# WCET analysis in shared resources real-time systems with TDMA buses

Hamza Rihani, Matthieu Moy, Claire Maiza, Sebastian Altmeyer

# WCET Analysis in Shared Resources Real-Time Systems with TDMA Buses

Hamza Rihani
Univ. Grenoble Alpes
F-38000 Grenoble, France
CNRS, VERIMAG, F-38000
Grenoble, France
hamza.rihani@imag.fr

Matthieu Moy
Univ. Grenoble Alpes
F-38000 Grenoble, France
CNRS, VERIMAG, F-38000
Grenoble, France
matthieu.moy@imag.fr

Claire Maiza
Univ. Grenoble Alpes
F-38000 Grenoble, France
CNRS, VERIMAG, F-38000
Grenoble, France
claire.maiza@imag.fr

Sebastian Altmeyer
University of Luxembourg
Luxembourg
sebastian.altmeyer@uni.lu

## ABSTRACT

Predictability is an important aspect in real-time and safety-critical systems, where non-functional properties – such as the timing behavior – have high impact on the system correctness. As many safety-critical systems have a growing performance demand, simple, but outdated architectures are not sufficient anymore. Instead, multi-core systems are more and more popular, even in the real-time domain. To combine the performance benefits of a multi-core architecture with the required predictability, Time Division Multiple Access (TDMA) buses are often advocated. In this paper, we are interested in accesses to shared resources in such environments. Our approach uses SMT (Satisfiability Modulo Theory) to encode the semantics and execution time of the analyzed program in an environment with shared resources. We use an SMT-solver to find a solution that corresponds to the execution path with correct semantics and maximal execution time. We propose to model a shared bus with TDMA arbitration policy. Using examples, we show how the WCET estimation is enhanced by combining the semantics and the shared bus analysis in SMT.

## 1. INTRODUCTION

Time matters in safety-critical real-time systems. The predictability of these systems is needed in order to guarantee certain security and safety requirements. Determining Worst-Case Execution Times (WCET) has been the focus of research in the field of embedded systems. Static analysis methods have been developed to provide safe bound on the WCET. The challenge remains in improving the pessimistic approaches that over-estimate the execution time of the analyzed program as well as the analysis time. An example of such an approach is the Implicit Path Enumeration Technique (IPET). IPET relies on methods such as Constraint Solving or Integer Linear Programming (ILP). However, the initial version of this approach does not exclude some 'obvious' infeasible paths in a program, leading to an over-estimation on the WCET. Algorithm 1 illustrates this situation.

**Algorithm 1** Example of mutually exclusive paths

```
LOAD ...                          (1)
... /* 3 cycles */                (2)
if c then
    ... /* 5 cycles */            (3)
end if
if ¬c then
    ... /* 1 cycles */            (4)
end if
STORE ...                         (5)
```

Simple IPET without infeasible path analysis gives the longest path {(1), (2), (3), (4), (5)}. However, (3) and (4) are mutually exclusive i.e., they cannot be part of the same execution path. The longest path in this case is {(1), (2), (3), (5)}. Infeasible paths can be excluded in IPET by adding constraints to the ILP formula [15, 14]. In this paper, we use another approach with Satisfiability Modulo Theory (SMT) that allows to encode the program's semantics and its execution time.

In a multi-core environment, the longest path does not necessarily imply the worst-case execution time. The access time to the shared resource can vary depending on the arbitration policy of the shared bus and the access patterns in the execution path of the program. In the example of Algorithm 1, the STORE instruction at (5) can access the bus at different instants depending on whether the code at (2) or at (3) is executed. This leads to a variation of the bus access delay depending on the arbitration policy of the shared bus. An analysis that does not consider the semantics together within the micro-architecture analysis would have to analyze the WCET of the STORE instruction with reduced information about the possible instants when it is executed, and may therefore overestimate its execution time.

Henry et al. [9] proposed an SMT-based approach to encode the analyzed program into SMT expressions in order to estimate the WCET. An SMT-solver is used to find the longest feasible path exhibiting the worst-case execution time. In this paper, we extend this approach to include detailed architectural information, thus increasing the precision of the analysis. Our approach encodes parts of the architecture

[1] This work has been funded by grant CAPACITE (PIA-FSN2 n°P3425-146798) from the French *Ministère de l'économie, des finances et de l'industrie*.

of the hardware and the program semantics within the same SMT expression, allowing the SMT solver to prove WCET bounds that could not be deduced by analyzing both aspects independently. We focus in this paper on a shared bus with the arbitration policy *Time Division Multiple Access (TDMA)*. TDMA is a time triggered arbitration policy that periodically allocates slots of communication time to each core.

We assume a *Fully Timing Composable* system architecture [17] without timing anomalies. A timing anomaly is a situation where the local worst-case does not necessarily lead to the global worst-case [16]. We do not support unbounded loops or unbounded recursion: the analysis has to be able to unroll all loops and inline function calls. Note that this is a common restriction for programs where a formal WCET analysis is applied. Our implementation is currently a proof-of-concept to show the feasibility of the approach, and makes simplifying assumptions: we assume the absence of cache which means that each *load* or *store* instruction issues an access to the shared bus, and we consider that each LLVM instruction takes 1 cycle. As future work, we intend to incorporate static cache and timing analysis tools, such as OTAWA [2], to include realistic execution time bounds for the instructions and to model the behavior of local caches.

To sum up, our contribution is a way to encode both the semantics of the program and a TDMA arbitration policy in a single SMT expression, and use it to compute a safe bound on the WCET of the program. The encoding is carefully optimized to avoid the performance issues of a naive encoding.

The rest of the paper is organized as follows: in Section 2, we give a background on TDMA buses and the SMT-based approach for WCET analysis. Section 3 explains how a program with accesses to a shared bus with a TDMA arbiter is modeled using SMT expressions. In Section 4, we evaluate our model using micro-benchmarks, then we apply our approach to benchmarks taken from real-life applications. Finally, related work is given in Section 5 and the conclusion and future work in Section 6. The examples with SMT expressions given in this paper are expressed in pseudo-code. In our experiments, we use SMT-LIBv2 [6] which provides standard descriptions of background theories used in SMT systems.

## 2. BACKGROUND

Multi-core platforms offer capabilities that respond to the growing performance demands of embedded real-time systems. However, predictability of these architectures remain a challenge. Shared resources represent the main hot topic in the predictability of such systems. In this work we are interested in shared buses with Time Division Multiple Access arbitration policy.

