmarks `sqrt` and `binarysearch`. For both programs $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ converge to a common value. This results from the fact that each possible path through the program covers an identical total number of data accesses. However it is noteworthy that the minimum and maximum arrival of events per interval of time differs.

4.3 Granularity Evaluation

The execution time of the algorithm used for extracting the arrival curves depends on the granularity considered. Figure 7 depicts the upper event arrival functions for the benchmark `compressdata` considering different granularities. The finest possible granularity (i.e., $\Delta t = 1$

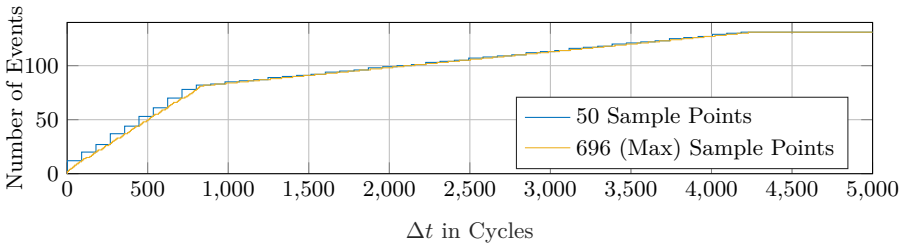


Figure 7 Event arrival curves with a different granularity and therefore number of samples.

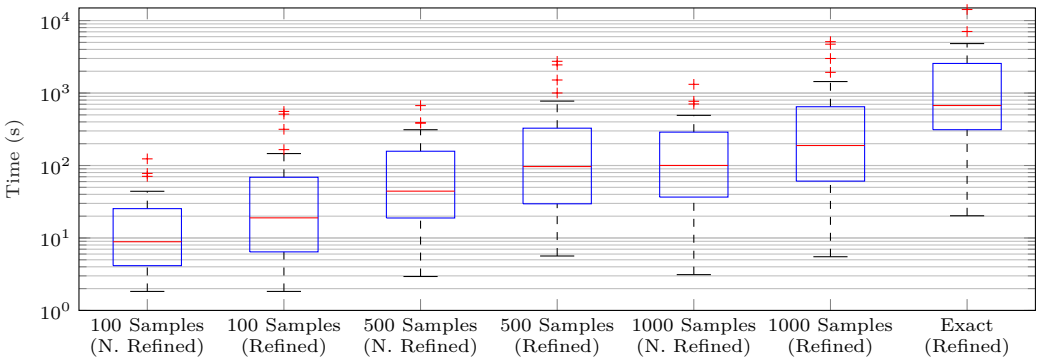


Figure 8 Overall runtimes of the extraction algorithms with different granularities.

cycle) is leading to a total number of 696 sample points (using the presented Algorithm 2). A more coarse granularity using only a total of 50 sample points is depicted as well. Note that the arrival curve with a coarse granularity always dominates the arrival curve with a finer granularity therefore leading to a *safe* approximation of the arrival curve. Even though we reduce the number of sample points, we still receive an arrival curve very close to the possible finest granularity but with the benefit of a smaller execution time. This obviously depends on the structure of the program under analysis.

Figure 8 depicts the overall execution times of the extraction, separated by the applied granularity. It is differentiated between the total number of sample points and considering the utilization of the proposed basic block refinement in Section 3.2. The right-hand side boxplot shows the execution time when using the binary search approach (Algorithm 2, 5h timeout). The central mark of each box denotes the median, while the edges depict the 25th and 75th percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75th and 25th percentile. Note that a higher number of sample points leads to a finer granularity and therefore more precision of the results. The refined BB approach also leads to more precise results as it isolates the instructions potentially accessing data, compared to the non-refined BB approach where a basic block may contain multiple data accesses. However, the refinement leads to a more complex ILP model and therefore longer execution times.

Therefore, we can clearly see that the execution time increases as we increase the number of sample points. As expected, the execution time increases as well with the utilization of the BB refinement. While the median of the extraction runtime without a basic block refinement and just 100 samples is about 9 seconds, it increases to 189 seconds with 1000 samples and refinement applied. The median of the binary search approach runtime is 673 seconds. Out of the 34 benchmarks evaluated, 10 benchmarks were canceled due to the 5h timeout when

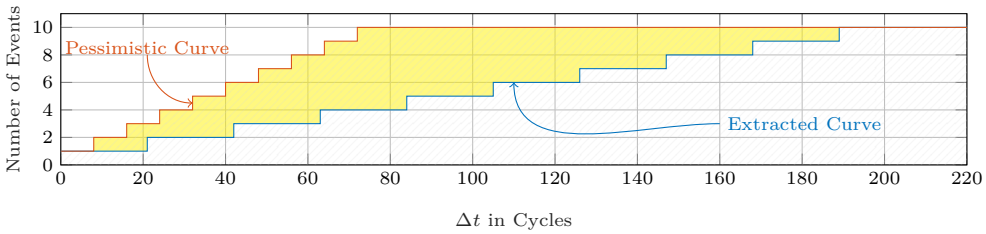


Figure 9 The metric used to evaluate the overapproximation is based on the area between the extracted curve and a corresponding pessimistic curve.

performing Algorithm 2. Therefore, there is clearly a trade-off to find between precision and execution time. In the following, we present a metric to measure the precision loss resulting from a coarser granularity approach.