### 2.1 Time Division Multiple Access (TDMA)

*Time Division Multiple Access (TDMA)* is an arbitration policy for shared buses. It allows cores to share a bus by dividing the accesses into time slots. Cores may receive different slot lengths or different number of slots in a period which gives more or less priority to some cores over the others. In our work we consider a TDMA policy where each core receives one slot per period. Figure 1 illustrates an example of a TDMA bus with 3 slots associated to 3 cores. In a period of time $\pi$, each core receives a slot of length $\sigma$. The duration of an access to the bus is *acc*. An access cannot be split over several slots and is processed only if the remaining time in the slot is sufficient. A request is processed only during its dedicated slot. $req_{A.1}$ is a bus request issued by core A during its associated communication slot. This request is executed directly within a time duration *acc*. $req_{A.2}$ is an example of a request issued outside the allowed communication slot. It is, thus, stalled until the next slot of core A. Its execution time is given by $T = \pi - (t \mod \pi) + acc$. Where $t$ is the absolute time, i.e., starting from the beginning of the execution of the analyzed program. The best case delay is when the request is issued during the slot and granted directly. In the worst-case, the request is issued when there is not enough remaining time to process it. Hence, the execution delay of a bus access varies between $[acc, \pi - (\sigma - acc)[$.

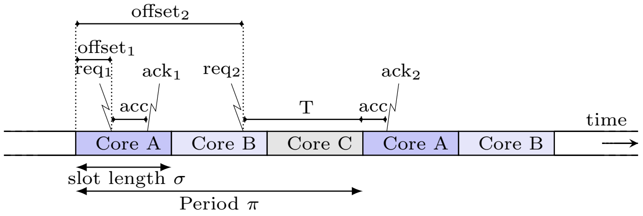


Figure 1: Bus arbitration with *Time Division Multiple Access* policy seen from the viewpoint of core A

**Algorithm 2** Example of a code fragment with bus accesses.

```
function exemple($*x, y, flag$)
    if $y < 0$ then
        $*x \leftarrow *x + 1$
    else
        $flag \leftarrow flag + 10$
    end if
    if $y \geq 0$ then
        $*x \leftarrow *x + 2$
    end if
    return flag
end function
```

We define the *offset* as $offset = t_{req} \mod \pi$. $t_{req}$ is the instant in time when the request is issued. A request is granted immediately, only if the offset at the issue instant falls in the communication slot's interval. Otherwise it is delayed until the next slot of the core. Expressing the timing of the bus in offsets simplifies the bus model since the possible values of the offsets are in $[0, \pi[$. The analyzed program can start at any offset on the TDMA period. We can indicate an initial interval or a set of values of offsets in the SMT expressions. For the simplicity of our proof-of-concept implementation, we suppose that all programs start initially at offset=0. In a real application we should consider all possible values of the offset.

### 2.2 WCET for TDMA Accesses: an Example

To illustrate the timing behavior of a TDMA bus, consider Algorithm 2. It is a simple example with two conditions and two accesses to the shared bus.

We consider each instruction to be executed in 1 processor cycle and assume that each *load* and *store* instructions access the shared bus. The shared bus has a TDMA period $\pi =$

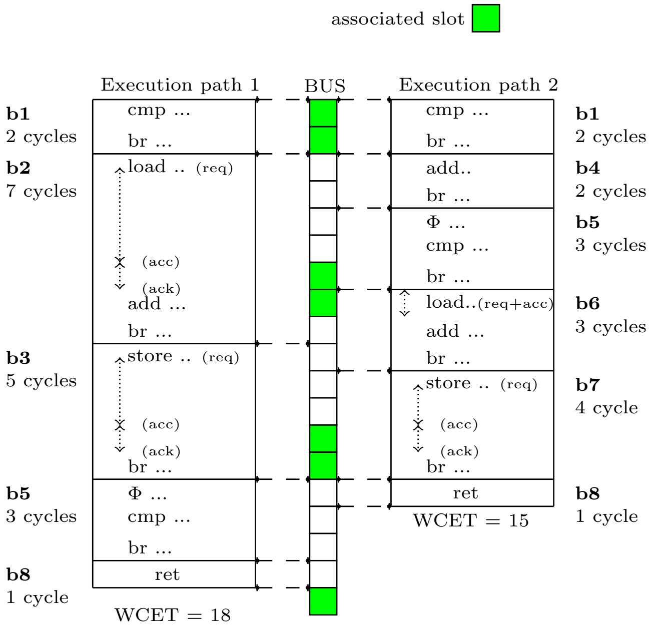


Figure 2: WCET of Algorithm 2 (CFG in Figure 3) with a shared TDMA bus with period $\pi = 6$ and $\sigma = 2$

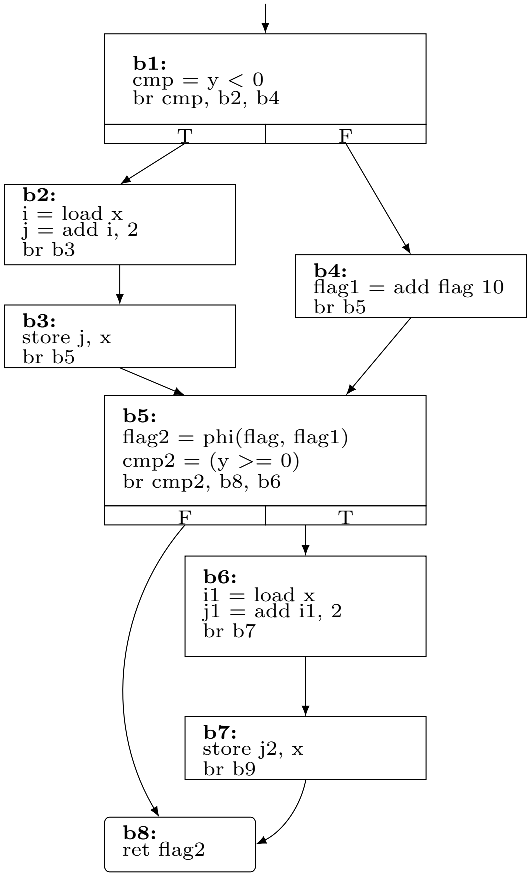
6 and a slot length of $\sigma = 2$ processor cycles. The slot associated to the core where the analyzed program is running is $[0, 2]$. A granted access is executed in 1 processor cycle.

Taking into account the parameters of the bus, a request emitted at offsets 0 or 1 is granted directly. Otherwise it is suspended until the next slot. The Control-flow graph (CFG) in Figure 3 shows two feasible paths: the first path ($y < 0$) {b1, b2, b3, b5, b8} and the second path ($y \geq 0$) {b1, b4, b5, b6, b7, b8}. Figure 2 shows both feasible paths and their execution times. We suppose that the program starts at instant $t = 0$ and *offset=0*. In the case of the first execution path, block (2) emits an access request, *load* instruction, at offset 2. This request is delayed until the next slot. The execution time of this path is 18 processor cycles. The second execution time has 15 processor cycles. In this case the worst-case execution time of Algorithm 2 is $\max(15, 18) = 18$ processor cycles.

A micro-architectural analysis that does not take the semantics of the program into account would have to consider the infeasible path {b1, b2, b3, b5, b6} when considering the *load x* instruction in block b6. This path would execute the *load* instruction with an offset of 5, hence not in the TDMA slot. As opposed to this, our analysis proves that the access is in the TDMA slot, and gives a tighter WCET.

## 2.3 WCET By SMT

Henry et al. [9] demonstrate how to measure the worst-case execution time using *Bounded Model Checking*. An SMT expressions is generated to encode the analyzed program and its execution time. The expressions mean: "Is there a path that satisfies the semantics and has an execution time greater than $T$?" The solution of this statement represents an execution path in the program with a constraint on the execution time. An SMT-solver is a program used to resolve the SMT expressions to answer the aforementioned question.


```
b1:
cmp = y < 0
br cmp, b2, b4

b2:
i = load x
j = add i, 2
br b3

b3:
store j, x
br b5

b4:
flag1 = add flag 10
br b5

b5:
flag2 = phi(flag, flag1)
cmp2 = (y >= 0)
br cmp2, b8, b6

b6:
i1 = load x
j1 = add i1, 2
br b7

b7:
store j2, x
br b9

b8:
ret flag2
```

Figure 3: Control-flow graph of Algorithm 2

In [9], the authors use a binary search method to find an upper bound of the execution time and disprove the existence of a solution with an execution time greater than the WCET.

The semantics of the program can be used in determining feasible paths. A Boolean variable is assigned to each basic block and each transition. A basic block $b_i$ is executed if any of its entering transitions is taken, i.e. $b_i = \bigvee_k t\_k\_i$. Note that loops and recursive function calls are not supported in this initial work. The compiler must unroll the loops and inline function calls.

To illustrate the SMT encoding, we use the example of block (3) from Figure 4. *b_3* is a Boolean assigned to block (3). This basic block is executed if any of the entering transitions is taken. Let *t_1_3* and *t_2_3* be the Booleans associated to the transitions from block (1) to (3) and from block (2) to (3) respectively. The generated SMT expression is:

$$b\_3 = (\ t\_1\_3 \vee t\_2\_3)$$

The outgoing transitions are obtained from the condition ($y < 0$). The transition *t_3_4* is taken when the condition is true. *t_3_5* is taken otherwise. This gives the following expressions:

$$y\_cmp = (y < 0)$$

$$t\_3\_4 = (b\_3 \wedge y\_cmp)$$

$$t\_3\_5 = (b\_3 \wedge \neg\ y\_cmp)$$

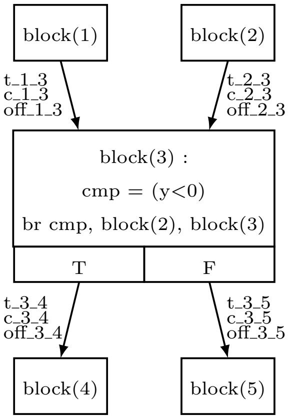


Figure 4: Example of a basic block with a join and a condition

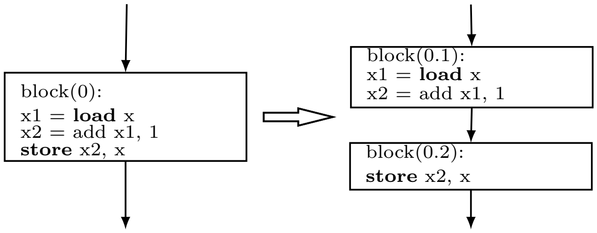


Figure 5: Basic blocks are split such that only a single bus access occurs at the beginning of the block

## 3. SMT-BASED ANALYSIS FOR TDMA

In this section, we explain how the SMT model is extended to encode accesses to a shared bus with TDMA arbitration policy. In this work we consider the first slot $[0, \sigma]$ to be associated to the core on which the analyzed program is executed. In order to simplify the analysis, we transform the control-flow graph (CFG) so that the basic blocks access the shared bus at most once and only at their first instruction. Due to this, we will refer to the *basic blocks* in the transformed CFG simply as *blocks*. Figure 5 illustrates this transformation: Considering that *load* and *store* instructions access the shared bus, block(0) is split into block(0.1) and block(0.2). Each block starts with a *load* or a *store* instruction.

We extend the work of [9] to include the model of the shared bus accesses. This model is given in Section 3.1. Introducing the access delays implies modifications in the SMT encoding of the execution time from the previous work. We explain the timing encoding in presence of bus delays in Section 3.2.

### 3.1 Shared Bus Model

Here we explain the SMT model of an access to a shared bus with a TDMA arbiter. A TDMA arbitration policy is determined by its period $\pi$ and slot length $\sigma$. The delay of a bus access at the exit of a block $T_{exit}$ is determined according to the instant $t$ of the request emission at the entry of the block $T_{entry}$.

A naive implementation of the bus access model computes first the offset from $T_{entry}$, i.e, (T$_{entry}$ mod $\pi$). Then, it checks if the offset falls in the communication slot. Algorithm 3 gives the pseudo code of a straightforward encoding in SMT of the bus access delay. The function *tdma_access* takes as argument the time instant of a bus access request and finds its offset relative to the start of the TDMA period. We then check whether the current offset falls in the allowed communication slot. In this case, the access request is directly granted and the function returns the time of the entry plus the access delay and the execution time of the remaining instructions that do not access the bus: $T_{exit} = (T_{entry} + acc + cost)$. Otherwise, the request is delayed until the next slot and the function returns $T_{exit} = \pi + (T_{entry} - offset_{entry}) + acc + cost$.

This method raises performance issues for the SMT-solver, caused by the use of the non-linear operator *mod*. Instead of modeling the absolute time through the program, we model only the offsets. The offset *off* is defined by $off = (T \mod \pi)$. Algorithm 4 gives the definition of *tdma_access* that returns the delay after a bus access with values in the interval $[acc, acc + (\pi - \sigma)]$. Another function *tdma_offset*, given in Algorithm 5, returns the offset after a bus access with values in the interval $[0, \sigma]$. Algorithms 4 and 5 are explained in Section 3.2.2.