4.4 Measuring the Overapproximation

In order to evaluate the level of overapproximation, we introduce the metric d_{appr}. The metric d_{appr} is defined as the area between the extracted curve and a corresponding simplistic pessimistic curve, normalized on the area below the pessimistic curve. The pessimistic curve only takes into account the maximum number of events over a complete program path and the minimum time between two events (given, e.g., by memory latencies). Therefore, d_{appr} is defined as follows:

$$d_{\text{appr}} = \frac{A_{\text{Pess}} - A_{\text{Extr}}}{A_{\text{Pess}}} \tag{52}$$

Whereas A_{Pess} is the area below the pessimistic curve and A_{Extr} is the area below the extracted curve. Figure 9 depicts the parameters used. The upper curve represents the pessimistic curve, solely generated using the maximum overall number of events and minimum time between events. The lower curve represents a curve extracted using the presented ILP model. The area difference (marked in yellow) is calculated and then normalized on the total area below the pessimistic curve. Thereby, d_{appr} reflects a magnitude to which extend the extracted curve is tighter in comparison to the pessimistic approach. When comparing the metric d_{appr} of curves extracted using different parameters of granularity (e.g., basic block refinement), the level of introduced overapproximation can be evaluated. A higher value of d_{appr} denotes a tighter curve, hence most likely leading to a tighter system-level analysis.

Figure 10 shows d_{appr} for the extracted upper arrival curves using 100, 500 and 1000 sample points. Accesses are considered separately considering a refined BB approach or bundled as initially structured by the program. It also depicts d_{appr} for upper arrival curves using the binary search approach (cf. Algorithm (2)) with basic block refinement applied. In cases the binary search algorithm was canceled due to the 5h timeout, the bar is not depicted in the diagram. The pessimistic reference curve for each benchmark was generated by using the maximum overall number of events and the minimum number of cycles between two events, given by the memory latencies. The benchmarks are listed on the x-axis. Benchmarks `janne_complex`, `expint`, `fac`, `fibcall`, `prime`, `recursion` and `cover` were evaluated but are not shown in the diagram, since no potential data accesses were detected (no data was allocated to the `.data` section).

As expected, d_{appr} is always greater or equal for a fixed number of samples when considering separated requests in comparison to bundled requests. The highest relative difference comparing separated and bundled accesses at a fixed number of sample points

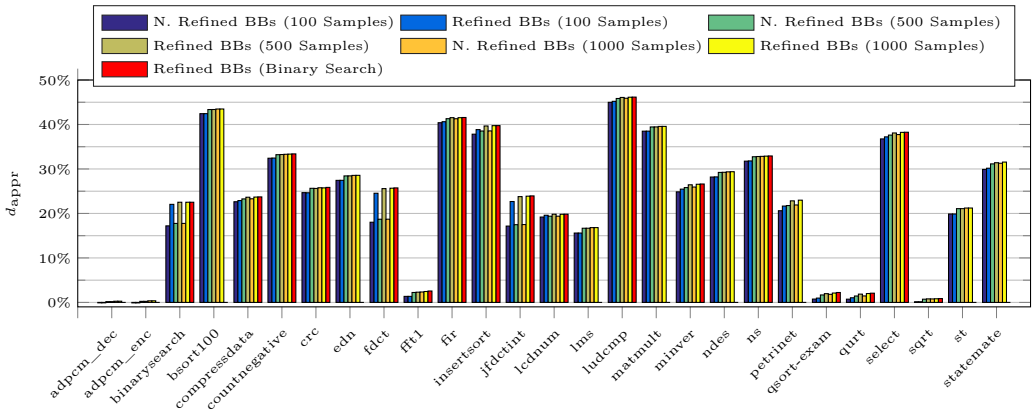


Figure 10 Metric d_{appr} for benchmarks of the MRTC benchmark suite [9].

occurs, amongst others, at the `fdct` benchmark. Using 1000 sample points and considering all requests separated, d_{appr} is at 25.7%, whereas considering accesses bundled per BB (same number of samples) results a d_{appr} value of only 18.7%. However, there are also several benchmarks for which the consideration of separated requests does not result in a lower value of d_{appr}. The benchmark `bsort100` represents such an example. Though d_{appr} increases with the number of samples, it is irrelevant whether requests are split into single blocks or not.

An exception can be seen for the benchmarks `adpcm_decoder` and `adpcm_encoder` when extracted with only 100 samples (BB refinement irrelevant), as they yield a value of d_{appr} of -1%. This is due to the low number of sample points in regard to the benchmarks' size and event arrival curves' steepness. Besides, for no other benchmark and granularity configuration a negative value of d_{appr} was observed. Overall it can be observed that d_{appr} is increasing with a higher number of samples as it is expected.