**Algorithm 3** Naive version of *tdma_access*: returns the absolute time after a bus access

```
Require: time: $T_{entry}$, execution time of the block: $cost$
$offset_{entry} \leftarrow T_{entry} \mod \pi$
if $offset_{entry} < \sigma$ then
    return $T_{entry} + acc + cost$
else
    return $T_{entry} + (\pi - \mathrm{offset}_{entry}) + acc + cost$
end if
```

**Algorithm 4** *tdma_access*: returns the delay after a bus access

```
Require: offset $off_{entry}$, execution time of the block: $cost$
if $\mathrm{off}_{entry} \in [0, \sigma - \mathrm{acc}]$ then
    return $cost + \mathrm{acc}$
else
    return $cost + (\pi - off_{entry}) + \mathrm{acc}$
end if
```

**Algorithm 5** *tdma_offset*: returns the offset after a bus access

```
Require: offset $off_{entry}$, execution time of the block: $cost$
if $off_{entry} \in [0, \sigma - \mathrm{acc}]$ then
    new_off $\leftarrow off_{entry} + \mathrm{acc} + (cost \mod \pi)$
else
    new_off $\leftarrow \mathrm{acc} + (cost \mod \pi)$
end if
if new_off $\geq \pi$ then
    return new_off - $\pi$
else
    return new_off
end if
```

### 3.2 Timing Encoding

In this section, we explain how the timing is encoded with SMT. The assumption made previously on the form of the CFG implies that blocks either access the shared bus or not. The blocks access the shared bus only at the first instruction. The timing encoding should take into account such configuration.

### 3.2.1 Blocks Without Bus Accesses

The encoding of blocks that do not access the shared bus comes straightforward from the previous work by Henry et al. [9]. A variable $c\_i\_j$ is associated to each transition between blocks $i$ and $j$. The worst-case execution time of each block is constant considering our assumption of a *fully timing composable architecture*:

$$c\_i\_j = \textbf{if } t\_i\_j \textbf{ then } wcet_i \textbf{ else } 0$$

The expression means: if the transition from block $i$ to block $j$ is taken, $c\_i\_j$ is equal to the worst-case execution time of block $i$, otherwise it is equal to 0.

The encoding of accesses to the shared TDMA bus requires knowledge about the offsets. These offsets are computed at each exit point of a block in the CFG. The function *get_offset* in Algorithm 6 is used to find the offset after a block that does not access the shared bus. This function takes the offset $off_{entry}$ at the entry of the block and the execution time *cost* of the block. The mod operator in line 2 is used to find the offset after $t$ time. This operator does not cause performance issues because its operands *cost* and $\pi$ are known constants.

Let $i$ and $j$ be the indices of two blocks such that block $j$ is a direct successor to block $i$. Let $N \geq 1$ the number of direct predecessor of block $i$. A first encoding of the offset between block $i$ and block $j$ is the SMT expression:

$$\begin{aligned} off\_i\_j = get\_offset(\ (&\textbf{if } t\_1\_i \textbf{ then } off\_1\_i \\ &\textbf{else if } t\_2\_i \textbf{ then } off\_2\_i \\ &\textbf{else } \ldots \\ &\textbf{else if } t\_N\_i \textbf{ then } off\_N\_i \textbf{ else } 0), \\ wcet_i) \end{aligned}$$

We refer to this encoding as "if..then..else" encoding below. This expression means that the offset between block $i$ and block $j$ is computed using the offset of the corresponding entering transition in case there are many predecessors. Another possible encoding (referred to as "sum" encoding) avoids using the nested *if..then..else* sequences in the SMT expression by using a *sum* instead. We give such encoding as follows:

$$off\_i = get\_offset(\textstyle\sum_{k=1}^{k=N}\ off\_k\_i,\ wcet_i)$$

$$off\_i\_j = \textbf{if } t\_i\_j \textbf{ then } off\_i \textbf{ else } 0$$

$off\_i$ is an intermediate variable to encode the offset at the exit of block $i$. $off\_k\_i$ are the offsets associated to the entering transitions from blocks $k$ to block $i$. Only one entry transition is taken in a specific execution path. Let $n$ be a block in an execution path $\mathcal{P}$:

$$\begin{cases} t\_k\_i = false,\ off\_k\_i = 0,\ \forall\ k \neq n. \\ t\_k\_i = true,\ off\_k\_i \in [0,\pi[,\ k{=}n \end{cases}$$

This means that at most one entering offset is not *null* which implies that the sum of all entering offsets equals the offset at the entry of the block in an execution path. We apply this to block (3) in Figure 4. $wcet_3$ is the execution time of block (3). The offsets $off\_3\_4$ and $off\_3\_5$ at the exit of block (3) are encoded by:

$$off\_3 = get\_offset((off\_2\_3 + off\_1\_3),\ wcet_3)$$

$$off\_3\_4 = \textbf{if } t\_3\_4 \textbf{ then } off\_3 \textbf{ else } 0$$

$$off\_3\_5 = \textbf{if } t\_3\_5 \textbf{ then } off\_3 \textbf{ else } 0$$

**Algorithm 6** *get_offset*: returns the offset after a block without bus accesses

```
Require: offset $off_{entry}$, execution time of the block $cost$
1: new_off ← $off_{entry}$ + (cost mod $\pi$)
2: if new_off > $\pi$ then
3:     return new_off - $\pi$
4: else
5:     return new_off
6: end if
```

### 3.2.2 Blocks With Bus Accesses

Blocks that access the shared bus should take into account the delay caused by the arbitration policy. The function *tdma_access* in Algorithm 4 returns the execution time of a block taking into account the offset at its entry ($off_{entry}$) and the execution time of the remaining instructions (*cost*). In line 1, it checks whether the current offset *off* falls in the communication slot. In this case, the request is granted and the returned time at the exit of the block is given by $T_{exit} = acc + cost$. In the other case, $T_{exit} = (\pi\text{-}off_{entry}) + acc + cost$.

The function *tdma_offset*, in Algorithm 5, returns the offset at the exit of a block that accesses the bus. This function takes as inputs the offset at the entry of the block $off_{entry}$ and the execution time *cost* of the remaining instructions that do not access the bus. It computes the new offset at the exit block which is $off_{entry} + acc + cost \mod \pi$, if the $\text{off}_{entry}$ falls in the communication slot, and $[0, \sigma - acc]$ or acc + (cost mod $\pi$) otherwise. Since the offset values can only be in the interval $[0, \pi[$, The modulo operation is computed using *if..then..else* instructions (see lines 6 to 10) to avoid the non-linear instruction mod .

The execution time and the offset at the exit of a block are encoded in a similar way as blocks without accesses to shared bus. Here is how the functions defined in Figure 4 are used:

$$c\_i = tdma\_access(\textstyle\sum_{k=1}^{k=N}\ off\_k\_i,\ wcet_i)$$

$$c\_i\_j = \textbf{if } t\_i\_j \textbf{ then } c\_i \textbf{ else } 0$$

$$off\_i = tdma\_offset(\textstyle\sum_{k=1}^{k=N}\ off\_k\_i,\ wcet_i)$$

$$off\_i\_j = \textbf{if } t\_i\_j \textbf{ then } off\_i \textbf{ else } 0$$

$wcet_i$ is the worst-case execution time of the remaining instructions after the instruction that access the shared bus.

## 3.3 Adding Cuts to the SMT Expression

Experiments show a poor performance of the SMT-solver while searching for the WCET on the expression without further optimization. The same issues were observed and addressed in [9]. *cuts* are additional clauses that add no information but allow the SMT-solver to prune a very large number of partial traces from the decision tree.

Knowing that the offsets can only have the values in $[0, \pi]$ gives straightforward cuts in the case of the "sum" encoding. Let $N$ be the number of entering transition to block $i$. The sum $\sum_{k=0}^{N} off\_k\_i$ is in the interval $[0, \pi]$ since there is at most one non-zero $off\_k\_i$. We add a cut for each block with at least two entering transition, i.e., with $N > 1$.

## 4. IMPLEMENTATION AND EVALUATION

Our implementation relies on PAGAI [10], a tool used for modeling programs to SMT expressions. It is used by Henry et al. [9] to estimate the worst-case execution time through semantic encoding with SMT expressions. PAGAI uses an intermediate representation based on the CFG obtained from LLVM[1]. Due to this constraint, our tests and proof-of-concept implementation use the intermediate representation instead of the executable binary. We explain in Section 6 how a realistic analysis can be achieved.

Figure 6 shows the work flow of the proof-of-concept. The source code is compiled with CLANG to generate LLVM bitcode. A number of optimization passes are then executed. The interesting pass in our case is the one that transforms the CFG as discussed in Section 3. PAGAI is then run on the transformed CFG, which is a bitcode file, to generate the SMT expressions of the program.

We implemented an LLVM optimization pass that transforms the CFG to fit our analysis. It splits blocks before each instruction accessing the bus, so that each block contains at most one such instruction, which must be the first of the block. Figure 3 shown in Section 2.2 is the CFG obtained after this transformation.

We use the SMT-solver Z3 [7]. Z3 offers a C API that is used in our binary search program. The SMT-solver parses the SMT expressions and answers with SAT, UNSAT or, UNDEF. In case of SAT, the SMT-solver gives a model of a solution that satisfies the SMT expression. We use this model to refine the binary search. For example, we look for an execution time in the interval $[X_0, Y_0]$. The binary search algorithm checks whether the execution time is greater than $\frac{X_0+Y_0}{2}$. If UNSAT is returned, the new search interval is $\left[X_1 = X_0, Y_1 = \frac{X_0+Y_0}{2}\right]$. If SAT is returned, the SMT-solver gives a model with an execution time $Z \in \left[\frac{X_0+Y_0}{2}, Y_0\right]$. The new search interval in this case is $[X_1 = Z, Y_1 = Y_0]$. The search continues until it reaches an interval $[X_n, Y_n]$ where $X_n = Y_n$.

This approach, when applied to Algorithm 2, gives the correct and optimal worst-case execution time of 18 processor cycles after 6 iterations of the binary search. The output of the binary search is:

```
Testing wcet >= 0...  SAT (value found = 18).
                              New interval = [18, 73].
Testing wcet >= 46...  UNSAT. New interval = [18, 45].
Testing wcet >= 32...  UNSAT. New interval = [18, 31].
Testing wcet >= 25...  UNSAT. New interval = [18, 24].
Testing wcet >= 21...  UNSAT. New interval = [18, 20].
Testing wcet >= 19...  UNSAT. New interval = [18, 18].
The maximum value of wcet is 18 .

Computation time is 0.010000s
```

In the following, we evaluate our model of the shared TDMA bus. First we propose a micro-benchmark to compare the results of the naive implementation of *tdma_access* and the offset-based implementation. Then we show how the semantics encoding combined with a TDMA bus model can enhance the WCET estimation using (i) a toy example to illustrate the differences and (ii) real-world applications.

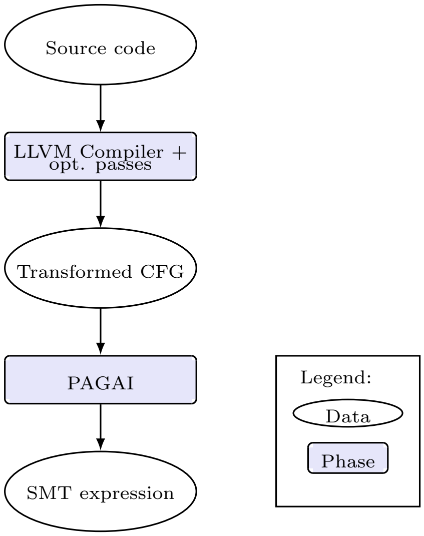


Figure 6: General work flow of the proof-of-concept to generate SMT expressions

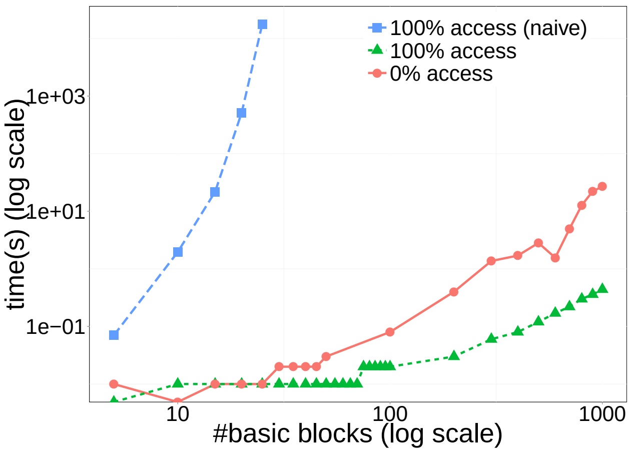


Figure 7: Comparison of the naive implementation of *tdma_access* and the offset-based implementation of *tdma_access*

## 4.1 Performance of SMT Encodings for TDMA

### 4.1.1 TDMA Functions

We now evaluate the analysis time of our model. A simple approach is to evaluate the analysis time on a linear path, i.e. without branches. The blocks are simple and have only one instruction each. Figure 7 shows a comparison of the different setups. We compare the naive implementation and the offset-based implementation of *tdma_access* on a CFG that contains only blocks with accesses to the shared bus. The naive implementation has an exponential growth of the analysis time. At only 25 blocks, it takes $17656s$ for the binary search with the SMT-solver to find the WCET. Whereas, it takes only $0.44s$ in the case of the offset-based implementation. This is mainly due to the non-linear mod operator used in the naive implementation. The line *"0% access"* represents a CFG composed with blocks that do not

[1] LLVM is a compilation framework with an intermediate representation (http://www.llvm.org)

access the shared bus which analyze *get_offset*.