Bringing together the results regarding the required runtime from Figure 8 and the quality of the approximated curves, we can conclude that approximating the event arrival curve offers a good trade-off between extraction time and quality. If we take the benchmark `crc` as an example, the required extraction time for 500 samples and without basic block refinement drops by 98% in comparison to the extraction using the binary search algorithm in combination with refinement. Yet, d_{appr} only drops by 0.2%.

5 Conclusion and Future Work

In this paper we presented an approach to extract safe and tight event arrival functions from code-level analysis. The extracted event arrival functions can be generated with an adjustable level of granularity in order to reduce the execution time of the proposed extraction algorithm. Despite the induced overapproximation by the choice of the granularity, the presented approach results safe upper bounds of the actual event arrival curve. Furthermore, it has been shown that for some benchmarks a very good trade-off can be achieved in order to extract rapidly event arrival functions with a very good quality precision.

As a part of future work, we plan to integrate calling contexts into the model. This could further improve the tightness. Besides, we plan to exploit the detailed event arrival function knowledge for optimizations, hence improving a system's worst-case timing using the gained informations.

References

1 B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proceedings of the 2011 Design, Automation & Test in Europe Conference & Exhibition*, 2011. doi:10.1109/DATE.2011.5763145.

2 S. Altmeyer, C. Burguière, and R. Wilhelm. Computing the Maximum Blocking Time for Scheduling with Deferred Preemption. In *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, 2009. doi:10.1109/STFSSD.2009.12.

3 Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A Generic and Compositional Framework for Multicore Response Time Analysis. In *Proceedings of the 2015 International Conference on Real Time and Networks Systems*, 2015. doi:10.1145/2834848.2834862.

4 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5), 2012. doi:10.1007/s11241-012-9152-2.

5 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. In *Proceedings of the 2012 International Workshop on Analysis Tools and Methodologies for Embedded and Real-time System*, 2012.

6 Leonardo Ecco, Selma Saidi, Adam Kostrzewa, and Rolf Ernst. Real-time DRAM throughput guarantees for latency sensitive mixed QoS MPSoCs. In *Proceedings of the 2015 IEEE International Symposium on Industrial Embedded Systems*, 2015. doi:10.1109/SIES.2015.7185038.

7 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings of the 2016 International Workshop on Worst Case Execution Time Analysis*, 2016. doi:10.4230/OASIcs.WCET.2016.2.

8 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2), 2010. doi:10.1007/s11241-010-9101-x.

9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of the 2010 International Workshop on Worst-Case Execution Time Analysis*, 2010. doi:10.4230/OASIcs.WCET.2010.136.

10 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2), 2005. doi:10.1049/ip-cdt:20045088.

11 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration. In *Proceedings of the 2015 International Conference on Real Time and Networks Systems*, 2015. doi:10.1145/2834848.2834872.

12 Bisschop Johannes. *AIMMS. Optimization Modeling.* Haarlem, The Netherlands, 2009.

13 Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems.* PhD thesis, TU Dortmund University, Dortmund / Germany, 2015.

14 J. C. Kleinsorge, H. Falk, and P. Marwedel. Simple analysis of partial worst-case execution paths on general control flow graphs. In *Proceedings of the 2013 International Conference on Embedded Software*, 2013. doi:10.1109/EMSOFT.2013.6658594.

15 Adam Kostrzewa, Selma Saidi, and Rolf Ernst. Dynamic Control for Mixed-Critical Networks-on-Chip. In *Proceeding of the 2015 IEEE Real-Time Systems Symposium*, 2015. doi:10.1109/RTSS.2015.37.

16 Y. Li, H. Salunkhe, J. Bastos, O. Moreira, B. Akesson, and K. Goossens. Mode-controlled data-flow modeling of real-time memory controllers. In *Proceedings of the 2015 IEEE Symposium on Embedded Systems For Real-time Multimedia*, 2015. doi:10.1109/ESTIMedia.2015.7351770.

17 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 1995 Annual ACM/IEEE Design Automation Conference*, 1995. doi:10.1145/217474.217570.

18 Matthieu Moy and Karine Altisen. Arrival Curves for Real-Time Calculus: The Causality Problem and Its Solutions. In *Proceedings of the 2010 International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2010. doi:10.1007/978-3-642-12002-2_31.

19 Dominic Oehlert, Arno Luppold, and Heiko Falk. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *Proceedings of the 2017 Euromicro Conference on Real-Time Systems*, June 2017.

20 K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002. doi:10.1109/DATE.2002.998348.

21 Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis*, 2015. doi:10.1109/CODESISSS.2015.7331385.

22 T. Sewell, F. Kam, and G. Heiser. Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016. doi:10.1109/RTAS.2016.7461326.

23 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century.*, 2000. doi:10.1109/ISCAS.2000.858698.

24 S. Wasly and R. Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *Proceedings of the 2013 Euromicro Conference on Real-Time Systems*, 2013. doi:10.1109/ECRTS.2013.28.

25 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.