### 4.1.2 Offset Encoding

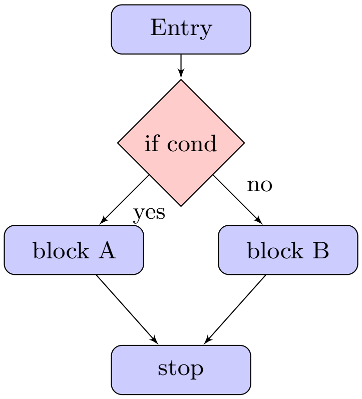


Figure 8: Example of a diamond formula

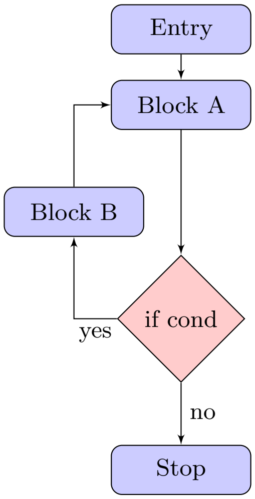


Figure 9: Example of a program with a loop

We now compare the two encodings explained in sections 3.2.1 and 3.2.2. Figure 8 shows an example with one *if* condition which will generate a "diamond formula" in SMT. We compare the analysis time of the nested *if..then..else* encoding against the *sum* encoding of an increasing number of sequences of "diamond formulas" in the analyzed program. Figure 10 shows the results for execution time of the analysis when *Block A* and *Block B* in Figure 8 access the shared TDMA bus. Both encodings have almost the same analysis time with a slight advantage of the *sum* encoding. To investigate further, we analyze the program represented in Figure 9. The loop bound is 100 iterations which will generate, when the loop is unrolled, a block with 100 entering transitions. We analyze programs with $N$ sequences of the same loop. Figure 11 shows the analysis time of the encodings with N in $\{1..10\}$. The *sum* encoding shows better performance than the nested *if..then..else* encoding. For the rest of the experiments, we will use the *sum* encoding.

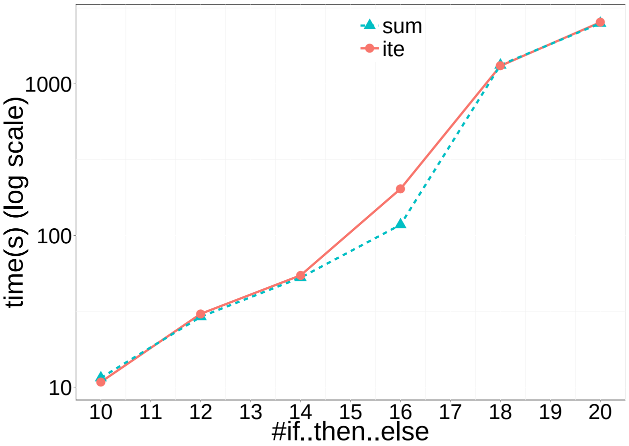


Figure 10: Comparison of nested *if..then..else* (ite) and *sum* encoding of sequences of *if..then..else* (Figure 8). TDMA bus ($\sigma = 40$, $\pi = 160$, $acc = 10$)

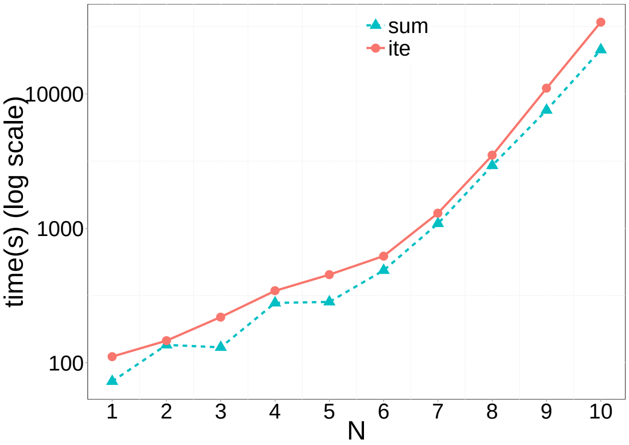


Figure 11: Comparison of nested *if..then..else* (ite) and *sum* encoding of sequences of loops with 100 iterations (Figure 9). TDMA bus ($\sigma = 40$, $\pi = 160$, $acc = 10$)

## 4.2 Realistic Benchmarks

### 4.2.1 Experimental Setup

We evaluate our approach with a subset of the TacleBench[2] benchmarks. The benchmarks are compiled with CLANG 3.6 to generate the LLVM bitcode. Loops are unrolled with an optimization pass of LLVM. The SMT expression is generated following the work flow in Figure 6. The examples are illustrated in Table 1 where "#LLVM instr." refers to the number of the instructions in the LLVM bitcode after inlining and unrolling functions and loops. "#bus access" represent the total number of *load* and *store* instructions since we consider an architecture without a cache memory. The LLVM bitcode has more instructions compared to the binary executable. Some *load* and *store* instructions in the LLVM bitcode do not exist in the executable binary which makes a direct comparison with other approaches irrelevant.

[2] http://tacle.knossosnet.gr/activities/taclebench

| Name | Description | #LLVM instr. | #bus access |
|---|---|---|---|
| bs | Binary search | 231 | 12 |
| insertsort | Insertion sort on a reversed array | 493 | 65 |
| jfdctint | Discrete Cosine Transformation | 2334 | 448 |
| fdct | Fast Discrete Cosine Transform | 2502 | 385 |
| compressdata | Data compression program adopted from SPEC95 | 674 | 131 |
| fly-by-wire | UAV fly-by-wire software | 2815 | 515 |

Table 1: Benchmarks

However, our proof of concept allows to demonstrate the feasibility of the SMT-based approach.

The analysis is run under Linux Debian, on an Intel® Core® i5-3470 at 3.20 GHz with 8GB of main memory. We consider each instruction to execute in 1 processor cycle and the platform has no cache memory.

### 4.2.2 *Results*

The TDMA policy statically isolates programs in their respective slots which means that the analysis for each program is independent from the other programs. We therefore run the analysis for individual programs, but the results hold in a context where several programs are executed in parallel.

We compare the WCET of the offset-based analysis with the pessimistic WCET where all accesses to the shared bus are considered worst-case. This implies that each *load* and *store* instructions have an execution time of $\pi - \sigma + 2.acc - 1$. Similarly to [13], the improvement is defined as *(“WCET$_{pess}$”/“WCET”–1)*.

We analyze different configurations of the TDMA bus. The results are illustrated in Tables 2, 3, 4, 5, and 6. The improvements we obtain from the offset-based analysis are proportional to the slot length and the period of the TDMA bus. The results also show that the greater the slot length is, the greater the improvement. This is expected since more accesses can be executed in the same slot. A greater TDMA period increases the pessimistic WCET. The highest improvement is 217.95% of the `bs` benchmark (231 LLVM instructions) in Table 5 with $\pi = 400$ and $\sigma = 200$.

Table 7 represents the lowest and highest observed analysis times. The offset encoding increases the analysis time of programs. The pessimistic WCET of benchmark `fly-by-wire`, from the `PapaBench` suite, is obtained in 4.02 seconds. The offset-based encoding has an analysis time of 149.01 seconds ($\pi = 400$, $\sigma = 100$, $acc = 40$). Despite the effort to linearize the SMT functions used to model the TDMA bus access, they are still very costly. The analysis time depends on the number of accesses to the shared bus as well as the number of “diamond formulas” which appears at the encoding of sequences of *if..then..else*.

| Name | WCET$_{pess}$ | WCET | Improvement |
|---|---|---|---|
| bs | 328 | 261 | 25.67% |
| insertsort | 1331 | 1313 | 1.37% |
| jfdctint | 19544 | 17893 | 9.22% |
| fdct | 17296 | 16012 | 8.01% |
| compressdata | 2650 | 2275 | 16.48% |
| fly-by-wire | 6201 | 5708 | 8.63% |

Table 2: $\pi = 40$, $\sigma = 20$, $acc = 10$

| Name | WCET$_{pess}$ | WCET | Improvement |
|---|---|---|---|
| bs | 448 | 261 | 71.64% |
| insertsort | 1951 | 880 | 121.70% |
| jfdctint | 28504 | 13213 | 115.72% |
| fdct | 24996 | 11545 | 116.50% |
| compressdata | 3790 | 1865 | 103.21% |
| fly-by-wire | 9061 | 4312 | 110.13% |

Table 3: $\pi = 80$, $\sigma = 40$, $acc = 10$

| Name | WCET$_{pess}$ | WCET | Improvement |
|---|---|---|---|
| bs | 928 | 501 | 85.22% |
| insertsort | 4431 | 1760 | 151.76% |
| jfdctint | 64344 | 26413 | 143.60% |
| fdct | 55796 | 23065 | 141.90% |
| compressdata | 8350 | 3705 | 125.37% |
| fly-by-wire | 20501 | 8682 | 136.13% |

Table 4: $\pi = 160$, $\sigma = 40$, $acc = 10$

| Name | WCET$_{pess}$ | WCET | Improvement |
|---|---|---|---|
| bs | 1768 | 556 | 217.95% |
| insertsort | 8771 | 3263 | 168.80% |
| jfdctint | 127064 | 44578 | 185.03% |
| fdct | 109696 | 38442 | 185.35% |
| compressdata | 16330 | 5799 | 181.60% |
| fly-by-wire | 40521 | 14195 | 185.45% |

Table 5: $\pi = 400$, $\sigma = 200$, $acc = 40$

| Name | WCET$_{pess}$ | WCET | Improvement |
|---|---|---|---|
| bs | 2368 | 1251 | 89.28% |
| insertsort | 11871 | 6463 | 83.67% |
| jfdctint | 171864 | 89288 | 92.48% |
| fdct | 148196 | 76842 | 92.85% |
| compressdata | 22030 | 12455 | 76.87% |
| fly-by-wire | 54821 | 29258 | 87.37% |

Table 6: $\pi = 400$, $\sigma = 100$, $acc = 40$

| Name | <40,20,10> | <400,100,40> |
|---|---|---|
| bs | 0.45 | 0.98 |
| insertsort | 1.37 | 6.56 |
| jfdctint | 44.10 | 48.54 |
| fdct | 41.36 | 34.57 |
| compressdata | 4.66 | 3.23 |
| fly-by-wire | 28.78 | 149.01 |

Table 7: Analysis time, in seconds, of the benchmarks with different configurations of the TDMA bus $<\pi,\sigma,acc>$

# 5. RELATED WORK

Chattopadhyay et al. [5] improves the analysis cost of loops by aligning each loop head execution with the TDMA period. A penalty term is added to the WCET of each loop. This allows a better scaling of the analysis at the cost of the precision. The approach by Kelter et al. offers a compromise for loop analysis by modeling the offsets in the TDMA bus with an ILP problem. The proposed solution gives a tighter estimation of the WCET compared to the pessimistic approach. Considering bounded loops, the authors gives two methods to estimate the WCET in presence of a TDMA bus with minimal unrolling. The first method unroll the loop until a fix point of offsets is reached. The second method uses dynamic flow graphs to model loops.

Schranzhofer et al. [18] propose an efficient analysis of the worst case response time (WCRT) of a shared TDMA bus. The proposed framework uses the access model in periodic tasks to analyze the worst-case response time of the bus and schedulabitly of tasks. By separating accesses to the bus and computations, this approach exhibits tighter bounds and reduces the WCRT.

Other research works were done to improve the WCET estimation with a shared bus. Gustavsson et al. [8] use timed automata to model the software and the hardware. An upper bound on the clock of the timed automata is obtained with Model Checking tools such as UPPAAL [3]. This approach suffers from a potential explosion in the number of automata's states. Lv et al. [13] propose a better use of timed automata. With an abstract interpretation of the cache, basic blocks in the CFG are annotated with *cache miss* and *cache hit*. A model with timed automata is associated according to the annotations. Arbitration policy of the shared bus is also modeled with an automata. The results show a better estimation on the WCET compared with the pessimistic approaches.

All of the mentioned related works give improvements on the upper bound of the execution time. However, they only estimate the WCET considering an already known feasible path obtained from the semantics. Our approach does both the infeasible path analysis and the TDMA model in the same step. Using an SMT expression allows our approach to consider all feasible path without having to enumerate them individually. Our approach is more precise, but more costly. It can be used in a complementary way with other approaches in a trade-off between quality of the results and analysis time. We discuss possible approaches for loop analysis in Section 6.2.

# 6. CONCLUSION

## 6.1 Summary

We introduce a new approach for WCET analysis of shared TDMA bus using Satisfiability Modulo Theory (SMT). This approach takes into account the semantics and the accesses to a shared TDMA bus to give a tighter estimation of the execution time. In our proof-of-concept, we consider a platform without cache memory which means that all *load* and *store* instructions access the shared bus. We also analyze programs in the form of LLVM bitcode due to the constraints imposed by the tool PAGAI. This is a limitation of our implementation, but not of the approach itself: the same approach can be applied to executable binaries given a generated model in SMT and with the presence of cache memory. Accesses to the bus can be obtained from an analysis of the cache's state where a cache miss is considered as an access to the shared memory through the shared bus.

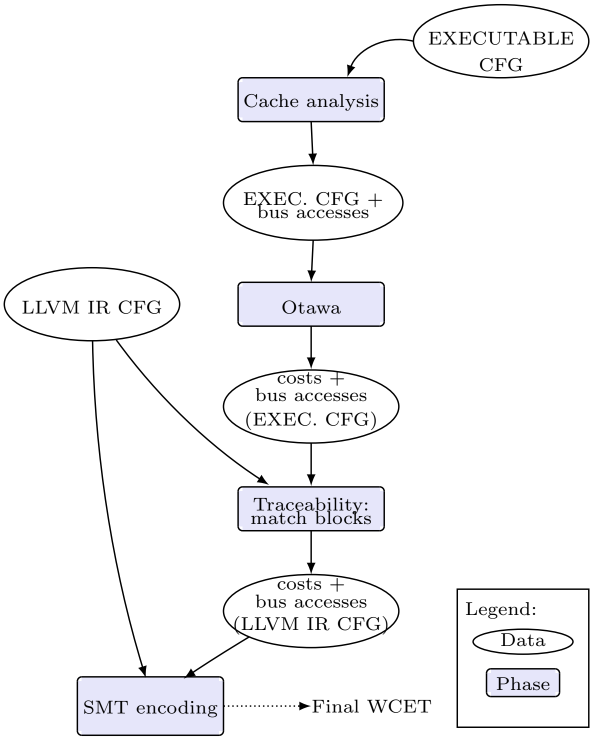


Figure 12: General work flow for realistic timing analysis

The naive model of the TDMA bus shows poor performance. To overcome the issue, we propose an offset based model. The micro-benchmarks show a better scalability but remains exponential. The added cuts on the offsets improve the analysis time by indicating to the SMT-solver "obvious" properties.

Finally, we show that the micro-architectural analysis of the shared TDMA bus, and the semantic analysis can be combined in one approach using an SMT model. This approach can achieve a more precise estimation of the WCET in presence of a shared TDMA bus. The naive encodings are very costly. We give alternative encodings that reduce considerably the solving time of the SMT expression.

## 6.2 Future Work

The current implementation of our approach is a proof of concept to check its viability and scalability. As such, taking into account a realistic model for the timing of the program is left to future works. Considering that each LLVM instruction takes exactly one cycle is clearly not realistic: the timing for each block should instead come from a micro-architectural analysis of the actual binary with a tool like OTAWA [2]. Keeping the analysis itself on LLVM bitcode allows exploiting high-level properties of the program that would be lost at the binary code level, and the SSA form of the bitcode greatly simplifies the encoding into SMT. As a consequence, a complete tool for a realistic analysis would need to work both on the binary code and the LLVM bitcode. The information obtained on the binary must be mapped to the LLVM bitcode. One solution to achieve this is through pattern matching of conditions [4] between the LLVM CFG and the executable CFG. The overall approach for such an information flow is described in Figure 12. It has already been applied to SMT-based WCET analysis in [9]. The idea of combining high-level semantic information with low-level binary analysis has also already been applied in e.g. [12, 15].

Similarly, considering LLVM *load* and *store* operations as bus accesses is an oversimplification. Some LLVM *load* and *store* will actually be cache hits and will not access the bus, and conversely, some operations on LLVM registers will actually need to access the memory in the real program. The actual bus accesses must therefore be obtained by a prior cache analysis on the binary code [1].

Our experiments show scalability issues which is expected in NP-complete problems. We are considering optimizations and improvements in the scalability in future work. Our approach already shows substantial improvements over a naive encoding, and the results show that we do scale to reasonably-sized programs. Still, we would probably encounter performance issues in the SMT solver to scale if we try to analyze very large case-studies globally with this approach. We therefore need an approach that uses our analysis on reasonably-sized pieces of code extracted from a possibly larger codebase. One option is to analyze the program in portions and propagate the obtained results on a global analysis. For example, considering only a small piece of code surrounding a bus access may be sufficient to prove that this access is in the TDMA slot (or to prove a tight bound on its execution time), and this information can be injected in a global cheaper analysis. The challenge here is how one defines the analyzed portions and their sizes.

Loops with a large iteration count, which cannot be unrolled completely, could be handled using partial unrolling with an unroll factor. Loop iterations are then analyzed separately with updated information on offsets between each iteration. Kelter et al. [11] already address the loop analysis with minimum unrolling. The SMT-based approach can be complementary to include the semantics in the loop body analysis.

# 7. REFERENCES

[1] M. Alt, C. Ferdin, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Science of Computer Programming*, pages 52–66. Springer, 1996.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In S. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer Berlin Heidelberg, 2010.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The auspicious couple: Symbolic execution and WCET analysis. *WCET*, 30:53–63, 2013.

[5] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, USA, 2010. ACM.

[6] R. C. David. *The SMT-LIBv2 Language and Tools: A Tutorial*, March 2013.

[7] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In B. Lisper, editor, *WCET 2010*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 101–112. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[9] J. Henry, M. Asavoae, D. Monniaux, and C. Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 43–52, New York, NY, USA, 2014. ACM.

[10] J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.*, 289:15–25, Dec. 2012.

[11] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Syst.*, 50(2):185–229, Mar. 2014.

[12] H. Li, I. Puaut, and E. Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 97. ACM, 2014.

[13] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, Washington, DC, USA, 2010. IEEE Computer Society.

[14] P. Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *International Conference on Embedded Software (EMSOFT 2014)*, New Dehli, India, oct 2014.

[15] P. Raymond, C. Maiza, C. Parent-Vigouroux, F. Carrier, and M. Asavoae. Timing analysis enhancement for synchronous program. *Real-Time Systems*, pages 1–29, 2015.

[16] J. Reineke and R. Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *WCET 2009*, page 101, 2009.

[17] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. *WCET*, 4, 2006.

[18] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. RTAS '10, pages 215–224, Washington, DC, USA, 2010. IEEE Computer Society.